The UNIVERSITY OF CHICAGO

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

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY

Huaicheng Li

CHICAGO, ILLINOIS
2018
# Table of Contents

LIST OF FIGURES ............................................................... VI
LIST OF TABLES .............................................................. VIII
ACKNOWLEDGMENTS .......................................................... IX
ABSTRACT .............................................................................. X

1 INTRODUCTION ................................................................. 1
   1.1 SSD RESEARCH PLATFORM INTRODUCTION ......................... 1
   1.2 EXTENDED MOTIVATION .................................................. 4
       1.2.1 THE STATE OF SSD RESEARCH PLATFORMS: ................. 4
       1.2.2 THE LACK OF LARGE-SCALE SSD RESEARCH .............. 5
       1.2.3 THE RISE OF SOFTWARE-DEFINED FLASH .................... 5
       1.2.4 THE RISE OF SPLIT-LEVEL ARCHITECTURE ............... 6
       1.2.5 THE STATE OF EXISTING EMULATORS ...................... 7

2 FEMU DESIGN ...................................................................... 9

3 FEMU SCALABILITY ............................................................ 12
   3.1 QEMU I/O SCALABILITY ............................................... 13
   3.2 QEMU I/O SCALABILITY PROBLEM ROOT CAUSES ............... 15
       3.2.1 NVMe PRIMER ......................................................... 16
       3.2.2 QEMU NVMe VIRTUALIZATION & SCALABILITY ROOT CAUSES . 17
   3.3 SCALABILITY SOLUTIONS ............................................... 20
       3.3.1 POLLING BASED QEMU NVMe DESIGN .................. 20
       3.3.2 CUSTOMIZED QEMU AIO PATH ............................. 21
       3.3.3 FEMU OPTIMIZATION RESULTS ............................... 23

4 FEMU ACCURACY .................................................................. 24
   4.1 DELAY EMULATION ...................................................... 24
       4.1.1 “UNSUCCESSFUL” DELAY EMULATION EFFORTS .......... 25
       4.1.2 ENDO DELAY EMULATION ...................................... 26
       4.1.3 ENDO DELAY EMULATION RESULTS ....................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>SSD PERFORMANCE MODELS</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1</td>
<td>BASIC DELAY MODEL</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2</td>
<td>ADVANCED “OC” DELAY MODEL</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>FEMU ACCURACY RESULTS</td>
<td>31</td>
</tr>
<tr>
<td>4.3.1</td>
<td>WORKLOADS &amp; EXPERIMENT SETUPS</td>
<td>31</td>
</tr>
<tr>
<td>4.3.2</td>
<td>MICROBENCH RESULTS</td>
<td>32</td>
</tr>
<tr>
<td>4.3.3</td>
<td>MACROBENCH RESULTS</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>FEMU USE CASE</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>USABILITY</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>FEMU EXTENSIBILITY</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>OTHER WORK</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>MITT (MILLISECOND TAIL TOLERANCE) INTRODUCTION</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>SSD MANAGEMENT (MITTSSD)</td>
<td>42</td>
</tr>
<tr>
<td>6.3</td>
<td>EVALUATION</td>
<td>44</td>
</tr>
<tr>
<td>6.3.1</td>
<td>MICROBENCHMARK RESULTS</td>
<td>45</td>
</tr>
<tr>
<td>6.3.2</td>
<td>MITTSSD RESULTS WITH AMAZON EC2 NOISE</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>PREDICTION ACCURACY</td>
<td>47</td>
</tr>
<tr>
<td>7</td>
<td>CONCLUSION</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>FEMU LIMITATIONS &amp; FUTURE WORK</td>
<td>49</td>
</tr>
<tr>
<td>8.1</td>
<td>FEMU LIMITATIONS</td>
<td>49</td>
</tr>
<tr>
<td>8.2</td>
<td>FUTURE WORK ON SCALABILITY</td>
<td>49</td>
</tr>
<tr>
<td>8.3</td>
<td>FUTURE WORK ON FINE-GRANULAR FTL FEATURES</td>
<td>50</td>
</tr>
<tr>
<td>8.4</td>
<td>OTHERS</td>
<td>51</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>52</td>
</tr>
</tbody>
</table>
List of Figures

1.1 **Categorization of SSD research.** The figure is explained in Section §1.2. The first bar reaches 195 papers. 

2.1 **FEMU architecture.** The left figure shows the location of FEMU in the software stack, where FEMU sits below QEMU NVMe protocol emulation layer and above QEMU block I/O layer. The right figure zooms in the FEMU part in the left and shows how control path (main emulation framework) and data path of each NVMe command diverge and merge during the command life time.

3.1 **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. The storage is memory-backed, thus the IO latency represents the software overhead. The experiments were run on a dual-thread (2x) 24-core machine, hence no CPU contention. For Figure (a), the IDE and NVMe lines representing VSSIM and LightNVM’s QEMU respectively are discussed in §1.2; virtio, dp, and FEMU lines in §3.1. For Figure (b), the “+50µs (Raw)” line is discussed in §4.1; the “+50µs (Adv)” line in “Result 3” part of §4.2.2.

3.2 **QEMU NVMe Architecture.** This figures briefly show QEMU NVMe architecture, including interaction between guest OS NVMe driver and QEMU side NVMe device emulation. The I/O processing logic is marked by 1 to 7, with detailed explanation in §3.2.1.

3.3 **QEMU latency breakdown under NVMe, virtio-blk (virtio) and virtio-dataplane (dp).** This figures show the latencies brought by each parts in I/O life cycle. It helps us analyze the scalability of each part. “Guest OS” accounts for the time since the IO enters guest kernel until driver I/O submission, “KVM” represents the time taken for I/O virtualization handling in the host kernel plus QEMU post processing before entering corresponding device emulation layer, and “QEMU” shows the time of I/O processing in QEMU I/O emulation.

3.4 **QEMU NVMe IOPS w/ FEMU optimizations.** The figure shows the scalability of QEMU’s NVMe implementation under our optimizations. New line represents stock QEMU NVMe, +dbbf represents QEMU NVMe with shadow doorbell buffer support, +poll adds polling based on +dbbf, and finally +heap applies our own heap storage backend on top of all previous optimizations. 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 aggregate IOPS achieved.
4.1 **Delay Emulation: 50μs and 100μs.** Emulating 50μs and 100μs device access latency, FEMU delay emulation doesn’t introduce tail latencies. ........................................... 26

4.2 **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. ....................................................... 30

4.3 **OpenChannel SSD (OC) vs. FEMU.** X: # of channels, Y: # of planes per channel. The figures are described in §4.3.2. The red line represents latency standard errors. For each configuration set (X, Y), we run the same fio workloads on both FEMU and OC and collect latency numbers. This experiment is to show FEMU can accurately emulate the queueing delay when multiple I/Os are sent to one NAND chip and parallelism when multiple I/Os are served evenly by multiple NAND chips. ........................................... 33

4.4 **Filebench on OpenChannel SSD (OC) vs. FEMU.** The figures are described in the “Result 2” segment of §4.3.3. 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). ................................. 34

5.1 **Use examples.** Figure 5.1a is described in the “FTL and GC schemes” segment of Section 5.1. Figure 5.1b is discussed in the “Distributed SSDs” segment of Section 5.1. ................................. 36

6.1 **Latency CDFs from microbenchmarks.** The figures are explained in Section 6.3.1. ..................... 45

6.2 **MITTSSD vs. Hedged.** The figures are explained in Section 6.3.2. ................................. 46

6.3 **Prediction inccuracy.** (As explained in §6.4). ................................. 47
List of Tables

1.1 SSD Research Platforms Comparison. SSD Research Platform Pros & Cons: Simulator v.s. Emulator v.s. Hardware platforms ................................................................. 2

4.1 FEMU emulated OpenChannel-SSD Parameters. We use the above parameters for the emulated SSD. Latency numbers are profiled from a real enterprise OpenChannel-SSD. The latencies are based on average values; actual latencies can vary due to read retry, different voltages, etc. Flash reads/writes must use the plane register. We use 128 GB out of 256 GB physical memory to serve as the emulated SSD backend storage. For the microbenchmark experiments, we change #Channels & #Planes/channel combinations to verify the latency accuracy under different settings. ................................................................. 32
Acknowledgments

The PhD journey is not done yet, leave the acknowledgement and the best for later.
Abstract

Cheap and extensible research platforms are a key ingredient in fostering wide-spread Solid-State Drives (SSDs) research. While SSD simulators are popular, they only support in-SSD research. SSD hardware development platforms are expensive and experience wear-out issues. This leaves software-based flash emulators an excellent alternative choice. 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. To this end, we present FEMU, a QEMU-based flash emulator, with the following four “CASE” benefits.

