Summer 2020

DevT: Let the Device Talk

Chander Bhushan Gupta

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Data Storage Systems Commons

Recommended Citation


This Creative Component is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Creative Components by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
DevT: Let the Device Talk

by

Chander Bhushan Gupta

A Creative Component submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Mai Zheng, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this creative component. The Graduate College will ensure this creative component is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2020

Copyright © Chander Bhushan Gupta, 2020. All rights reserved.
TABLE OF CONTENTS

LIST OF TABLES .......................................................... iv

LIST OF FIGURES ......................................................... v

ACKNOWLEDGMENTS ....................................................... vii

ABSTRACT ................................................................. viii

CHAPTER 1. INTRODUCTION .............................................. 1
  1.1 Motivation ....................................................... 3
  1.2 Related Work ................................................... 5
  1.3 Outline .......................................................... 6

CHAPTER 2. REVIEW OF LITERATURE ................................ 7
  2.1 Why FEMU? ...................................................... 7
  2.2 QEMU Overview ................................................ 8
    2.2.1 Difference between QEMU and KVM ....................... 9
  2.3 Storage Protocol .............................................. 10
    2.3.1 NVMe Protocol ........................................... 10
    2.3.2 SCSI Protocol ........................................... 11

CHAPTER 3. IMPLEMENTATION ........................................ 14
  3.1 System Design .................................................. 14
  3.2 FEMU Installation ............................................. 16
    3.2.1 QEMU Command line interpretation .................... 18

CHAPTER 4. EXPERIMENT AND RESULT .............................. 23
  4.1 System Configuration ......................................... 23
  4.2 Code Changes .................................................. 23
  4.3 Diagnosis ...................................................... 28
  4.4 Overhead ....................................................... 28
    4.4.1 Runtime Overhead ...................................... 30
    4.4.2 Space Overhead .......................................... 30

CHAPTER 5. CHALLENGES AND CONCLUSION ......................... 32
  5.1 Introduction ................................................... 32
    5.1.1 Time Sharing & Synchronization between guest and host 32
    5.1.2 Conclusion ................................................ 39
BIBLIOGRAPHY ................................................................. 40

APPENDIX A. OPCODES FOR NVME AND SCSI ................................. 44

APPENDIX B. WORKLOADS ....................................................... 49
   B.1  SCSI-test ................................................................. 49
   B.2  NVMe-test ............................................................... 52
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td><strong>Filebench Parameters</strong></td>
<td>29</td>
</tr>
<tr>
<td>Table 4.2</td>
<td><strong>Space Overhead</strong> It shows the space overhead of the device trace.</td>
<td>31</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure 1.1</th>
<th>This Figure displays the lines of code that a software is composed of to show the enormous size of complicated softwares.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.2</td>
<td>This Figure [1] shows drivers have an error rate up to 7 times higher than the rest of the kernel.</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>NVMe Architecture [2]</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>6-byte Command Descriptor Block format [3]</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>10-byte Command Descriptor Block format [3]</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>16-byte Command Descriptor Block format [3]</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>System Overview of FEMU and its placement in the Software Stack [4]</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>List of block device as part of QEMU with command line shown in section 3.2.1. Drives in the QEMU command line hd0, hd1, and id0 represents 'sda', 'sdb' and 'nvme0n1'.</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Shows that the command SYNC_CACHE ”53” didn’t reach to the device in case of buggy kernel. command 42 is ”WRITE_10”, 40 is ”READ_10”.</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Shows that the command SYNC_CACHE ”53” reaches to the device in case of patched kernel. command 42 is ”WRITE_10”, 40 is ”READ_10”, 53 is SYNCHRONIZE_CACHE.</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Runtime overhead for collecting SCSI commands. This figure shows the normalized runtime under three workloads with native and DevT enabled.</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Runtime overhead for collecting NVMe commands. This figure shows the normalized runtime under three workloads with native and DevT enabled.</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Shows the placement of vdso in process address space [5]</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Shows the gtod data structure.</td>
</tr>
</tbody>
</table>
Figure 5.3  Snippet of the clocksource shown in the dmesg. . . . . . . . . . 35
Figure 5.4  Flow of the kvm-clock during guest bootup process. . . . . . . . 38
ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research. First and foremost, Dr. Mai Zheng for giving me opportunity to work on this project under his guidance, and supporting patiently throughout this research and the writing of this report. His insights and word of encouragement have often inspired me and renewed my hopes during this journey.

