The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator

Huaicheng Li, Mingzhe Hao, and Michael Hao Tong, *University of Chicago*;
Swaminatahan Sundararaman, *Parallel Machines*; Matias Bjørling, *CNEX Labs*;
Haryadi S. Gunawi, *University of Chicago*

https://www.usenix.org/conference/fast18/presentation/li

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

Open access to the Proceedings of
the 16th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator

Huaicheng Li, Mingzhe Hao, Michael Hao Tong,
Swaminatahan Sundararaman[†], Matias Bjørling[‡], Haryadi S. Gunawi

University of Chicago [†]Parallel Machines [‡]CNEX Labs

ABSTRACT: *FEMU is a software (QEMU-based) flash emulator for fostering future full-stack software/hardware SSD research. FEMU is cheap (open-sourced), relatively accurate (0.5-38% variance as a drop-in replacement of OpenChannel SSD), scalable (can support 32 parallel channels/chips), and extensible (support internal-only and split-level SSD research).*

1 Introduction

Cheap and extensible research platforms are a key ingredient in fostering wide-spread SSD research. SSD simulators such as DiskSim's SSD model [9], FlashSim [13] and SSDSim [16], despite their popularity, only support internal-SSD research but not kernel-level extensions. On the other hand, hardware research platforms such as FPGA boards [28, 34, 46], OpenSSD [7], or OpenChannel SSD [11], support full-stack software/hardware research but their high costs (thousands of dollars per device) impair large-scale SSD research.

This leaves software-based emulator such as QEMU-based VSSIM [45], FlashEm [47], and LightNVM's QEMU [6], as the cheap alternative platform. Unfortunately, the state of existing emulators is bleak; they are either outdated, non-scalable, or not open-sourced.

We argue that it is a critical time for storage research community to have a new software-based emulator (more in §2). To this end, we present FEMU, a QEMU-based flash emulator, with the following four "CASE" benefits.

First, FEMU is **c**heap ($0) as it will be an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences [14, 43]. We hope FEMU will be useful to broader communities.

Second, FEMU is (relatively) **a**ccurate. For example, FEMU can be used as a drop-in replacement for OpenChannel SSD; thus, future research that extends LightNVM [11] can be performed on top of FEMU with relatively accurate results (*e.g.*, 0.5-38% variance in our tests). With FEMU, prototyping SSD-related kernel changes can be done without a real device.

Third, FEMU is **s**calable. As we optimized the QEMU stack with various techniques, such as exitless interrupt and skipping QEMU AIO components, FEMU can scale to 32 IO threads and still achieve a low latency (as low as 52µs under a 2.3GHz CPU). As a result, FEMU can accurately emulate 32 parallel channels/chips, without unintended queueing delays.

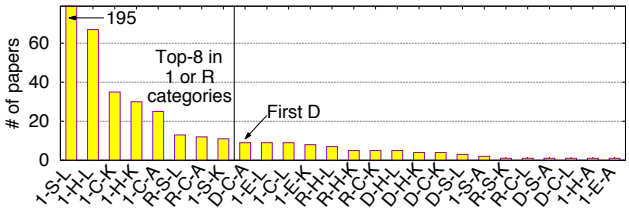

Figure 1: **Categorization of SSD research.** *The figure is explained in Section §2.1. The first bar reaches 195 papers.*

Finally, FEMU is **e**xtensible. Being a QEMU-based emulator, FEMU can support internal-SSD research (only FEMU layer modification), kernel-only research such as software-defined flash (only Guest OS modification on top of unmodified FEMU), and split-level research (both Guest OS and FEMU modifications). FEMU also provides many new features not existent in other emulators, such as OpenChannel and multi-device/RAID support, extensible interfaces via NVMe commands, and page-level latency variability.