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).

First, FEMU is cheap ($0) as it is an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences. We hope FEMU will be useful to broader communities and accelerate research in broader areas.

Second, FEMU is (relatively) accurate. For example, FEMU can be used as a drop-in replacement for OpenChannel SSD; thus, future research that extends LightNVM 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 scalable. 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.

Finally, FEMU is extensible. 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.
Chapter 1

Introduction

1.1 SSD Research Platform Introduction

Cheap and extensible research platforms are a key ingredient in fostering wide-spread Solid-State Drives (SSDs) research. Existing SSD research platforms can be categorized into three types: simulator, emulator and hardware based platforms, as shown in table 1.1. Since SSDs’ first debut as server storage more than a decade ago, we have seen numerous SSD architecture designs and FTL algorithms innovations. For example, software defined flash (SDF) which offloads NAND management tasks to the host, FTL algorithm innovations like wear leveling mechanisms for increasing SSD lifetime, better IO scheduling / Garbage Collection (GC) algorithms to improve overall I/O performance, etc. While simulation is a quick and easy way to evaluation new designs, it experiences several drawbacks we discuss below.

SSD simulators such as DiskSim’s SSD model [15], FlashSim [25] and SSDSim [30], despite their popularity, only support internal-SSD research but not kernel-level extensions. That means, users can only run workload traces to verify new designs. Even today, SSD researchers are still depending on IO workload traces which were published a decade ago retrieved from disk-based systems. This imposes an embarrassing situation here as no new workload traces are open sourced and researchers have to rely on those decade-old disk traces. On the other hand, hardware research
<table>
<thead>
<tr>
<th>Platform</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>Cheap; Easy; Time-saving</td>
<td>Trace-driven; Internal research only</td>
</tr>
<tr>
<td>Emulator</td>
<td>Cheap; Full-stack research support</td>
<td>Poor scalability; Poor accuracy</td>
</tr>
<tr>
<td>Hardware</td>
<td>Full-stack research support; Accurate</td>
<td>Expensive; Complex; Wear-out</td>
</tr>
</tbody>
</table>

Table 1.1: SSD Research Platforms Comparison. SSD Research Platform Pros & Cons: Simulator v.s. Emulator v.s. Hardware platforms

platforms such as FPGA boards [48, 54, 70], OpenSSD [13], or OpenChannel SSD [19], support full-stack software/hardware research but their high costs (thousands of dollars per device) impair large-scale SSD research. Also, hardware based platforms are complex to use since the development environment is quite different from application development in the host. It requires very low level knowledge about the System-on-Chip (SoC) to make viable modifications. This would greatly lengthen the project development cycle. Wear-out issues would also jump in when the device is not programmed carefully. As a return, we get the most accurate results since everything is “real”.

This leaves software-based emulator such as QEMU-based VSSIM [68], FlashEm [71], and LightNVM’s QEMU [12], as the cheap alternative platform. Emulators have the potential to achieve benefits of both the hardware platforms in full system stack support and simulators in easiness to use. Essentially, emulators are software, they simulate the hardware logic and expose a “fake” device (emulated) to guest OS, thus enabling research at different levels. This also guarantees that it’s flexible to use as users are free to propose device level innovations and experiment it using application level benchmarks (real workloads).

Unfortunately, the state of existing emulators is bleak; they are either outdated, non-scalable, or not open-sourced. Of the popular SSD Emulators we are aware of, VSSIM design is based on IDE interface, which exposes performance constraints over utilizing high parallelism exposed by today’s SSDs. Its simple whole-GC design where GC will lock down the whole device cannot represent today’s fine-granular GC algorithms where user and GC request can co-exist. FlashEmu is no longer maintained and LightNVM’s QEMU is only designed for OS level FTL development.
It’s not a performance platform and only supports single channel configuration. The poor design of LightNVM’s QEMU plus the virtualization overhead prevent it from being able to emulate hundreds of microsecond level latencies in state-of-the-art NAND Flash.

We argue that it is a critical time for storage research community to have a new software-based emulator (more in §1.2). To this end, we present FEMU, a QEMU-based flash emulator, with the following four “CASE” benefits.

First, FEMU is cheap ($) as it is an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences [26, 66]. We hope FEMU will be useful to broader communities and accelerate research in broader areas.

Second, FEMU is (relatively) accurate. For example, FEMU can be used as a drop-in replacement for OpenChannel SSD; thus, future research that extends LightNVM [19] 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 scalable. 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.

Finally, FEMU is extensible. 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.

In the following sections, we first present an extended motivation (§1.2).
1.2 Extended Motivation

1.2.1 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 [68] or FlashEm [71]); [H]: hardware platforms (FPGA boards, OpenSSD [13], or OpenChannel SSD [12]); or [S]: trace-based simulators (DiskSim+SSD [15] or FlashSim [25] and SSDSim [30]).

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 1.1 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),

---

1. ASPLOS, EuroSys, FAST, MSST, OSDI, SOSP, SYSTOR, TECS, TPDS, TOC, TOS, etc..
and modify the low-level SSD controller logic (L). However, simulators do not support running applications and operating systems.

### 1.2.2 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.) is high, especially in large deployments, required for distributed storage experiments. The top-8 categories in Figure 1.1, a total of 324 papers (83%), target single SSD (I-...) 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 [48] and Tencent [70]). Other hardware platforms such as OpenSSD [13] and OpenChannel SSD [12] also cost thousands of dollars each, impairing multi-node non-simulation research, especially in academia.

### 1.2.3 The Rise of Software-Defined Flash

Today, research on host-managed (aka. “software-defined” or “user-programmable”) flash is growing [42, 48, 54, 55, 64, 70]. The idea is to have, not only the inflexible SSD firmware, but also the software (e.g., the host OS or application) manage the flash devices. 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 [12], managed by Linux-based Light-NVM [19]. The SSD exposes all the internal physical page addresses (channels, chips, blocks, and pages) to the host. 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 as a viable hardware platform for software-defined flash research.

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.

1.2.4 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 [14, 33, 41, 59, 65], wherein some functionalities move up to the OS kernel (K) and some other move down to the SSD firmware (L) [32, 51, 56]. For example, page-level synchronization for preventing read-write data race should move up because testing data-race free firmware is long and expensive\(^2\); garbage-collection (GC) management should move up as it must be tied to user SLAs [37]; and some features such as atomic writes, persistent trim, and deduplication should move down [56]. While these are just a few examples, there is a vast research space to explore. 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 [33] or academic researchers with Linux+OpenSSD [35, 52] or with block-level emulators (e.g., Linux+FlashEm) [49, 71]. OpenSSD with its single-threaded, single-CPU, whole-blocking GC architecture also has many known major limitations [66]. 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 Controller & its firmware) 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.

\(^2\) Per our conversations with SSD engineers
1.2.5 The State of Existing Emulators

We are only aware of three “popular” software-based emulators: FlashEm, LightNVM’s QEMU and VSSIM.

FlashEm [71] 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 [10] 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, due to the virtualization overhead and excessive use of heavy syscalls in virtual I/O path, 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 3.1a). Later, we show that FEMU can be used as a drop-in replacement of OpenChannel SSD with relative accuracy, for prototyping purposes.

VSSIM [68] 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 3.1a 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 583 µs. The root cause is that IDE is not supported with virtualization optimizations.

When we add just a 50 µs delay emulation in VSSIM (i.e., as if a page read takes 50 µs), the resulting average latency multiplies further, up to 5 ms with 16 IO threads (although the IOs are directed to different channels/chips). This is because VSSIM delay emulation is built within the single-threaded IDE entry path. Thus busy loop creates queueing delays. In our setup, VSSIM’s maximum user throughput is only 10 KIOPS and drops to 1 KIOPS with an ongoing GC.

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 [68, 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. When multiple applications run concurrently, VSSIM monitor only reports the overall IO latency but cannot distinguish the timing of each application.

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.
Chapter 2

FEMU Design

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.

For simplicity, FEMU guarantees data correctness by reusing original QEMU I/O path and embeds control path along the data path but as a separate module. This design simplifies future functionality upgrades. I/O data from the guest OS will be served/stored in FEMU backend storage backed by DRAM to avoid extra latency overheads by accessing physical drives (low latency is needed for accurate SSD latency emulation).

FEMU can act as either an OpenChannel/whitebox-SSD (e.g. CNEX Westlake SDK) or a blackbox commercial SSD with a FTL running inside the controller. It consists of the following components:

1. An optimized QEMU I/O stack which enables SSD parallelism and NAND access latency emulation at hundreds of microsecond level.