I would also like to thanks my team members in this research work, Om Gatla and Duo Zhang for working with me during various aspects of this project.

Last but not the least, I would like to thank the almighty, my parents and my friends for being a constant support throughout this journey.
ABSTRACT

With the advancement in the Storage systems, the complexity to debug and diagnose the software stack will keep increasing. Every year with addition of new features to the software stack makes the system large and complex. For example, Linux kernel has 27.8 millions LoC and it keeps on growing every year with around 800k commits. One example of complex debugging is the incident that took place at the Algolia [6] data center where Samsung SSDs were blamed whereas the issued lied with kernel. To resolve this problem, engineers struggled over a month to figure out the actual area of fault. Many efforts have been made by the storage community to build tracing tools to address such problems. However, they don’t support the capture of I/O commands at the device level without any real hardware.

To overcome such problem, DevT provides an approach to capture the device commands while leveraging the virtualization. It utilizes the virtual machine and effective emulator to accomplish the tracing task which can help diagnose the problem quickly and effectively at the device level. It generate over all runtime overhead of 0.01% over native run and hence this makes it more efficient with less overhead.
CHAPTER 1. INTRODUCTION

In 2020, Linux kernel enters 27.8 million LoC, including docs, Kconfig files, user-space utilities in-tree, etc, 887,925 commits, and around 21,074 different authors [7], Google has 2 billion LoC, windows approximately 50 million LoC [8]. They still continue to grow adding new features in order to solve new challenges. For example, Linux Kernel grows by approximately 3.5k LoC daily. Such advancement comes with increased complexity of the Software Systems, and these systems need continuous refinement due to the ever-increasing demand in the system performance and efficiency along with inclusion of new features. Figure -1.1 shows the LoC of different softwares. The code flow and its logic are intricate comprising of non-trivial software component interactions, and is specific to operating systems and hardware platforms. To ensure continued support for running applications, software projects rarely remove existing feature. Most Professionals understand the enormity of such systems and effort it takes to maintain them at such wide scale. It creates an enormous challenge for engineers to understand the code and system behavior.

Storage Systems are especially complex due to multiple operations such as I/O operations, page faults, page mapping, swap space, managing file systems, garbage collection and many other operations. Failure in any of the module would lead to the data corruption, service downtime, or irrecoverable data in case of unavailable backup system. Also, Distributed storage systems are especially complex due to the large volume of network communication required to support complex semantics (e.g., POSIX, NFSv4.1). Software developers and engineers face difficulty in debugging and adding new features efficiently to the software due to the complexity of the storage systems. In addition it is complemented by the fact that in many cases original developers of the code move to different projects or companies. Simply reviewing millions of LoC and scanning scarce documentation and commit logs is difficult, inefficient, and typically not sufficient.
Figure 1.1  This Figure displays the lines of code that a software is composed of to show the enormous size of complicated softwares.

System complexity also impacts the users of storage systems, who struggle to understand issues of running applications while accessing the storage system. Maintaining such huge software stacks could be daunting for many professionals and would consume nearly 40 percent to 50 percent, less or more depending upon the severity of the issue, of the project time which may lead to delay in the overall accomplishments. One of the real-world example that had great impact was happened in the Algolia data center [6], where their servers of search API stopped working and they blamed and blacklisted Samsung’s SSD for this failure. After a month of investigation, Samsung found that issue was related to the Linux Kernel and not SSDs. Later they provided the patch for the Linux Kernel. Generally, the reason for such cases are that mostly it is believed that mature software would have no issue and is overlooked by many developers. Both these groups of low level developers and application layer users need better tools to debug and diagnose the storage system to solve issues and improve application performance.
Therefore, it is very essential to have an apt tracing tool during the development and testing. It helps developers or users with the following cases: (a) Verification of the code, (b) Performance analysis, (c) bug analysis, (d) workload characterization, and (e) detailed understanding of the storage system behavior.

In the development of the tracing that is presented in this work is fundamentally based on the tracing of the IO commands and logging them along with their time-stamp. It provides a great insight that which command is being received at what time. This knowledge can be used by the developer to understand the issues related with the low level system such as communication failure between the Linux kernel device driver and the peripheral storage device. Mostly such tracing are done through the bus analyzer [9] which is a hardware system that helps in tracing the commands and used for test, verification of the drivers. The approach used in this is naive and is software equivalent of the bus analyzer.