2 Extended Motivation

2.1 The State of SSD Research Platforms: We reviewed 391 papers in more than 30 major systems and storage conferences and journals published in the last 10 years, and categorized them as follows:

1. What was the **scale** of the research? **[1]:** single SSD; **[R]:** RAID of SSDs (flash array); or **[D]:** distributed/multi-node SSDs.
2. What was the **platform** being used? **[C]:** commodity SSDs; **[E]:** software SSD emulators (VSSIM [45] or FlashEm [47]); **[H]:** hardware platforms (FPGA boards, OpenSSD [7], or OpenChannel SSD [6]); or **[S]:** trace-based simulators (DiskSim+SSD [9] or FlashSim [13] and SSDSim [16]).

3. What **layer** was modified? **[A]:** application layer; **[K]:** OS kernel; **[L]:** low-level SSD controller logic.

Note that some papers can fall into two sub-categories (*e.g.*, modify both the kernel and the SSD logic). Figure 1 shows the sorted order of the combined categories. For example, the most popular category is **1-S-L**, where 195 papers target only single SSD (**1**), use simulator (**S**), and modify the low-level SSD controller logic (**L**). However, simulators do not support running applications and operating systems.

2.2 The Lack of Large-Scale SSD Research: Our first motivation is the lack of papers in the distributed SSDs category (**D-**...), for example, for investigating the impact of SSD-related changes to distributed computing and graph frameworks. One plausible reason is the cost of managing hardware (procurement, installation, maintenance, etc.). The top-8 categories in Figure 1, a total of 324 papers (83%), target single SSD (**1-**...) and flash array (**R-**...). The highest **D** category is **D-C-A** (as highlighted in the figure), where only 9 papers use commodity SSDs (**C**) and modify the application layer (**A**). The next **D** category is **D-H-L**, where hardware platforms (**H**) are used for modifying the SSD controller logic (**L**). Unfortunately, most of the 6 papers in this category are from large companies with large research budget (*e.g.*, FPGA usage in Baidu [28] and Tencent [46]). Other hardware platforms such as OpenSSD [7] and OpenChannel SSD [6] also cost thousands of dollars each, impairing multi-node non-simulation research, especially in academia.

2.3 The Rise of Software-Defined Flash: Today, research on host-managed (aka. "software-defined" or "user-programmable") flash is growing [25, 28, 34, 35, 41, 46]. However, such research is mostly done on top of expensive and hard-to-program FPGA platforms. Recently, a more affordable and simpler platform is available, OpenChannel SSD [6], managed by Linux-based LightNVM [11]. Before its inception (2015), there were only 24 papers that performed kernel-only changes, since then, 11 papers have been published, showing the success of OpenChannel SSD.

However, there remains several issues. First, not all academic communities have budget to purchase such devices. Even if they do, while prototyping the kernel/application, it is preferable not to write too much to and wear out the device. Thus, replacing OpenChannel SSD (during kernel prototyping) with a software-based emulator is desirable.

2.4 The Rise of Split-Level Architecture: While most existing research modify a single layer (application/kernel/SSD), some recent works show the benefits of "split-level" architecture [8, 19, 24, 38, 42], wherein some functionalities move up to the OS kernel (**K**) and some other move down to the SSD firmware (**L**) [18, 31, 36]. So far, we found only 40 papers in split-level **K+L** category (*i.e.*, modify *both* the kernel and SSD logic layers), mostly done by companies with access to SSD controllers [19] or academic researchers with Linux+OpenSSD [21, 32] or with block-level emulators (*e.g.*, Linux+FlashEm) [29, 47]. OpenSSD with its single-threaded, single-CPU, whole-blocking GC architecture also has many known major limitations [43]. FlashEm also has limitations as we elaborate more below. Note that the kernel-level LightNVM is not a suitable platform for split-level research (*i.e.*, support **K**, but not **L**). This is because its SSD layer (*i.e.*, OpenChannel SSD) is not modifiable; the white-box part of OpenChannel SSD is the exposure of its internal channels and chips to be managed by software (Linux LightNVM), but the OpenChannel firmware logic itself is a black-box part.

2.5 The State of Existing Emulators: We are only aware of three popular software-based emulators: FlashEm, LightNVM's QEMU and VSSIM.

FlashEm [47] is an emulator built in the Linux block level layer, hence less portable; it is rigidly tied to its Linux version; to make changes, one must modify Linux kernel. FlashEm is not open-sourced and its development stopped two years ago (confirmed by the creators).