2. Controller and NAND architecture emulation via a performance model emulating command arbitration logic, detailed NAND characteristics, etc.
Figure 2.1: **FEMU architecture.** The left figure shows the location of FEMU in the software stack, where FEMU sits below QEMU NVMe protocol emulation layer and above QEMU block I/O layer. The right figure zooms in the FEMU part in the left and shows how control path (main emulation framework) and data path of each NVMe command diverge and merge during the command life time.

3. FTL (only in blackbox mode), currently FEMU is powered by a simple page-level mapping based flash translation layer with aggressive GC algorithm working at different granularities (chip/channel/whole-blocking [66])

4. Delay emulation module, which delays I/O completion by certain amount of time before triggering interrupt to guest OS but without affect overall I/O throughput.

Optimized QEMU I/O stack is needed to deliver good performance even under high concurrent I/O workloads from guest OS applications. With this capability, FEMU can support emulating parallel channels and chips without unintended queueing delays caused by virtualization/QEMU overhead. This is the first step towards usable and accurate SSD logic emulation. Detailed optimization techniques will be described in Chapter 3.

SSD controller and NAND architecture emulation represent today’s SSD internal hardware components, which can schedule and execute I/O commands. By profiling our CNEX OpenChannel-SSD, we gained detailed knowledge about NAND MLC page latency variance, advanced NAND
commands such as multiplane command and caching commands which are used to exploit internal parallelism. Detailed performance model will be discussed in Chapter 4.

FTL is the software running inside SSD controller and is responsible for various tasks to manage NAND Flash, such as mapping, caching, I/O scheduling, GC, and wear leveling, etc. When FEMU is used as a blackbox SSD, it runs a basic page FTL inside for I/O management. We extended VSSIM’s FTLs with several new features, such as multi-drive support, fine granular GC algorithms, etc. We put detailed FTL designs in FEMU release document [1].

When I/Os goes through emulated channels and chips, they will be scheduled/queued according to the idle status of those parallel units. Then, each I/O will carry a future timestamp which represent the time when I/O will be completed by the NAND, which also essentially represent the time when I/Os should be returned to guest OS.

Last but not least, delay emulation is an important part to allow applications to perceive the parallelism exposed by FEMU. The goal of delay emulation module is twofold: (1) delay an I/O operation accurately (2) delay multiples I/O without negatively affecting each other (not blocking the processing of other I/Os). The techniques to tackle these challenges will be described in Chapter 4 in details.

In the rest of the article, we focus on the main challenges of designing FEMU: achieving scalability (§3.1) and accuracy (§4) and increasing usability and extensibility (§5.1).

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.
Chapter 3

FEMU Scalability

By scalability, we mean the ability of the target system to handle concurrent I/Os within a certain latency threshold without causing unintended queueing delays. Scalability is an important property of a flash emulator, especially with high internal parallelism of modern SSDs.

Modern SSDs are equipped with hundreds of independent NAND flash chips across dozens of channels. Each NAND flash chip is an independent unit which can execute a NAND command (e.g. Page Read, Page Program or Block Erase, etc.) at a time. This implies that modern SSDs can process hundreds of inflight I/Os simultaneously without causing queueing delays. Exploiting this massive amount of hardware level parallelism has been a key research area for storage researchers. However, as we will show later, stock QEMU/KVM is far from being able to provide similar parallelisms with concurrent I/Os even under the ideal case where there is no delay emulation at all. Thus, to architect an accurate emulator platform which can deliver similar parallelisms and latency timings as real hardware platforms do, we need to solve the scalability bottleneck of current QEMU/KVM implementation. In this section, we first show experimental results demonstrating QEMU/KVM is not scalable to deliver stable performance for low latency and parallelism emulation, then we focus on explaining where such scalability bottleneck comes from and how FEMU tackles this challenge.
3.1 QEMU I/O Scalability

QEMU supports various types of storage interface emulations, which can be categorized into three types. The first one (type I) is fully virtualized interfaces, such as IDE/SATA and NVMe, where QEMU emulates the device IO according to corresponding device specifications. In this mode, guest OS can run out-of-box with vanilla device driver without modification, thus providing flexible support to different operating systems. However, this flexibility comes at cost of low performance as those interfaces are not optimized for virtualization. Existing popular SSD emulators, such as VSSIM and LightNVM’s QEMU, are based on this type of I/O virtualization, thus they cannot provide scalability we need to emulate high scalability in today’s SSDs. The second type (type II) is para-virtualized storage interfaces, e.g. virtio. This is a special type of storage interface designed to work with virtual machines only, which brings better performance than full virtualization. However, even with virtio, we will show later that it’s still not scalable enough to support our scalability needs. The third type is hardware assisted virtual I/Os, such as VT-d and SR-IOV. They depend on hardware support to assign (partial) device to the VM and achieve close to bare metal performance. Even so, due to QEMU software overhead in interrupt handling, it cannot deliver the scalability we need neither.

This leaves us to use ramdisk (e.g. tmpfs or block ram device) as the backend storage device and stack an emulated storage interface (type I or II) on top. Ramdisks are backed by DRAM, whose access latency is at 100ns level and thus is negligible compared to the NAND latency we are trying to emulate. In this case, we measure I/O latencies under different number of I/O threads to see how scalable QEMU/KVM is. The results are shown in Figure 3.1a. Since the backend is DRAM, the perceived latencies represent capability of QEMU/KVM in handling concurrent I/Os.

Unfortunately, stock QEMU exhibits a scalability limitation. For example, as shown in Figure 3.1a, the red line shows QEMU IDE cannot scale with number of increasing I/O threads at all since it’s a two decade old interface designed to process one I/O at a time without parallelism.
When number of I/O threads doubles, we can directly observe that I/O latency doubles. Thus, it’s not a good fit. 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 $106\mu$s). This is highly undesirable because typical read latency of modern SSDs can be below $100\mu$s, let alone we need to emulated tens to hundreds of parallel I/Os.

More scalable alternatives to NVMe are para-virtual interfaces, as shown by the virtio and virtio-blk dataplane (dp) lines [7, 50]. (virtio/dp vs. NVMe lines in Figure 3.1a). virtio-blk dataplane (dp) extends the basic virtio-blk interface with a dedicated thread working in polling mode, thus it can achieve better scalability compared to virtio. However, these interfaces are not as extensible as NVMe since they only support simple read, write and flush I/O commands. NVMe, as a new standard storage interface designed for today’s fast NVM devices, is lightweight and more extensive, thus more popular now. 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 $185\mu$s and
126µs, respectively.

## 3.2 QEMU I/O Scalability Problem Root Causes

![QEMU NVMe Architecture Diagram](image)

Figure 3.2: **QEMU NVMe Architecture.** This figures briefly show QEMU NVMe architecture, including interaction between guest OS NVMe driver and QEMU side NVMe device emulation. The I/O processing logic is marked by 1 to 7, with detailed explanation in §3.2.1.

As shown in §3.1, QEMU NVMe can only scale to 4 I/O threads, imposing severe scalability challenge in achieving high parallelism emulation. In this section, we first go through how NVMe protocol works and then introduce QEMU NVMe emulation. Finally, we identify the root causes of QEMU NVMe scalability bottleneck.
3.2.1 NVMe Primer

NVMe Express (NVMe) is a new storage protocol specifically designed for today’s fast storage devices, such as NAND Flash SSDs, 3D Xpoint SSDs, etc. It utilizes queue pair based mechanism to facilitate IO processing while exposing minimal software overhead. NVMe defines a set of registers (usually mapped to host memory space) that can be used for communication between OS and the device. Similar to what’s shown in Figure 3.2, the queue pairs (Submission Queue (SQ) and Completion Queue (CQ)) are shared memory buffers between host and the device thus accessible from both sides with no overhead (Device side can access the memory area through DMA). The host OS creates I/O queues during driver initialization phase. According to number of CPU cores in the host, usually equal number of queue pairs are created to achieve best performance in current blk-mq stack. This enables lockless access to the queue, better cache locality and thus better I/O scalability (This part illustrates that current software I/O stack in the OS has been well optimized to adapt with fast storage device without triggering “too much” software overhead).

With NVMe, the I/O processing logic is quite simple. Although Figure 3.2 shows the case for emulated NVMe in QEMU, we can use it for an illustration of I/O handling since QEMU NVMe implementation follows the standard NVMe protocol. The I/Os are handled in following steps:  

1. I/O submission to SQ tail. I/Os are encapsulated into NVMe commands by the driver. Each NVMe command takes up 64bytes and stores all the information which represents the I/O, including SLBA (starting logical block address), NLB (number of logical blocks, i.e., I/O length), PRPs (Physical Region Pages, representing corresponding I/O buffers), and other meta- and management bits. The NVMe command entry is put into the tail of the submission queue, indicating a new I/O for the NVMe device to process.