1.1 Motivation

Even for professionals, System Level tracing has been daunting, there are not enough tools for tracing the system IO commands. One such example for the tracing is bus analyzer, being a hardware tool it is very accurate but at the same time it is not cost-effective and has certain life limit due to wearing out issues. This makes it difficult for many naive researchers or academician to employ such devices and thus impair the wide-scale research. Bus analyzer is a great tool to understand the device behavior their latency as well as performance. With the advancement and availability of the Virtual Machines and the effective emulators, it has become doable to create a tracer almost equivalent to bus analyzer. It provides an effective approach to understand the working of the storage systems. In case of any fault, it can be checked whether a particular IO command has been received or not by the device from Operating Systems. The device drivers are the one that connects with the peripheral devices such as storage devices and with increase in the complexity of the driver, it is very common to have error rate as high as three to seven times higher than the Linux kernel [1] as shown in figure-1.2.
Figure 1.2 This Figure [1] shows drivers have an error rate up to 7 times higher than the rest of the kernel.

There are multiple testing and tracing tools for the device drivers [10] [11] [12] but not for the underlying storage devices. The reason for these drivers being complex is that the developers need greater understanding of both hardware and the software where drivers create a bridge between OS kernel and the devices. Some important tasks that are done by device drivers are like initializing device, interpreting commands from OS, schedule multiple outstanding requests, manage data transfers, accept and process interrupts, and maintain integrity of driver and kernel data structure. It is possible when device driver send these IO commands it may not reach to the device. It is one of the major issue to misjudge and blame the devices for error considering that kernel or its drivers may not have any issue due to their maturity. For example a real world scenario occurred at Algolia data center where they blamed the Samsung SSD’s for the fault happened with their server API. Later, it was found that the issue was with the Linux kernel and not with the SSD. This is natural in practice as users often have to access block devices via system calls provided by the kernel, and do not have the luxury of inspecting the device behavior directly. These issues can easily lead to scenarios such as data loss where a write command not detected by the device lead to failure and could not store the data. It won’t guarantee data consistency and can also cause data
corruption due to faulty read or no read. Therefore, it is very crucial if we could trace the incoming commands in order to test and verify the inclusion of new features or trace the existing bug. The approach used here to trace and log the commands is naive which makes it more important. To achieve this, FEMU [13] has played a very significant role which is used for emulating the real SSD. In the next chapter I will describe the reason for choosing FEMU.

1.2 Related Work

For testing and diagnosing the software stacks many tools have been introduced by the developers to analyze the underlying behaviour of the system which is generally a tedious task. For example, IOPin [14] is meant for dynamic instrumentation which helps understand the complex interaction across I/O layers from application to parallel file system. However, does not capture the device level commands. Another example like EXPLODE [15] and B3 [16] apply fault injections to detect crash-consistency bugs in file systems. However none of them can really capture or deal with the device level commands.

- **Debuggers** [17; 18; 19] These debuggers are very effective in diagnosing the system failures and usually involve lot of manual inspection time. They provide fine-grained results such as memory leakage, breakpoints. However, the effort of diagnosing the storage software stack would increase with the increase in the complexity. In addition to that they work more on the host-side and does not deal with the collection of the device commands directly.

- **Software Tracers** [20; 21; 22; 23; 14; 10; 11; 12] can collect various events from a target system to help understand the behavior. However, similar to debuggers, they focus on host-side events only, and usually do not have device level command tracing system.

- **Bus Analyzers** [24; 9] are hardware equipments that can capture the communication data between a host system and a device, which are particularly useful for analyzing the device behavior. This work is closely related with my work. However, being a hardware component it is costly, have wear and tear over time and usage. But with DevT which is an equivalent
of bus analyzer is the software counterpart which uses virtual machine and device and has less space and runtime overhead that could tell about the commands captured at the device layer.

1.3 Outline

This section provides an overview of the report structure and summary of each chapter.

- **Chapter 1: Introduction** This Chapter talks about the introduction about the tracing, motivation behind achieving device level tracing and some of the related works.

- **Chapter 2: Review of Literature** This chapter is more on the background knowledge needed to understand the project. It describes regarding the reason fro choosing FEMU, overview of the QEMU, difference between QEMU and KVM, and then different storage protocols that are targeted for tracing viz. NVMe and SCSI.