LightNVM's QEMU platform [6] is still in its early stage. Currently, it cannot emulate multiple channels (as in OpenChannel SSD) and is only used for basic testing of 1 target (1 chip behind 1 channel). Worse, LightNVM's QEMU performance is not scalable to emulate NAND latencies as it depends on vanilla QEMU NVMe interface (as shown in the NVMe line in Figure 2a).

VSSIM [45] is a QEMU/KVM-based platform that emulates NAND flash latencies on a RAM disk, and has been used in several papers. The major drawback of VSSIM is that it is built within QEMU's IDE interface implementation, which is not scalable. The upper-left red line (IDE line) in Figure 2a shows the user-perceived IO read latency through VSSIM without any NAND-delay emulation added. More concurrent IO threads (x-axis) easily multiply the average IO latency (y-axis). For example from 1 to 4 IO threads, the average latency spikes up from 152 to 583µs. The root cause is that IDE is not supported with virtualization optimizations.

With this drawback, emulating internal SSD parallelism is a challenge. VSSIM worked around the problem by only emulating NAND delays in another background thread in QEMU, disconnected from the main IO path. Thus, for multi-threaded applications, to collect accurate results, users solely depend on VSSIM's monitoring tool [45, Figure 3], which monitors the IO latencies emulated in the background thread. In other words, users

cannot simply time the multi-threaded applications (due to IDE poor scalability) at the user level.

Despite these limitations, we (and the community) are *greatly indebted* to VSSIM authors as VSSIM provides a base design for future QEMU-based SSD emulators. As five years have passed, it is time to build a new emulator to keep up with the technology trends.

3 FEMU

We now present FEMU design and implementation. FEMU is implemented in QEMU v2.9 in 3929 LOC and acts as a virtual block device to the Guest OS. A typical software/hardware stack for SSD research is {Application+Host OS+SSD device}. With FEMU, the stack is {Application+Guest OS+FEMU}. The LOC above excludes base OC extension structures from LightNVM's QEMU and FTL framework from VSSIM.

Due to space constraints, we omit the details of how FEMU works inside QEMU (*e.g.*, FEMU's FTL and GC management, IO queues), as they are similarly described in VSSIM paper [45, Section 3]. We put them in FEMU release document [1]. In the rest of the paper, we focus on the main challenges of designing FEMU: achieving scalability (§3.1) and accuracy (§3.2) and increasing usability and extensibility (§3.3).

Note that all latencies reported here are user-perceived (application-level) latencies on memory-backed virtual storage and 24 dual-thread (2x) CPU cores running at 2.3GHz. According to our experiments, the average latency is inversely proportional to CPU frequency, for example, QEMU NVMe latency under 1 IO thread is 35µs on a 2.3GHZ CPU and 23µs on a 4.0GHz CPU.

3.1 Scalability

Scalability is an important property of a flash emulator, especially with high internal parallelism of modern SSDs. Unfortunately, stock QEMU exhibits a scalability limitation. For example, as shown in Figure 2a, with QEMU NVMe (although it is more scalable than IDE), more IO threads still increases the average IO latency (*e.g.*, with 8 IO threads, the average IO latency already reaches 106µs). This is highly undesirable because typical read latency of modern SSDs can be below 100µs.

More scalable alternatives to NVMe are virtio and dataplane (dp) interfaces [3, 30] (`virtio`/`dp` vs. `NVMe` lines in Figure 2a). However, these interfaces are not as extensible as NVMe (which is more popular). Nevertheless, virtio and dp are also not scalable enough to emulate low flash latencies. For example, at 32 IO threads, their IO latencies already reach 185µs and 126µs, respectively.