2. I/O submission notification. This step is for the host to notify the NVMe device about new I/O arrival. This is done by writing the new tail position of SQ to the corresponding doorbell register.

3. NVMe device fetching NVMe command from submission queue head position. In this step, NVMe commands in the queue will be fetched from head and

---

1. Refers to the machine which is talking to the NVMe device, it can be a physical machine or a VM.
Figure 3.3: QEMU latency breakdown under NVMe, virtio-blk (virtio) and virtio-dataplane (dp). This figures show the latencies brought by each parts in I/O life cycle. It helps us analyze the scalability of each part. “Guest OS” accounts for the time since the IO enters guest kernel until driver I/O submission, “KVM” represents the time taken for I/O virtualization handling in the host kernel plus QEMU post processing before entering corresponding device emulation layer, and “QEMU” shows the time of I/O processing in QEMU I/O emulation.

then for 4 I/O enters the NVMe controller, where it will be serviced, either by the chaching layer or sent down to backend storage media (e.g. NAND or PCM). 5) Adding I/O completion entry to completion queue. After I/O is done, the NVMe controller will compose a NVMe completion entry and put it to the tail of completion queue. Then, the device will trigger an interrupt to the host, indicating I/O completions to the host for completion path processing. 6) Host side I/O completion handling. When corresponding interrupt handler is scheduled, the host will fetch an entry from CQ head and return I/O status back to user applications. Lastly, 7) CQ head position updates. Similar to 2, this step is for host to communicate the new head position of CQ to the device.

3.2.2 QEMU NVMe Virtualization & Scalability Root Causes

QEMU/KVM is a full system emulator/hypervisor, which can emulate various types of hardware. It supports running guest OS on top at near bare-metal performance for CPU/Memory intensive
workloads with the help of hardware assisted virtualization techniques (e.g. Intel VT and EPT). However, I/O virtualization suffers significant overhead due to the necessity to trap and emulate I/Os in the QEMU process, instead of running it inside the guest OS (a different context).

To the Operating System, an I/O device can be abstracted as a set of registers either mapped to OS memory space (i.e., MMIO\textsuperscript{2}) or can be accessed via PIO (Port I/O) operations. Accesses to these registers are sensitive operations and needs to be babysitted by QEMU using a trap-and-emulate method. Put it simple, QEMU monitors access to these areas and takes control of the VM execution, in the meanwhile, vCPU is stalled until QEMU signals completion of above memory access. Those register accesses are triggered during I/O submissions/completions and will frequently interrupt VM execution by jumping back and forth. Once QEMU is done with emulating I/O by serving it from corresponding backend image file in the host, it will pause the VM execution again to inject an interrupt to the guest OS (simulating the way how real interrupt works).

QEMU’s NVMe implementation uses traditional trap-and-emulation method to emulate I/Os, thus the performance suffers due to frequent VM-exits. Each trap will cause an expensive VM-exit to be executed, which usually takes several microseconds to save VM context and restore host context. Under concurrent I/O workloads VM-exit delay will be queued up for inflight I/Os thus cascadingly affect overall I/O latency.

Moreover, as shown in Figure 3.3, each guest I/O needs go through guest kernel stack, KVM processing (host kernel) and QEMU (host user space). For QEMU/KVM, guest OS codes runs in dedicated vCPU threads, which are introspected by KVM module, QEMU I/O emulation mainly runs in an event loop thread. These are three major parts where I/O latency comes from. Compared to para-virtualized interfaces (\texttt{virtio} and \texttt{dp}), QEMU NVMe shows a significant overhead in “KVM” and “QEMU” due to its full-virtualization nature. For example, under 16 I/O threads, the time taken in “KVM+QEMU” for \texttt{NVMe} is 150\,µs while it’s only 70\,µs and 45\,µs for \texttt{virtio} and \texttt{dp} respectively. This shows QEMU NVMe virtualization is the main reason of its scalability.

\footnote{\textsuperscript{2} Memory Mapped I/O, which enables device register accesses via volatile memory reads and writes}
bottleneck. Detailed root causes is below.

Root Causes: Collectively, all of the scalability bottlenecks above are due to two reasons: (1) **Software overheads caused by frequent VM-exits.** For each NVMe I/O, the Guest OS’ NVMe driver first “rings the doorbell [11]” to the device (QEMU in our case) that some IOs are in the device queue. This “doorbell” write is an MMIO operation that will cause an expensive VM-exit (“world switch” [60]) from the Guest OS to QEMU. Such VM-exits have been well-known as I/O virtualization performance bottlenecks for years. Worse, a similar doorbell operation must also be done upon IO completion to update completion queue head position, this doubles the number of VM-exits for each I/Os (corresponding to step 2 and 7 in Figure 3.2). Under concurrent I/Os, the VM will be frequently interrupted so as to leaving limited vCPU resource for inflight I/O handling. This is one reason why QEMU NVMe scale to even 4 I/O threads in Figure 3.2. (2) **QEMU AIO processing overheads.** 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. According to our evaluations, QEMU AIO may take more than 20µs to finish while accessing data directly from the ramdisk backed image file only takes less than 1µs. Although QEMU uses a thread pool to distribute I/Os evenly, they don’t help here since the overhead comes from QEMU’s single threaded block I/O layers where I/O submissions are sequentialized. To be specific, upon receiving MMIO signal for I/O arrival notification, QEMU needs to go through NVMe device emulation layer, block driver layer, image format driver and raw device driver before I/Os can be put into QEMU’s global AIO queue. This is already a long I/O path. Further, worker threads need to process these AIOs by submitting them to the host OS by traversing the whole host I/O stack. While QEMU/KVM depends on these different layers for features implemented in QEMU block layer, such as I/O throttling, VM migration, etc. These features are not needed in our case.
3.3 **Scalability Solutions**

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

### 3.3.1 Polling based QEMU NVMe Design

In order to overcome the excessive VM-exit overhead, we transform QEMU from an interrupt-driven (trap) 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 and get better performance 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.

In details, our polling design is enabled by the Shadow Doorbell Buffer Support proposed in NVMe Specification version 1.3\[^{11}\], §7.10. It introduces a set of shadow doorbell buffers which are shared memory buffers between guest OS and QEMU. Upon I/O submission and completion, corresponding shadow doorbell buffer will be updated for SQ/CQ tail/head updates and only when necessary will the doorbell register be written. Essentially shadow doorbell buffer adds para-virtualization capability to virtual NVMe controllers, like QEMU NVMe. Thus, it can used to enhance QEMU NVMe I/O performance. Potentially it will reduce the number of VM-exits, thus provide better scalability. Although Shadow Doorbell Buffer Support has been implemented in Linux kernel, QEMU NVMe lacks support to this feature.

To reap the benefits brought by this feature, we first enhance QEMU with shadow doorbell buffer capability. With this, we can observe obvious performance boost (Figure 3.4, +dbbuf line). However, the performance is still not scalable enough to support 32 parallel I/Os (not shown).
The reason is that there still exits VM-exits caused by doorbell writes. While shadow doorbell buffer mechanism greatly reduce number of doorbell writes needed, they are not eliminated. Thus, I/O performance is affected under intensive workloads and show worse tail latencies. Moreover, QEMU still depends on doorbell writes for new I/O arrivals. As guest OS will send multiple I/Os before it rings the doorbell (thanks to shadow doorbell buffer support), QEMU is passively passed the control at a longer intervall, which may hurt overall I/O latencies.

Thus, to make QEMU NVMe more scalable, we use polling techniques. Polling works by proactively querying new I/O arrivals instead of passively waiting for control transfer. This way, polling can handle I/Os in a more timely manner. Our polling design leverage the fact that SQ/CQ tail/head updates are updated to shadow doorbell buffer as well as doorbell registers, we can check the shadow doorbell buffer periodically for latest I/O submissions or completions. If no updates are made to the shadow buffers (i.e., no new I/O submission or completion), we can simply skip QEMU I/O emulation logic and poll the status change again next time. In our current polling design, we utilize QEMU’s timer APIs to setup a periodic event for this purpose. We will discuss the limitation of this method in §8.1 part. With polling, we can see from Figure 3.4, +poll that can sustain 400K IOPS under 64 I/O threads. However, from 32 to 64 threads, the aggregate IOPS only increase 19%, which implies we are hitting another bottleneck, which we solve in next section.