- **Chapter 3: Implementation** This chapter will talk about the system design used in the project, FEMU installation procedure, Hardware specification and Interpretation of the virtual hardware.

- **Chapter 4: Experiment and Result** This chapter talks about the system configuration that has been used in this project, changes that has been done in the code, diagnosis and overhead using filebench workload.

- **Chapter 5: Challenges and Conclusion** This Chapter gives detailed view of the findings that were done to overcome the challenges encountered in this project and then finally concluding with conclusion.
CHAPTER 2. REVIEW OF LITERATURE

In this chapter we will discuss about the reason for choosing the FEMU along with QEMU characteristics and the NVMe and SCSI protocols.

2.1 Why FEMU?

Wide-level research fosters with the availability of proper and economical research facilities. In market different software or hardware systems are available to conduct extensive research on the SSD but they all have certain limitations. For example, hardware research platforms such as FPGA boards [25; 26; 27], OpenSSD [28], or OpenChannel SSD [29] supports full stack software/hardware research but their high cost impairs the large-scale research. In addition to this, to make workable modification, it requires low-level knowledge of System-on-chip and thus make it difficult to use and thus extends the project completion cycle. Although they provide real and correct results, they suffer from the wear out issues with wrong settings. To counter these problems of the hardware we have simulators such as DiskSim’s SSD model [30], FlashSim [31] and SSDSim [32] only support internal-SSD research and not kernel-level extentsions which means that users can only run workload traces to verify new designs. Most of these traces are decade old and are new workloads are not open sourced which again imposes restriction in the research. To overcome the issues of the both simulators and hardware, there are emulators which are essentially software but they imitate the hardware exactly and expose a virtual fake device to the guest OS. Some of the examples are QEMU-based VSSIM (SATA SSDs) [33], FlashEm [34], and LightNVM’s QEMU [35] but unfortunately they are either outdated, not open-sourced.

However, FEMU [13] is a QEMU-based flash emulator, has given four reasons for its acceptance in SSD research.
1. **FEMU** is cheap being open-sourced and has been used and appeared in many project including top storage conferences.

2. **FEMU** is relatively accurate with 0.5 - 38% as a drop-in replacement for OpenChannel SSD. SSD-related kernel changes can be done without real device.

3. **FEMU** is scalable because it can support 32 IO threads and still achieve a low latency. Due to this, it can emulate 32 parallel channels/chips, without unintended queuing delays.

4. **FEMU** is extensible as it can support full stack changes such as only FEMU layer modification (SSD research), only Guest OS modifications on top of unmodified FEMU (Kernel only research), split-level research which includes both Guest OS and FEMU modifications.

**FEMU [13]** 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. As FEMU is based on QEMU [36], it is important to understand what QEMU is capable of and what abilities are provided with the use of the QEMU.

### 2.2 QEMU Overview

QEMU [36] is a generic and open source machine emulator and virtualizer. It can be used as a machine emulator as well as virtualizer. When it is used as a machine emulator, it can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance. When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, 64-bit POWER, S390, 32-bit and 64-bit ARM, and MIPS guests. You can also use QEMU for debugging purposes - you can easily stop your running virtual machine, inspect its state and save and restore it later.

QEMU consists of the following parts:
• processor emulator like x86, s390x, PowerPC, Sparc and few more.

• emulated devices like graphic card, network card, hard drives, mice.

• generic devices used to connect the emulated devices to the related host devices.

• descriptions of the emulated machines like PC, Power Mac.

• debugger

• user interface used to interact with the emulator

QEMU and KVM are related to each other as QEMU uses KVM for leveraging the full virtualization to run the guest code directly onto the host CPU in order to achieve near native performance. However, they both are often confused, hence, it would be great to differentiate both the terms.

2.2.1 Difference between QEMU and KVM

A hypervisor is a software that creates and runs virtual machines (VMs). It is also called virtual machine monitor which is when run on physical hardware (called Host Machine) manages all the resource allocation with the virtual machine (called Guest Machine). It can operate more than one VMs and could share the physical resource or make resources available to VMs as per the requirement. There are 2 types of hypervisor, type 1 and type 2.

• **QEMU** [36] is a type 2 hypervisor when it is operated in userspace mode and emulates all the hardware resources to operate VMs. This approach uses the binary translation and tiny code generator to operate of host machine. But this approach is not so efficient.