Problems: Collectively, all of the scalability bottlenecks above are due to two reasons: **(1)** QEMU uses a traditional trap-and-emulate method to emulate IOs. The

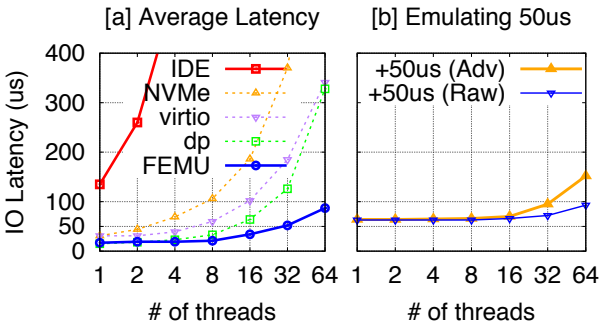

Figure 2: **QEMU Scalability.** *The figure shows the scalability of QEMU's IDE, NVMe, virtio, and dataplane (dp) interface implementations, as well as FEMU. The x-axis represents the number of concurrent IO threads running at the user level. Each thread performs random 4KB read IOs. The y-axis shows the user-perceived average IO latency. For Figure (a), the IDE and NVMe lines representing VSSIM and LightNVM's QEMU respectively are discussed in §2.5; virtio, dp, and FEMU lines in §3.1. For Figure (b), the "+50µs (Raw)" line is discussed in §3.2.1; the "+50µs (Adv)" line in "Result 3" part of §3.2.3.*

Guest OS' NVMe driver "rings the doorbell [5]" to the device (QEMU in our case) that some IOs are in the device queue. This "doorbell" is an MMIO operation that will cause an expensive VM-exit ("world switch" [39]) from the Guest OS to QEMU. A similar operation must also be done upon IO completion. **(2)** QEMU uses asynchronous IOs (AIO) to perform the actual read/write (byte transfer) to the backing image file. This AIO component is needed to avoid QEMU being blocked by slow IOs (*e.g.*, on a disk image). However, the AIO overhead becomes significant when the storage backend is a RAM-backed image.

Our solutions: To address these problems, we leverage the fact that FEMU purpose is for research prototyping, thus we perform the following modifications:

(1) We transform QEMU from an interrupt- to a polling-based design and disable the doorbell writes in the Guest OS (just 1 LOC commented out in the Linux NVMe driver). We create a dedicated thread in QEMU to continuously poll the status of the device queue (a shared memory mapped between the Guest OS and QEMU). This way, the Guest OS still "passes" control to QEMU but without the expensive VM exits. We emphasize that FEMU can still work without the changes in the Guest OS as we report later. This optimization can be treated as an optional feature, but the 1 LOC modification is extremely simple to make in many different kernels.

(2) We do not use virtual image file (in order to skip the AIO subcomponent). Rather, we create our own RAM-backed storage in QEMU's heap space (with configurable size `malloc()`). We then modify QEMU's DMA emulation logic to transfer data from/to our heap-

backed storage, transparent to the Guest OS (*i.e.*, the Guest OS is not aware of this change).

Results: The bold `FEMU` line in Figure 2a shows the scalability achieved. In between 1-32 IO threads, FEMU can keep IO latency stable in less than 52µs, and even below 90µs at 64 IO threads. If the single-line Guest-OS optimization is not applied (the removal of VM-exit), the average latency is 189µs and 264µs for 32 and 64 threads, respectively (not shown in the graph). Thus, we recommend applying the single-line change in the Guest OS to remove expensive VM exits.

The remaining scalability bottleneck now only comes from QEMU's *single-thread* "event loop" [4, 15], which performs the main IO routine such as dequeueing the device queue, triggering DMA emulations, and sending end-IO completions to the Guest OS. Recent works addressed these limitations (with major changes) [10, 23], but have not been streamlined into QEMU's main distribution. We will explore the possibility of integrating other solutions in future development of FEMU.

3.2 Accuracy

We now discuss the accuracy challenges. We first describe our delay mechanism (§3.2.1), followed by our basic and advanced delay models (§3.2.2-3.2.3).

3.2.1 Delay Emulation

When an IO arrives, FEMU will issue the DMA read/write command, then label the IO with an emulated completion time (T_{endio}) and add the IO to our "end-io queue," sorted based on IO completion time. FEMU dedicates an "end-io thread" that continuously takes an IO from the head of the queue and sends an end-io interrupt to the Guest OS, once the IO's emulated completion time has passed current time (T_{endio}>T_{now}).