3.3.2 CUSTOMIZED QEMU AIO PATH

As pointed out earlier, the long I/O path (many layers) in QEMU’s AIO module bring much overhead and straggle I/O performance. An initial thought is to figure out a way to shorten I/O path without hurting correctness. Considering FEMU purpose is only for research prototyping and doesn’t care about rich features provided by QEMU, we compose our own memory backend and skip QEMU AIO component completely.

To be specific, 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
Figure 3.4: **QEMU NVMe IOPS w/ FEMU optimizations.** The figure shows the scalability of QEMU’s NVMe implementation under our optimizations. `nvme` line represents stock QEMU NVMe, `+dbbuf` represents QEMU NVMe with shadow doorbell buffer support, `+poll` adds polling based on `+dbbuf`, and finally `+heap` applies our own heap storage backend on top of all previous optimizations. 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 aggregate IOPS achieved.

This way, we totally skip the QEMU block I/O layer and I/O path on the host stack, thus I/Os can be handled immediately after it enter the NVMe device emulation layer. Another problem arises when doing this is DMA correctness. Traditionally, QEMU emulate DMA operations are tightly coupled with the block I/O layer to transfer data between guest OS and backend image file on the host. With our new memory backend, QEMU’s DMA engine doesn’t work. To tackle this problem, 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). With these changes, we achieve ultimate performance approaching 1 million IOPS as shown in Figure 3.4 +heap line.
3.3.3 **FEMU Optimization Results**

**Latency:** The bold **FEMU** blue line in Figure 3.1a 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.

**IOPS:** Figure 3.4 shows the IOPS we can achieve after applying shadow doorbell buffer support, polling design and our customized heap backend storage. It clearly demonstrates that our optimizations can greatly improve overall IOPS by $10 \times$ under 64 I/O threads. This enables FEMU to emulate 32 parallel channels/chips.

The remaining scalability bottleneck now only comes from QEMU’s *single-thread* “event loop” [8, 28], which performs the main IO routine such as dequeueing the device queue, triggering DMA emulations, and sending end-IO completions to the Guest OS. Worse, this thread must synchronize with all the running vCPU threads, incurring additional performance loss. Recent works addressed these limitations (with major changes) [18, 39], but have not been streamlined into QEMU’s main distribution. We are exploring the possibility of integrating other solutions in future development of FEMU. Ongoing efforts will be described in §8.1.
Chapter 4

FEMU Accuracy

We now discuss the accuracy challenges. To accurately emulate an SSD in QEMU, two problems need to be solved: (1) NAND Flash Access Latency Emulation: since NAND reads are at 100µs level, FEMU needs to be able to accurately delay an I/O for certain amount of time to emulate NAND access and inter-firmware I/O queueing delays, without affecting other inflight I/Os coming from guest OS. (2) SSD Performance Model: we need detailed knowledge about the controller architecture and firmware logic to emulate I/O processings inside the SSD. In this section, we first describe our delay mechanism (§4.1), and then dive into our basic and advanced performance models (§4.2). Finally, we present FEMU accuracy results towards emulating an enterprise level OpenChannel-SSD.

4.1 Delay Emulation

NAND Flash supports page read, page program and block erase operations, at the latency of 100µs, 1.5ms, and 6ms respectively\(^1\). That said, when a read is sent to the SSD, user will get the data after around 100µs. Traditional SSD simulators don’t simulate such delays using wall clock time. Instead, they usually maintain an internal state machine and advance it accordingly when handling

---

\(^1\) Latency numbers profiled from CNEX 2TB SSD using Micron L95B eMLC NAND
an I/O event. For example, with an incoming read, it would simply do +100µs to its state structure and return the I/O immediately. Later, it reports the I/O takes (100+Δ)µs, where Δ is the extra latency due to queueing, GC, etc. As an emulator, FEMU needs to delay I/O completion using wall clock time so as to emulate hardware latency behavior, giving guest OS/application an illusion that I/O processing in the emulated device takes that amount of time to finish, just as hardware SSD platforms do.

4.1.1 “UNSUCCESSFUL” DELAY EMULATION EFFORTS

An intuitive way to do delay emulation is to use busy-loops or sleep(). sleep() would stall the thread thus it’s not a good choice, especially under the case that current simulators and QEMU use single thread for I/O handling. Busy-loops can provide most precise delay emulation at the cost of CPU resource by constantly checking if end timestamp has expired. Doing delay emulation inside QEMU is tricky as it needs to be well incorporated with existing QEMU I/O processing framework, without affecting overall I/O performance. Below, we first describe our prior unsuccessful delay emulation efforts and then introduce our current endio-based delay emulation mechanism.

- **“putback” method**: This method utilizes the parallelism exposed by QEMU thread pool. We let each QEMU request carry an ending timestamp, and when worker threads fetch an I/O, they will check if request timestamp has expired, if not, they will put it back to the AIO queue and wait for next time processing.

- **“busyloop” method**: While busyloop in QEMU’s single threaded submission path is not feasible, we perform busy-loop in the worker thread (we can busy-loop 64 inflight I/O at the same time which is limited by thread pool size of 64).

- **“timer” method**: For each I/O, we setup a timer event to delay it for certain amount of time before submitting it into QEMU AIO queue. Timer event works in a similar manner to sleep(), but it won’t block other I/Os (asynchronous manner).
Figure 4.1: **Delay Emulation: 50µs and 100µs.** *Emulating 50µs and 100µs device access latency, FEMU delay emulation doesn’t introduce tail latencies.*

Figure 4.1a shows the results achieved by using different delay emulation methods. For a guest application I/O, the guest kernel stack overhead is 10-20µs in our platform. When we try to emulate a device latency of 50µs for each I/O, the expected I/O latency perceived by the guest application should be in the range of 60-70µs, as shown in the gray area. The `putback`, `busyloop` and `timer` lines correspond to the above mentioned delay emulation methods. However, all of them show a 20-30µs offset in the emulated latencies. The reasons are (1). These methods run deep in QEMU’s block I/O layer and don’t consider QEMU overhead during I/O emulation. (2). These methods lower QEMU AIO processing efficiency by hogging more CPU resources, thus preventing stable I/O latencies. With these experiences, we design `endio` based delay emulation technique to overcome drawbacks in previous designs.

### 4.1.2 Endio 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 \( T_{endio} > T_{now} \).

4.1.3 Endio Delay Emulation Results

The “+50us (Raw)” line in Figure 3.1b shows a simple (and stable) result where we add a delay of 50µs to every IO \( T_{endio} = T_{entry} + 50\mu s \). 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\textsuperscript{th} percentile) of the IOs exhibit latency tail of more than 105µs, which already exists in stock QEMU. For example, in VSSIM, the 99\textsuperscript{th}-percentile latency is already over 150µs. This shows that endio method is salable and efficient in emulate device latencies.

Let’s take a closer look at the I/O latency distribution of this endio method. As shown in Figure 4.1a, endio line sits entirely in the expected gray area, proving its advantage over other methods (to the right). Further, in Figure 4.1b, we also get similar results when trying to emulate 100µs latency for each I/O.

4.2 SSD Performance Models

FEMU scalability and endio delay emulation have paved the way for FEMU to be able to emulate an SSD. What’s missing here is the performance model of the SSD, which defines the internal controller architecture and firmware (FTL) used to manage I/Os working around NAND limitations and guaranteeing high performance.

**Overview of an SSD controller architecture:** Due to NAND Flash material level limitations, such as asymmetric access granularity for read/program and erase, non-in-place updates, limited P/E (program/erase) cycles and data retention time, today’s SSDs are usually made as a SoC (System on Chip) with its own processor, SRAM/DRAM, and firmware/FTL (Flash Transla-
FTL is the soul of an SSD and it’s responsible for mapping table management, caching, I/O scheduling, GC (Garbage Collection), background scrubbing, etc. NAND Flash chips are organized into channels (e.g. 16), with several (e.g. 8) chips mounted on one channel. Controller communicates with NAND Flash chips by sending/receiving information through the channel, which includes NAND commands, address and data transfer.

**NAND Operations:** Take NAND read process as an example. The controller first sends NAND read command to the command register, put page address to the addr register, and then NAND chip will go busy reading data from NAND cell array into its internal page buffer. At this time, the controller is free to do anything else as NAND chip is an independent operation unit. Controller can query the NAND read progress by sending read status command, and this is usually offloaded to a specific hardware module by proactive status polling without blocking the controller. When NAND read operation is done (data in page buffer), the controller will send read data to controller command, and transfer data to controller DRAM through the channel. The data is first decoded by ECC engine to check if any error happens before returning it back to user. Otherwise, read retry will kick in to try hard to read the data out without errors. NAND program operation is similar, but needs to send command to transfer data to NAND page buffer first and then issue NAND program command which will write store data into NAND cell arrays. Erase is simple, the controller sends block erase and the NAND will become busy with erasing data in corresponding block. Once operation is finished, NAND Flash chip will be in ready status again, meaning it can accept new commands for processing.