• **KVM** [37] is an open source kernel based virtual machine and is a type 1 hypervisor. It is part of Linux kernel module that provides a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.
In our project, we are operating QEMU with the help of KVM [37] that provides near native performance and allows QEMU to use the physical resources with the help of kvm kernel module (kvm.ko).

2.3 Storage Protocol

2.3.1 NVMe Protocol

Earlier storage devices like Hard Disk Drives and Solid State Drives used SATA (Serial ATA) for the interface with the system and it was sufficient back then. However, with the requirement of higher I/O throughput SATA interface could not match with the standards of modern storage devices. To improve this bottleneck, NVMe protocol was designed and developed with a PCIe interface. This improved the optimization of command issue between host and storage devices and supports parallel operation for upto 64k commands within single I/O queue to the device [38].

![NVMe Architecture](image)

Figure 2.1 NVMe Architecture [2]

NVMe protocol is meant to provide greater speed and performance to the modern storage devices. NVMe defines two types of commands Admin commands and IO commands as shown in the Figure-2.1. It maintains two queues Submission Queue (SQ) and the Completion Queue (CQ). During I/O operations the commands are received in the SQ and doorbell register is set. After processing the command it is placed in the corresponding CQ. The host system maintains only one Admin SQ and CQ to manage and control the storage commands whereas it can maintain
maximum of 64k I/O queues of each type. The Admin Queue can store at most 4096 entries, while the depth of I/O Queues is 64K. NVMe maintain the io_uring (SQ and CQ) which is shared with the device that can be accessed by DMA. The head and tail pointer of this ring buffer is maintained using doorbell register. SQ and CQ should work in pair and in general one SQ utilizes one CQ or multiple SQ on one CQ which helps meet high performance in multi-threaded I/O processing.

### 2.3.2 SCSI Protocol

SCSI stands for Small Computer System Interface [3]. SCSI provides the standards to communicate with the SCSI devices using SCSI commands protocol. This communication between linux kernel and the SCSI device is done through the Command Descriptor Blocks (CDBs). These command descriptors vary in sizes based on the the command type which is decided through its opcode.

Below are the 3 different kind of CBD has been shown:

![Figure 2.2 6-byte Command Descriptor Block format](image)

As shown in Figure-2.2, 2.3, and 2.4 each field has its own importance and are described as follows:

1. **Operation Code**: Operation code is a 8 bit field combination of group code and command code which identifies the operation requested by the CDB. Group code is of 3 bits and
command code of 5 bits. Group code specify the type of the command such as 6 byte, 10 byte, 12 byte or 16 byte. In total, 256 codes are available.

2. **Service Action**: This field is part of all CDBs except 6-byte format CDB. It identifies the function to be performed when specified in the Operation Code. In this case, these two fields combine together to identify the operation.

3. **Logical Block Address**: The Logical Block Addresses (LBAs) are of different length for different format and should be contiguous within a particular block or partition till end, generally it starts with 0. The different address format such as 6-byte has 21-bit field, 10-byte and 12-byte may contain 32-bit, and 16-byte CDB allows either 32-bit or 64-bit LBAs.

4. **Transfer Length**: This field describes the number of blocks containing data to be transferred. For example, if 8-bit is used for the Transfer Length then it can process upto 256 blocks or bytes of data for transfer by one command.

5. **Parameter List Length**: There are different types of parameters that can be send to other devices such as mode, diagnostic or log parameters from Data-out buffer. In case this field is 0, then number of bytes to be transferred is also 0 unless otherwise specified.
6. **Allocation Length**: It is a very important field which specifies the allocation of maximum number of bytes in Data-In-Buffer for an application client. In case transfer information exceeds the allocation length, the device server will not send the data and terminate the commands with CHECK CONDITION status and key set to ILLEGAL REQUEST.

7. **Control**: The Control field is a mandatory and same in case of all the commands. It is of 8-bit comprising 2-bit for vendor specific, 3 reserved bit, 2 Obsolete bit and 1 NACA bit also known as Normal ACA.
CHAPTER 3. IMPLEMENTATION

This chapter will talk about the system design used in the project, installation procedure, Hardware specification and its further implementation.

3.1 System Design

We have already discussed about the reason for choosing FEMU for our research but it is important to understand placement of FEMU in the software stack and how it interacts with the other software systems. FEMU [13] was included in QEMU to support the extensive scalable virtual SSD emulator. It was implemented with 3929 LoC in version 2.9 and act as a virtual block device to the Guest OS. A typical software/hardware stack for SSD research is Application + Host OS + SSD device whereas FEMU alters the stack and replaces SSD devices with itself emulating the SSD device viz. Application + Guest OS + FEMU.