The "+50us (Raw)" line in Figure 2b shows a simple (and stable) result where we add a delay of 50µs to *every* IO (T_{endio}=T_{entry}+50µs). Note that the end-to-end IO time is more than 50µs because of the Guest OS overhead (roughly 20µs). Important to say that FEMU also does not introduce severe latency tail. In the experiment above, 99% of all the IOs are stable at 70µs. Only 0.01% (99.99^{th} percentile) of the IOs exhibit latency tail of more than 105µs, which already exists in stock QEMU. For example, in VSSIM, the 99^{th}-percentile latency is already over 150µs.

3.2.2 Basic Delay Model

The challenge now is to compute the end-io time (T_{endio}) for every IO accurately. We begin with a basic delay model by marking every plane and channel with their next free time (T_{free}). For example, if a page write arrives to currently-free channel #1 and plane #2, then we will advance the channel's next free time

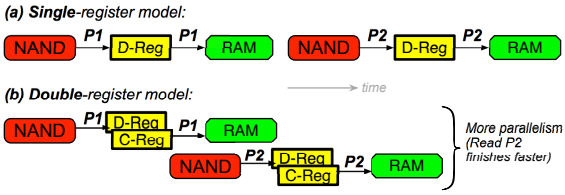

Figure 3: **Single- vs. double-register model.** *(a) In a single-register model, a plane only has one data register (D-Reg). Read of page P2 cannot start until P1 finishes using the register (i.e., the transfer to the controller's RAM completes). (b) In a double-register model, after P1 is read to the data register, it is copied quickly to the cache register (D-Reg to C-Reg). As the data register is free, read of P2 can begin (in parallel with P1's transfer to the RAM), hence finishes faster.*

($T_{freeOfChannel1}$=T_{now}+$T_{transfer}$, where $T_{transfer}$ is a configurable page transfer time over a channel) and the plane's next free time ($T_{freeOfPlane2}$+=T_{write}, where T_{write} is a configurable write/programming time of a NAND page). Thus, the end-io time of this write operation will be T_{endio}=$T_{freeOfPlane2}$.

Now, let us say a page read to the same plane arrives while the write is ongoing. Here, we will advance $T_{freeOfPlane2}$ by T_{read}, where T_{read} is a configurable read time of a NAND page, and $T_{freeOfChannel1}$ by $T_{transfer}$. This read's end-io time will be T_{endio}=$T_{freeOfChannel1}$ (as this is a read operation, not a write IO).

In summary, this basic *queueing* model represents a *single-register* and *uniform page latency* model. That is, every plane only has a single page register, hence cannot serve multiple IOs in parallel (*i.e.*, a plane's T_{free} represents IO serialization in that plane) and the NAND page read, write, and transfer times (T_{read}, T_{write} and $T_{transfer}$) are all *single* values. We also note that GC logic can be easily added to this basic model; a GC is essentially a series of reads/writes (and erases, T_{erase}) that will also advance plane's and channel's T_{free}.

3.2.3 Advanced "OC" Delay Model

While the model above is sufficient for basic comparative research (*e.g.*, comparing different FTL/GC schemes, some researchers might want to emulate the detailed intricacies of modern hardware. Below, we show how we extend our model and achieve a more accurate delay emulation of OpenChannel SSD ("**OC**" for short).

The OC's NAND hardware has the following intricacies. First, OC uses *double-register* planes; every plane is built with two registers (data+cache registers), hence a NAND page read/write in a plane can overlap with a data transfer via the channel to the plane (*i.e.*, more parallelism). Figure 3 contrasts the single- vs. double-register models where the completion time of the second IO to page P2 is faster in the double-register model.