**Channel:** All the NAND Flash chips on the same channel share one bus, which is multiplexed for command, address and data transfer. When channel is busy with data transfer from/to a NAND chip, it’s in busy status, and at this time, other pending operations which need to use channel must wait for prior operation completion. Thus, the channel may be contentious and limit overall parallelism among NAND chips mounted on the same channel.

Since command and address transfer only take several nanoseconds, In our performance mod-
els, FEMU only emulates the data transfer latency. Below we present two delay model aiming for accurate emulation toward a commercial OpenChannel-SSD. With these delay models, FEMU is also able to run a FTL inside QEMU (as prior project VSSIM does) taking I/O latencies from various sources into account.

### 4.2.1 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 ($T_{freeOfChannel1}=T_{now}+T_{transfer}$, 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 $T_{free}$ accordingly. Similarly, queueing delay can also be well emulated. If an incoming I/O finds target channel/chip with a free timestamp in the future, then it means corresponding channel/chip is busy and we will mark the I/O latency as waiting time plus the time needed to serve this I/O when it’s scheduled for execution.
4.2.2 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). OC is a new kind of SSDs which expose its internal architecture to the host, thus allowing moving FTL management up to the OS. Our OC is a CNEX Westlake PCIe SSD with 16 channels and in total 128 NAND chips. Nowadays, Linux supports running OC with a kernel-level FTL named LightNVM. LightNVM implements basic sector-based mapping table, data placement and GC functions.

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 4.2 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.

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 48/64\(\mu\)s for lower/upper-page read and 900/2400\(\mu\)s 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, “LLLLLuLLuLLuu…”, 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”.

In addition, we built an efficient OC-extension to FEMU NVMe (based on LightNVM’s QEMU base OC extension structures). With these, we are able to run LightNVM on top of FEMU.

4.3 FEMU ACCURACY RESULTS

4.3.1 WORKLOADS & EXPERIMENT SETUPS

We use fio to stress test FEMU and OC under different configurations with direct=1 on raw devices. We use filebench to compare FEMU and OC under real world application workloads, including File Server, Network FS, OLTP, Varmail, Video Server and Web Proxy.

We assign our OC to a VM and access it from inside the guest OS for fair comparison with FEMU emulated OC. Need to mention that assigned OC in VM doesn’t suffer performance drop compared to OC running on physical machines. Our host machine consists of 2x Intel(R) Xeon(R) CPU E5-2670 v3 running at a base frequency of 2.30GHz with 256GB DRAM. FEMU emulated SSD takes up 128GB memory. HyperThreading, C/P-states and Intel Turbo Boost are turned off for consistent and max performance. We also pin FEMU threads to physical cores to avoid latency jitters caused by process migrations. Hugepage is used for minimum GPA-HVA (guest physical addr to host virtual addr) address translations.

By incorporating this detailed model, FEMU can act as an accurate drop-in replacement of OC, which we demonstrate with the following results.
Table 4.1: **FEMU emulated OpenChannel-SSD Parameters.** We use the above parameters for the emulated SSD. Latency numbers are profiled from a real enterprise OpenChannel-SSD. The latencies are based on average values; actual latencies can vary due to read retry, different voltages, etc. Flash reads/writes must use the plane register. We use 128 GB out of 256 GB physical memory to serve as the emulated SSD backend storage. For the microbenchmark experiments, we change #Channels & #Planes/channel combinations to verify the latency accuracy under different settings.

<table>
<thead>
<tr>
<th>Sizes</th>
<th>Latencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD Capacity</td>
<td>Page Read 48µs; 64µs (MLC)</td>
</tr>
<tr>
<td>#Channels</td>
<td>(flash-to-register)</td>
</tr>
<tr>
<td>#Planes/channel</td>
<td>Page Write 900µs; 2400µs (MLC)</td>
</tr>
<tr>
<td>Plane size</td>
<td>(register-to-flash)</td>
</tr>
<tr>
<td>#Planes/chip</td>
<td>Page data transfer 60µs</td>
</tr>
<tr>
<td>#Blocks/plane</td>
<td>(via channel)</td>
</tr>
<tr>
<td>#Pages/block</td>
<td>Block erase 6ms</td>
</tr>
<tr>
<td>Page size</td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 **MicroBench Results**

*Result 1:* Figure 4.3 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 (\(z>1550\mu s\)) 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 (\(z<130\mu s\)). Overall, Figures 4.3a and 4.3b exhibit a highly similar pattern, showing the success of our queuing delay emulation. The latency difference (error) is only between 0.8-11.6%; \(Error = (Lat_{femu} - Lat_{oc})/Lat_{oc}\).

4.3.3 **MacroBench Results**

*Result 2:* Figure 4.4a 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 §1.2, 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 4.4b. 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 4.4b, which is the same configuration used in experiments in Figure 4.4a), 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
Figure 4.4: **Filebench on OpenChannel SSD (OC) vs. FEMU.** The figures are described in the “Result 2” segment of §4.3.3. The y-axis shows the latency difference (error) of the benchmark results on OC vs. FEMU (Error=(Lat_{emu}−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).

Compute overhead will backlog with higher thread count. To show this, Figure 3.1b compares the simple +50µs delay emulation in our raw implementation (§4.1) 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.
Chapter 5

FEMU Use Case

5.1 Usability

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 [26, 66] as well as possible future work that FEMU features enable.

- **FTL and GC schemes:** In default mode, our FTL employs a dynamic mapping and a channel-blocking GC as used in other simulators [15, 30]. One of our projects uses FEMU to compare different GC schemes: controller, channel, and plane blocking [66]. In controller-blocking GC, a GC operation “locks down” the controller, preventing any foreground IOs to be served (as in OpenSSD [13]). In channel-blocking GC, only channels involved in GC page movement are blocked (as in SSDSim [30]). In plane-blocking GC, the most efficient one, page movement only flows within a plane without using any channel (i.e., “copyback” [5]). Sample results are shown in Figure 5.1a. Beyond our work, recent works also show the benefits of SSD partitioning for performance isolation [19, 31, 37, 47, 58], which are done on either a simulator or a hardware platform. More partitioning schemes can also be explored with FEMU.

  *Possible future research:* Decades of FTL/GC research mainly uses simulators, but future
Figure 5.1: Use examples. Figure 5.1a is described in the “FTL and GC schemes” segment of Section 5.1. Figure 5.1b is discussed in the “Distributed SSDs” segment of Section 5.1.

FTL/GC research can also be done with FEMU.

- **White-box vs. black-box mode:** FEMU can be used as (1) a white-box device such as Open-Channel 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. We are not aware of any software-based emulator that can emulate a flash array. To setup an emulated SSD array, one must assemble network block device connections to multiple machines running QEMU. A nested virtualization is another alternative, but nevertheless, these methods add a significant overhead to an already stringent latency requirement (§3.1).

- **Disaggregated Flash:** Disaggregation makes centralized resource management easier and helps improve resource utilization. Historically, iSCSI has been the dominating networking storage protocol for remote block storage access. To keep up with fast speeds of today’s SSDs, a new
protocol named NVMe over Fabrics (NVMe-oF) has been standar dized to support NVMe Flash Disaggregation. To overcome the software overhead brought by context swithces, interrupts due to operating systems management overhead, user space based I/O framework, such as SPDK is a popular choice in conjunction with NVMe-oF. As a first step, we have identified FEMU’s full support to run SPDK and FEMU emulated NVMe device supports NVMe-oF. Since NVMe-oF requires RDMA to work, we use a software-based solution (SoftROCE) without using an expensive RDMA capable NIC. In our setup with two VMs, we can successfully access the NVMe SSD exposed by the other VM through NVMe-oF.

5.2 FEMU Extensibility

- Extensible OS-SSD NVMe commands: (1) As FEMU supports NVMe, new OS-to-SSD commands can be added (e.g., for host-aware SSD management or split-level architecture [51]). 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 [21, 34, 49, 63, 67] and our NVMe-powered FEMU can be extended to support more commands such as transactions, deduplication, and multi-stream.

(2) As mentioned earlier, split-level architecture is trending; our NVMe-powered FEMU can be extended to support other commands such as transactions, deduplication, and multi-stream [21, 34, 38, 46, 49, 63, 67]. (3) Techniques that manage GCs across an array of SSDs, many were only done with simulators [40], can also be supported by FEMU multi-device and NVMe supports.