![System Overview of FEMU and its placement in the Software Stack](image)

Figure 3.1 System Overview of FEMU and its placement in the Software Stack [4]
FEMU operates in two modes:

- **Whitebox mode:** In this mode FEMU acts like OpenChannel.

- **Blackbox mode:** This mode incorporates the functionality of FTL (Flash Translation Layer) inside SSD controller.

It can be seen in Figure 3.1 that we have underlying physical hardware with the virtualization extension for Intel x86 architecture (Intel-VT). KVM being part of the host Linux kernel module supports the QEMU execution with near-native performance or bare metal performance by directly executing the x86 commands. QEMU contains the userspace APIs of the KVM module to interact with it. On executing QEMU command shown below,

```bash
sudo x86_64-softmmu/qemu-system-x86_64 \
   -name "FEMU-blackbox-SSD" \
   -D test_log.txt \
   -enable-kvm \
   -cpu host \
   -smp 4 \
   -m 4G \
   :
   :
   :
```

which means QEMU runs x86-64 based architecture CPU, with the help of KVM support (hardware based virtualization). QEMU sends the ioctl() command to the KVM to create VM that runs Guest OS and the attached FEMU module [4].
3.2 FEMU Installation

For the installation of FEMU, below instruction are the guidelines:

```
1. git clone https://github.com/ucare-uchicago/femu.git
2. cd femu
3. mkdir build-femu
4. # Switch to the FEMU building directory
   cd build-femu
5. # Copy femu script
6. cp ../femu-scripts/femu-copy-scripts.sh .
7. ./femu-copy-scripts.sh .
8. # only Debian/Ubuntu based distributions supported
9. sudo ./pkgdep.sh
```

Compile and Install FEMU:

```
./femu-compile.sh
```

FEMU binary will appear as x86\_64-softmmu/qemu-system-x86\_64

To prepare the VM image there are two option:

- **Option 1**: To use the already built VM image provided by the FEMU developers. It can be found at FEMU-VMimage-site, it requires you to fill the form and in return it will mail you a link of the VM image. Once you download the image save it at $HOME/images/u14s.qcow2.

- **Option 2**: To build own VM image. Please follow the instruction provided by the link referenced here. Once the guest OS is installed, it is important to redirect VM output to the console, instead of using the separate GUI windows. Inside your guest Ubuntu server, edit /etc/default/grub, make sure the following options are set.

```
GRUB_CMDLINE_LINUX="ip= dhcp console/ttyS0,115200 console= tty console= ttyS0"
GRUB_TERMINAL=serial
GRUB_SERIAL_COMMAND="serial --unit=0 --speed=115200 --word=8 --parity=no --stop=1"
```
Note: For performance reasons, it is suggested by the FEMU developers to use a server version guest OS [e.g. Ubuntu Server 16.04, 14.04]).

FEMU is set to run now. If a Desktop version guest OS is used, it is needed to remove "-nographics" command option from the running script before running FEMU. FEMU can be run in two modes as described before, whitebox and blackbox. Each mode has its own script which can be found in the folder "build-femu" creating while installing the FEMU. Boot the VM using the following script.

Run FEMU as an emulated **blackbox SSD** (device-managed FTL)
```
./run-blackbox.sh
```

Run FEMU as an emulated **whitebox SSD** (OpenChannel-SSD)
```
./run-whitebox.sh
```

**Login to FEMU VM:** If everything is correctly set up then a text-based VM appears on the same terminal where script was ran. It is more convenient to run commands on FEMU through ssh. To do so If you correctly setup the aforementioned configurations, you should be able to see text-based VM login in the same terminal where you issue the running scripts. FEMU running script has mapped host port 8080 to guest VM port 22. ssh into the VM via below command line. (Please run it from your host machine)
```
$ ssh -p8080 $user@localhost
```

FEMU can be also run without SSD logic emulation ./run-nossd.sh In this nossd mode, no SSD emulation logic (either blackbox or whitebox emulation) will be executed. Base NVMe specification is supported, and FEMU in this case handles IOs as fast as possible. It can be used for basic performance benchmarking.