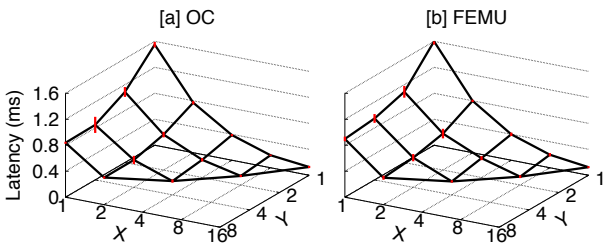

Figure 4: **OpenChannel SSD (OC) vs. FEMU.** *X: # of channels, Y: # of planes per channel. The figures are described in the "Result 1" segment of Section 3.2.3.*

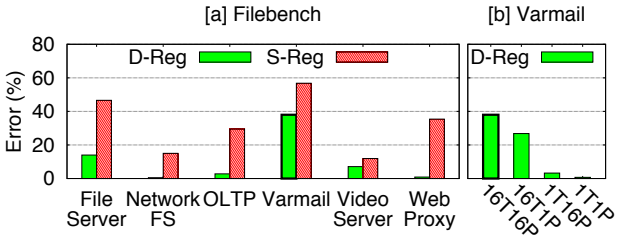

Figure 5: **Filebench on OpenChannel SSD (OC) vs. FEMU.** *The figures are described in the "Result 2" segment of Section 3.2.3. The y-axis shows the latency difference (error) of the benchmark results on OC vs. FEMU ($Error=(Lat_{femu}-Lat_{oc})/Lat_{oc}$). D-Reg and S-Reg represent the advanced and basic model respectively. The two bars with bold edge in Figures (a) and (b) are the same experiment and configuration (varmail with 16 threads on 16 planes).*

Second, OC uses a *non-uniform* page latency model; that is, pages that are mapped to upper bits of MLC cells ("upper" pages) incur higher latencies than those mapped to lower bits ("Lower" pages); for example 48/64µs for lower/upper-page read and 900/2400µs for lower/upper-page write. Making it more complex, the 512 pages in each NAND block are not mapped in a uniformly interleaving manner as in "`LuLuLuLu...`", but rather in a specific way, "`LLLLLLuLLuLLuu...`", where pages #0-6 and #8-9 are mapped to Lower pages, pages #7 and #10 to upper pages, and the rest ("...") have a repeating pattern of "`LLuu`".

Results: By incorporating this detailed model, FEMU can act as an accurate drop-in replacement of OC, which we demonstrate with the following results.

Result 1: Figure 4 compares the IO latencies on OC vs. FEMU. The workload is 16 IO threads performing random reads uniformly spread throughout the storage space. We map the storage space to different configurations. For example, x=1 and y=1 implies that OC and FEMU are configured with only 1 channel and 1 plane/channel, thus as a result, the average latency is high (z>1550µs) as all the 16 concurrent reads are contending for the same plane and channel. The result for x=16 and y=1 implies that we use 16 channels with 1 plane/channel (a total of 16 planes). Here, the concurrent reads are absorbed in parallel by all the planes and channels, hence a faster average read latency (z<130µs). Overall, Figures 4a and 4b exhibit a highly similar pattern, showing the success of our queuing delay emulation. The latency difference (error) is only between 0.8-11.6%; $Error=(Lat_{femu}-Lat_{oc})/Lat_{oc}$.

Result 2: Figure 5a shows the results from running several macrobenchmarks with six filebench personalities, with 16 IO threads of concurrent reads/writes on 16 planes across 4 channels. The figure only shows the latency difference ($Error$) which contrasts the accuracy of our basic and advanced delay models. With the basic model, the resulting latencies are highly inaccurate (12-57%), but with the advanced model, the error drops to only 0.5-38%, which are 1.5-40× more accurate across the six benchmarks.

We believe that these errors are reasonable as we deal with delay emulation of tens of µs granularity. We leave further optimization for future work; we might have missed other OC intricacies that should be incorporated into our advanced model (as explained at the end of §2.4, OC only exposes channels and chips, but other details are not exposed by the vendor). Nevertheless, we investigate further the residual errors, as shown in Figure 5b. Here, we use the `varmail` personality but we vary the #IO threads [T] and #planes [P]. For example, in the 16 threads on 16 planes configuration (x="16T16P" in Figure 5b, which is the same configuration used in experiments in Figure 5a), the error is 38%. However, the error decreases in less complex configurations (*e.g.*, 0.7% error with single thread on single plane). Thus, higher errors come from more complex configurations (*e.g.*, more IO threads and more planes), which we explain next.