Successful project: Combining FEMU’s NVMe and multi-device supports, we have built an optimized Linux Software RAID-5 on an array of transparent SSDs. Specifically, if an SSD cannot
serve a read request because of a conflicting GC, the transparent SSD will return an EBUSY error to the OS, which then will trigger our RAID-5 to reconstruct the non-available data from other SSDs. Moreover, if our RAID layer receives multiple EBUSYs from more than one SSD (within a read stripe), our RAID layer will resubmit the read with a new preemption flag turned on (via NVMe), which will slightly postpone the GC.

- **Page-level latency variability:** As discussed before (§4), 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 wordcount workload on a cluster of machines running FEMU, but with different GC schemes as shown in Figure 5.1b. 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 across the three GC mechanisms. We hope FEMU can spur more work that modifies the SSD layer to speed up distributed computing frameworks (e.g., distributed graph processing frameworks).

  **Possible future research:** Many out-of-core distributed graph processing frameworks were recently proposed [43, 72]. This type of research modifies the frameworks to work well on disks/SSDs. On the contrary, there is little work that modifies the SSD layer to speed up the frameworks (§1.2). We hope FEMU can spur more work in this important area.

- **Page-level fault injection:** Beyond performance-related research, flash reliability research [45, 53] can leverage FEMU as well (e.g., by injecting page-level corruptions and faults and observing how the high-level software stack reacts).

- **Interface:** FEMU supports all modern interfaces, NVMe and virtio. The GuestOS will read/write
through either of the interface. Inside FEMU, we have all the basic SSD internal functionalities such as the FTL module, GC and wear-leveling algorithms, delay emulation, performance monitor.
Chapter 6

Other Work

6.1 MITT (MILLISECOND TAIL TOLERANCE) INTRODUCTION

Low and stable latency is a critical key to the success of many services, but variable load and resource sharing common in cloud environments induces resource contention that in turn produces “the tail latency problem.” Early efforts to cut latency tails focused on coarse-grained jobs (tens to hundreds of seconds) [24], where there is sufficient time to wait, observe, and launch extra speculative tasks if necessary. Such a “wait-then-speculate” method has proven to be highly effective; many variants of the technique have been proposed and put into widespread use [17, 61, 69]. More challenging are applications that generate large numbers of small requests, each expected to finish in milliseconds. For these, techniques that “wait-then-speculate” are ineffective, as the time to detect a problem is comparable to the delay caused by it.

One approach to this challenging problem is cloning, where every request is cloned to multiple replicas and the first to respond is used [17]; this proactive speculation however doubles the IO intensity. To reduce extra load, applications can delay the duplicate request and cancel the clone when a response is received (a “tied requests”) [23]; to achieve this, IO queueing and revocation management must be built in the application layer [20]. A more conservative alternative is “hedged requests” [23], where a duplicate request is sent after the first request is outstanding for more than,
for example, the 95th-percentile expected latency; but the slow requests (5%) must wait before being retried. Finally, “snitching” [2] – the application monitoring request latency and picking the fastest replica – can be employed; however, such techniques are ineffective if noise is bursty.

All of the techniques discussed above attempt to minimize tail in the absence of information about underlying resource busyness. While the OS layer may have such information, it is hidden and unexposed. A prime example is the `read()` interface that returns either success or error. However, when resources are busy (disk contention from other tenants, device garbage collection, etc.), a `read()` can be stalled inside the OS for some time. Currently, the OS does not have a direct way to indicate that a request may take a long time, nor is there a way for applications to indicate they would like “to know the OS is busy.”

To solve this problem, we advocate a new philosophy: the OS should be aware of application SLOs and quickly reject IOs with unmet SLOs (due to resource busyness). The OS arguably knows “everything” about its resources, including which resources suffer from contention. If the OS can quickly inform the application about a long service latency, applications can better manage impacts on tail latencies. If advantageous, they can choose not to wait, for example performing an instant failover to another replica or taking other corrective actions.

To this end, we introduce MITTSSD, an OS on top of OpenChannel-SSD storage stack that employs a fast rejecting SLO-aware interface to support millisecond tail tolerance. We materialize this concept within the storage software stack, primarily because storage devices are a major resource of contention [20, 27, 37, 57, 62]. In a nutshell, MITTSSD provides an SLO-aware read interface, “`read(..., slo)`,” such that applications can attach SLOs to their IO operations (e.g., “`read()` should not take more than 20ms”). If the SLO cannot be satisfied (e.g., long disk queue), MITTOS immediately rejects the IOs and returns `EBUSY` (i.e., no wait), hence allowing the application to quickly failover (retry) to another node.

The biggest challenge in supporting a fast rejecting interface is the development of latency prediction used to determine whether the IO request should be accepted or rejected (returning
Such prediction requires understanding the nature of contention and queueing discipline of the underlying resource (e.g., SSD channels/chips, FIFO vs. priority). Furthermore, latency prediction must be fast; the computing effort to produce good predictions should be negligible to maintain high request rates. Finally, prediction must be accurate; vendor variations and device idiosyncrasies must be incorporated.

We demonstrate that these challenges can be met; we will describe our MITTSSD design in the LightNVM/OCSSD management. To examine MITTSSD can benefit applications, we study data-parallel storage such as distributed NoSQL systems. Examination shows that many NoSQL systems (e.g., MongoDB) do not adopt tail-tolerance mechanisms, and thus can benefit from MITTSSD support.

To evaluate the benefits of MITTSSD in a real multi-tenant setting, we collected statistics of SSD contentions in Amazon EC2, observed from the perspective of a tenant. Our most important finding is that the “noisy neighbor” problem exhibits sub-second burstiness, hence coarse latency monitoring (e.g., snitching) is not effective, but timely latency prediction in MITTSSD will help.

6.2 SSD MANAGEMENT (MITTSSD)

Latency variability in SSD is an ongoing problem [6, 9, 23]. Read requests from a tenant can be queued behind writes by other tenants, or the GC implications (more read-write page movements and erases). A 4KB read can be served in 100µs while a write and an erase can take up to 2ms and 6ms, respectively. While there are ongoing efforts to achieve a more stable latency (GC impact reduction [29, 66] or isolation [31, 37]), none of them cover all possible cases. For example, under write bursts or no idle period, read requests can still be delayed significantly [66, §6.6]. Even with isolation, occasional wear-leveling page movements will introduce a significant noise [31, §4.3].

Fortunately, not all SSDs are busy at the same time, a situation that empowers MITTSSD. A read-mostly tenant can set a deadline of <1ms; thus, if the read is queued behind writes or erases then the tenant can retry elsewhere.
Resource and deadline checks: There are two initial challenges in building MITTSSD. First, Complete Fair Queueing (CFQ)\(^1\) optimizations are not applicable as SSD parallelizes IO requests without seek costs; the use of noop is suggested [4]. In addition, Noop scheduling for disk doesn’t work with SSDs neither. This is because unlike disks where a spindle (a single queue) is the contended resource [16, 44], an SSD is composed of multiple parallel channels and chips. Calculating IO serving time in the block-level layer will be inaccurate (e.g., ten IOs going to ten separate channels do not create queueing delays). Thus, MITTSSD must keep track of outstanding IOs to every chip, which is impossible without white-box knowledge of the device (in commodity SSDs, only the firmware has full knowledge of the internal complexity).

Fortunately, host-managed/software-defined flash [48] is gaining popularity and publicly available (e.g., Linux LightNVM [19] on OpenChannel SSDs [12]). Here, all SSD internal channels, chips, physical blocks and pages are all exposed to the host OS, which also manages all SSD managements (FTL, GC, wear leveling, etc.). With this new technology, MITTSSD in the OS layer is possible.

As an additional note, a large IO request can be striped to sub-pages to different channels/chips. If any sub-IO violates the deadline, EBUSY is returned for the entire request; all sub-pages are not submitted to the SSD.

Performance: Similar to MITTNOOP’s approach, MITTSSD maintains the next available time of every chip (as explained below), thus the wait-time calculation is \(O(1)\). For every IO, the overhead is only 300 ns.

Accuracy: Making MITTSSD accurate involves solving two more challenges. First, MITTSSD needs to know the chip-level read/write latency as well as the channel speed, which can be obtained from the vendor’s NAND specification or profiling. For measuring chip-level queueing delay, our profiler injects concurrent page reads to a single chip and for channel-level queueing delay, concurrent reads to multiple chips behind the same channel. As a result, for our OpenChannel SSD:

\(^1\) I/O scheduling algorithm for disk drives
$T_{\text{chipNextFree}} = 100\mu s$ per new page read. That is, a page (16KB) read takes 100\mu s (chip read and channel transfer); >16KB multi-page read to a chip is automatically chopped to individual page reads. Thus, $T_{\text{wait}} = T_{\text{now}} - T_{\text{chipNextFree}} + (60\mu s \times \#I{\text{OSameChannel}})$. That is, the IO wait time involves the target chip’s next available time plus the number of outstanding IOs to other chips in the same channel, where 60\mu s is the channel queueing delay (consistent with the 280 MBps channel bandwidth in the vendor specification). If there is an erase, $T_{\text{chipNextFree}} = 6$ ms.

Second, while read latencies are uniform, write latencies (flash programming time) vary across different pages. Pages that are mapped to upper bits of MLC cells incur 2ms programming time, while those mapped to lower bits only incur 1ms. To differentiate upper and lower pages, one-time profiling is sufficient. Our profiled write time of the 512 pages of every NAND block is “1111121121122...2112.” That is, 1ms write time is needed for pages #0-6, 2ms for page #7, 1ms for pages #8-9, and the middle pages (“...”) have a repeating pattern of “1122.” The pattern is the same for every block (consistent with the vendor specification); hence, the profiled data can be stored in an 512-item array.

To summarize, unlike disks, SSD internal complexity is arguably more complex (in terms of address mapping and latency variability). Thus, accurate prediction of SSD performance requires white-box knowledge of the device.

### 6.3 Evaluation

We use YCSB [22] to generate 1KB key-value get() operations, create a noise injector to emulate noisy neighbors, and deploy 3 MongoDB nodes for microbenchmarks, 20 nodes for macrobenchmarks, and the same number of nodes for the YCSB client nodes. Data is always replicated across 3 nodes; thus, every get() request has three choices. For MTT-SSD, we only have one machine with an OpenChannel SSD (4GHz 8-core i7-6700K with 32GB DRAM and 2TB OpenChannel SSD with 16 internal channels and 128 flash chips).

All the latency graphs in Figures 6.1 show the latencies obtained from the client get() requests.
6.3.1 MICROBENCHMARK RESULTS

The goal of the following experiments is to show that MITTOS can successfully detect the contention, return EBUSY instantly, and allow MongoDB to failover quickly. We setup a 3-node MongoDB cluster and run our noise injector on one replica node. All get() requests are initially directed to the noisy node.

Figure 6.1 shows the results for MITTSSD (note that we use our lab machine for this one with a local client). First, SSD can serve the requests in <0.2ms (NoNoise). Second, when read IOs are queued behind write IOs (the noise), the latency variance is high (Base); the noise injector runs a thread of 64KB writes. Third, with MITTSSD, MongoDB instantly reroutes the IOs that cannot be served in 2ms (the small gap between Base and MittSSD lines is the cost of software failover).
6.3.2 **MittSSD Results with Amazon EC2 Noise**

For MittSSD, we can only use our single OpenChannel SSD in one machine with 8 core-threads. We carefully (a) partition the SSD into 6 partitions with no overlapping channels, hence no contention across partitions, (b) set up 6 MongoDB nodes/processes on a single machine serving only 6 concurrent client requests, each mounted on one partition, (c) pick noise distributions only from 6 nodes, and (d) set the deadline to the p95 value, which is 0.3 ms (as there is no network hop).

While latency is improved with MITTOS (the gap between MittSSD and Base in Figure 6.2a), we surprisingly found that hedge (Hedged line) is worse than the baseline. After debugging, we found another limitation of hedge (in MongoDB architecture). In MongoDB, the server creates a request handler for every user, thus 18 threads are created (for 6 clients connecting to 3 replicas). In stable state, only 6 threads are busy all the time. But for 5% of the requests (after the timeout expires), the workload intensity doubles, making 12 threads busy simultaneously (note that SSD is fast, thus processes are not IO bound). These hedge-induced CPU contentions (12 threads on an 8-thread machine) cause the long tail. Figure 6.2b shows the resulting % of latency reduction.
6.4 Prediction Accuracy

Figure 6.3 shows the results of MITSSD accuracy tests. For a more thorough evaluation, we use 5 real-world block-level traces from Microsoft Windows Servers (the details are publicly available [36, §III][3]), choose the busiest 5 minutes, and replay them on just one machine. For a fairer experiment, as the traces were disk-based, we re-rate the trace 128× more intensive (128 chips) for SSD tests. For each trace, we always use the p95 value for the deadline.

The % of inaccuracy includes: false positives (EBUSY is returned, but $T_{\text{processActual}} \leq T_{\text{deadline}}$) and false negatives (EBUSY is not returned, but $T_{\text{processActual}} > T_{\text{deadline}}$). During accuracy tests, EBUSY is actually not returned; if error is returned, the IO is not submitted to the device, hence the actual IO completion time cannot be measured, which is also the reason why we cannot report accuracy numbers in real experiments. Instead, we attach EBUSY flag to the IO descriptor, thus upon IO completion, the accuracy can be measured.

Figure 6.3 shows the % of false positives and negatives over all IOs. In total, MITSSD inaccuracy is only up to 0.8%. Without the improvements (§6.2), its inaccuracy can rise up to 6% (no hard-to-predict disk seek time). The next question is how far our predictions are off within the inaccurate IO population. We found that all the “diff”s are <1ms on average, for SSDs. We leave further optimizations as future work.
Chapter 7

Conclusion

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.
Chapter 8

FEMU Limitations & Future Work

8.1 FEMU Limitations

- Limitations: FEMU is DRAM-backed, hence cannot emulate large-capacity SSDs due to DRAM capacity constraints. 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 (§4), there is room for improving accuracy.

The future work of FEMU includes two directions: the first one is to continuously optimize QEMU scalability to support higher level of emulated parallelism. And the second direction is to integrate a sophisticated FTL implementation into FEMU framework and enrich the FTL with new features (e.g. multistreaming support) that’s popular in state-of-the-art SSDs.

8.2 Future Work on Scalability

Current Scalability Solution Limitations: Currently FEMU only utilizes one polling threads for all the NVMe SQ/CQ queue pairs. This can be a bottleneck for contentious I/O workloads. That’s a main reason why FEMU is only able to support 32 parallel channels/chips emulation (bare FEMU without delay emulation can sustain I/O latency less than 52μs for 64 I/O threads, and due to extra
overhead brought by emulating detailed SSD controller internals and FTL, FEMU is finally able to support 32 parallel channels/chips).

**Further Scalability Optimizations:** For next step, we are looking into utilize multiple polling threads (e.g. one polling thread for each SQ/CQ queue pair) to alleviate the problem. To make FEMU more robust, we are also working on moving FEMU’s polling design to use QEMU’s well-defined IOThread API.

Another branch of our scalability optimizations will focus on interrupt coalescing. As describe earlier, FEMU’s polling design eliminates VM-exits for guest-to-QEMU communications, but not for QEMU-to-guest communication path. Everytime QEMU completes an I/O, it needs to interrupt the VM by injecting a virtual interrupt to guest OS, which essentially causes an VM-exit to the guest and pause the VM execution. FEMU polling design makes I/O submissions super fast and thus shift the overhead over to QEMU-guest interrupt injection path. To solve this problem, we are working on implementing an interrupt coalescing algorithm to reduce the number of interrupts needed. An initial idea is to use a time-window based interrupt coalescing mechanism, which only sends interrupt to guest when all I/Os submitted within the same time windows has finished. This would affect individual I/O latency a little bit, but would greatly benefit overall I/O throughput and average latency.

**8.3 Future Work on Fine-granular FTL Features**

**Current FEMU FTL Limitations:** Basically FEMU provides an environment where FTLs can be easily adapted to emulate a blackbox SSD behavior. For an initial start, FEMU FTL is based FTL framework from VSSIM project with extended GC algorithms. The FTL only implements very simple functionalities of Flash management, and may parts of the code is not well-optimized to work with FEMU to achieve best accuracy.

**FEMU FTL Plans:** We are exploring importing other well implemented FTLs (e.g. the one in SSDSim) into FEMU and add new features such as multistream support. This will enable study of
new holistic host/SSD features without needing an expensive real device.

8.4 Others

1. Multi-core support: Today’s SSD controller consists of multiple ARM processors to enable more efficient FTL and I/O handling. With FEMU, we can emulate multi-core by preserving several physical cores for such purposes and only allow FTL threads to run on them. However, this will require the host systems to have enough CPU cores for such purpose.

2. Multi-threaded FTL architecture: Current open-source FTL implementation are all single-threaded, which doesn’t keep up with state-of-the-art commercial SSDs. It’s well known that today’s FTL structures are multi-threaded to improve resource management efficiency.

3. More accurate delay emulation considering more sources of virtualization overheads by integrating guest OS tunings.
References


[27] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and


[38] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model