Result 3: We find that using an advanced model requires more CPU computation, and this compute overhead will backlog with higher thread count. To show this, Figure 2b compares the simple +50µs delay emulation in our raw implementation (§3.2.1) vs. advanced model. Here, both cases simply add +50µs, but the advanced model must traverse many `if-else` statements (to check register, plane, and channel next free time), hence the compute overhead. Further scalability optimizations, as discussed at the end of §3.1 can help.

3.3 Usability and Extensibility

Being a *software*-based emulation platform, FEMU can be extended in many different ways. We now describe existing features/usabilities of FEMU, briefly showcase successful extensions used in our recent work [14, 43] as well as possible future work that FEMU features enable.

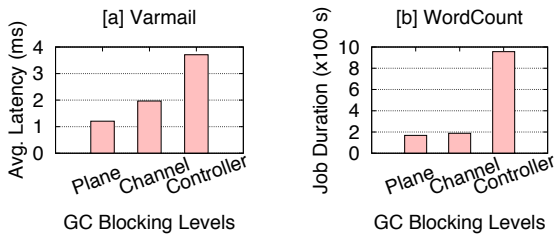

Figure 6: **Use examples.** *Figure 6a is described in the "FTL and GC schemes" segment of Section 3.3. Figure 6b is discussed in the "Distributed SSDs" segment of Section 3.3.*

• **FTL and GC schemes:** In default mode, our FTL employs a dynamic mapping and a channel-blocking GC as used in other simulators [9, 16]. One of our projects uses FEMU to compare different GC schemes: controller, channel, and plane blocking [43]. In controller-blocking GC, a GC operation "locks down" the controller, preventing any foreground IOs to be served (as in OpenSSD [7]). In channel-blocking GC, only channels involved in GC page movement are blocked (as in SSDSim [16]). In plane-blocking GC, the most efficient one, page movement only flows within a plane without using any channel (*i.e.*, "copyback" [2]). Sample results are shown in Figure 6a. Beyond our work, recent works also show the benefits of SSD partitioning for performance isolation [11, 17, 22, 27, 37], which are done on either a simulator or a hardware platform. More partitioning schemes can also be explored with FEMU.

• **White-box vs. black-box mode:** FEMU can be used as **(1)** a white-box device such as OpenChannel SSD where the device exposes physical page addresses and the FTL is managed by the OS such as in Linux LightNVM or **(2)** a black-box device such as commodity SSDs where the FTL resides inside FEMU and only logical addresses are exposed to the OS.

• **Multi-device support for flash-array research:** FEMU is configurable to appear as multiple devices to the Guest OS. For example, if FEMU exposes 4 SSDs, inside FEMU there will be 4 separate NVMe instances and FTL structures (with no overlapping channels) managed in a single QEMU instance. Previous emulators (VSSIM and LightNVM's QEMU) do not support this.

• **Extensible OS-SSD NVMe commands:** As FEMU supports NVMe, new OS-to-SSD commands can be added (*e.g.*, for host-aware SSD management or split-level architecture [31]). For example, currently in LightNVM, a GC operation reads valid pages from OC to the host DRAM and then writes them back to OC. This wastes host-SSD PCIe bandwidth; LightNVM foreground throughput drops by 50% under a GC. Our conversation with LightNVM developers suggests that one can add a new "`pageMove fromAddr toAddr`" NVMe command from the OS to FEMU/OC such that the data movement does not cross the PCIe interface. As mentioned earlier, split-level architecture is trending [12, 20, 29, 40, 44] and our NVMe-powered FEMU can be extended to support more commands such as transactions, deduplication, and multi-stream.

• **Page-level latency variability:** As discussed before (§3.2), FEMU supports page-level latency variability. Among SSD engineers, it is known that "not all chips are equal." High quality chips are mixed with lesser quality chips as long as the overall quality passes the standard. Bad chips can induce more error rates that require longer, repeated reads with different voltages. FEMU can also be extended to emulate such delays.

• **Distributed SSDs:** Multiple instances of FEMU can be easily deployed across multiple machines (as simple as running Linux hypervisor KVMs), which promotes more large-scale SSD research. For example, we are also able to evaluate the performance of Hadoop's word-count workload on a cluster of machines running FEMU, but with different GC schemes as shown in Figure 6b. Since HDFS uses large IOs, which will eventually be striped across many channels/planes, there is a smaller performance gap between channel and plane blocking. We hope FEMU can spur more work that modifies the SSD layer to speed up distributed computing frameworks (*e.g.*, distributed graph processing frameworks).

• **Page-level fault injection:** Beyond performance-related research, flash reliability research [26, 33] can leverage FEMU as well (*e.g.*, by injecting page-level corruptions and faults and observing how the high-level software stack reacts).

• **Limitations:** FEMU is DRAM-backed, hence cannot emulate large-capacity SSDs. Furthermore, for crash consistency research, FEMU users must manually emulate "soft" crashes as hard reboots will wipe out the data in the DRAM. Also, as mentioned before (§3.2), there is room for improving accuracy.

4 Conclusion & Acknowledgments

As modern SSD internals are becoming more complex, their implications to the entire storage stack should be investigated. In this context, we believe FEMU is a fitting research platform. We hope that our cheap and extensible FEMU can speed up future SSD research.

We thank Sam H. Noh, our shepherd, and the anonymous reviewers for their tremendous feedback. This material was supported by funding from NSF (grant Nos. CNS-1526304 and CNS-1405959).

References

[1] https://github.com/ucare-uchicago/femu.

[2] Using COPYBACK Operations to Maintain Data Integrity in NAND Flash Devices. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2941_idm_copyback.pdf, 2008.

[3] Towards Multi-threaded Device Emulation in QEMU. KVM Forum, 2014.

[4] Improving the QEMU Event Loop. KVM Forum, 2015.

[5] NVMe Specification 1.3. http://www.nvmexpress.org, 2017.

[6] Open-Channel Solid State Drives. http://lightnvm.io, 2017.

[7] The OpenSSD Project. http://openssd.io, 2017.

[8] Violin Memory. All Flash Array Architecture, 2017.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[10] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[11] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[12] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software Orchestrated Flash Array. In *The 7th Annual International Systems and Storage Conference (SYSTOR)*, 2014.

[13] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[14] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[15] Nadav Har'El, Nadav, Gordon, Abel, Landau, Alex, Ben-Yehuda, Muli, Traeger, Avishay, Ladelsky, and Razya. Efficient and Scalable Paravirtual I/O System. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[16] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, 2011.

[17] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[18] Xavier Jimenez and David Novo. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST)*, 2014.

[19] William K. Josephson, Lars A. Bongo, David Flynn, Fusion-io, and Kai Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[20] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[21] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[22] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[23] Tae Yong Kim, Dong Hyun Kang, Dongwoo Lee, and Young Ik Eom. Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework. In *Proceedings of the 31st IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2015.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[25] Sungjin Lee, Ming Liu, Sang Woo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[26] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[27] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[28] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[29] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[30] Rusty Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. In *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[31] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[32] Mohit Saxena, Yiying Zhang, Michael M. Swift, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[33] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[34] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[35] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[36] Liang Shi, Kaijie Wu, Mengying Zhao, Chun Jason Xue, Duo Liu, and Edwin H.-M. Sha. Retention Trimming for Lifetime Improvement of Flash Memory Storage Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(1), January 2016.

[37] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[38] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[39] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2001.

[40] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas, Kornilios Kourtis, and Thomas R. Gross. FlashNet: Flash/Network Stack Co-design. In *The 10th Annual International Systems and Storage Conference (SYSTOR)*, 2017.

[41] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[42] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[43] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[44] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic Stream Management for Multi-streamed SSDs. In *The 10th Annual International Systems and Storage Conference (SYSTOR)*, 2017.

[45] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[46] Jianquan Zhang, Dan Feng, Jianlin Gao, Wei Tong, Jingning Liu, Yu Hua, Yang Gao, Caihua Fang, Wen Xia, Feiling Fu, and Yaqing Li. Application-Aware and Software-Defined SSD Scheme for Tencent Large-Scale Storage System. In *Proceedings of 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[47] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.