# Still in the VM, update the grub

```
$ sudo update-grub
```
3.2.1 QEMU Command line interpretation

QEMU command line defines the **Virtual Machine hardware** as shown below.

```bash
sudo x86_64-softmmu/qemu-system-x86_64 \
   -name "FEMU-blackbox-SSD" \ 
   -D test_log.txt \ 
   -enable-kvm \ 
   -cpu host \ 
   -smp 4 \ 
   -m 4G \ 
   -device virtio-scsi-pci,id=scsi0 \ 
   -device scsi-hd,drive=hd0 \ 
   -device scsi-hd,drive=hd1 \ 
   -drive file=$OSIMGF,if=none,aio=native,cache=none,format=qcow2,id=hd0 \ 
   -drive file=$NVMEIMGF1,if=none,aio=threads,format=raw,id=hd1 \ 
   -drive file=$NVMEIMGF,if=none,aio=threads,format=raw,id=id0 \ 
   -device nvme,femu_mode=1,drive=id0,serial=serial0,id=nvme \ 
   -net user,hostfwd=tcp::8080-:22 \ 
   -fsdev local,id=shr1,path=/home/chander/test,security_model=passthrough,writeout=writeout \ 
   -device virtio-9p-pci,fsdev=shr1,mount_tag=hostshare \ 
   -net nic,model=virtio \ 
   -nographic \  
   -qmp unix:./qmp.sock,server,nowait 2>&1 | tee stream_output.txt \  
   #-object iothread,id=iothread0 \  
   #-display none \  
   #--nographic \  
   #-monitor stdio \  
   #-s -S \  
   #
```
sudo x86_64-softmmu/qemu-system-x86_64: This defines that it is the QEMU PC System emulator for x86 architecture.

-name "FEMU-blackbox-SSD": Names the type of mode we are using for the SSD emulator which is blackbox in this case. It would differ with whitebox.

-D test_log.txt: It would help to generate the qemu log on the host side with specified file name, in this case it is test_log.txt.

-enable-kvm: Enable KVM full virtualization support. This option is only available if KVM support is enabled when compiling.

-cpu host: Select and simulate CPU model (-cpu help for list and additional feature selection). Here we have host as model.

-smp 4: Simulate an SMP system with 4 CPUs. This number can be changed.

-m 4G: Sets guest RAM size to 4 gigabytes. The default value is 128 MiB.

-device virtio-scsi-pci,id=scsi0: Adds virtio-scsi adapter. virtio-scsi is a advance storage stack for the kvm. KVM supports the SCSI pass-through feature with the virtio-scsi-pci device.

-device scsi-hd,drive=hd0: Add the device scsi-hd with its driver properties mentioned in -drive with identifier hd0.

-device scsi-hd,drive=hd1: Add the device scsi-hd with its driver properties mentioned in -drive with identifier hd1.

-drive file=$OSIMGF,if=none,aio=native,cache=none,format=qcow2,id=hd0: This option appears as the main file "/dev/sda" on executing 'lsblk'. 'file' defines the disk images to be used with the drive. "$OSIMGF" is
stored at "$HOME/images/u14s.qcow2". The qcow2 file format is the
downloaded VM image that is described during FEMU installation.

It can be also created using command:

```bash
qemu-img create -f qcow2 $VDISK_FILENAME 50G
```

The VDISK_filename will be the ubuntu image that can be downloaded
from ubuntu site. The size is 50GB and can be changed.

`if` defines on which type on interface the drive is connected.
Available types are: ide, scsi, sd, mtd, floppy, pflash, virtio, none.
`aio` is native, that selects native Linux AIO.
`format` specify which disk format will be used rather than detecting the
format. It is qcow2 here.

`id` is matched with the -device drive name. It is for scsi-hd device.

```bash
-drive file=$NVMEIMGF1,if=none,aio=threads,format=raw,id=hd1:-
```

this option appears as the main file "/dev/sdb" on executing 'lsblk'.
`file` defines the disk images to be used with the drive. "$NVMEIMGF1" is
stored at "$HOME/images/vssd11.raw". The raw file is created using:

```bash
./qemu-img create -f raw $NVMEIMGF1 1G
```

It would create a image of 1GB.

`if` is none that means no interface.
`aio` is threads, that selects pthread based disk I/O
`format` specify which disk format will be used rather than detecting the
format. Can be used to specify format=raw to avoid interpreting an
untrusted format header.

`id` is matched with the -device drive name. It is for scsi-hd device.

```bash
-drive file=$NVMEIMGF,if=none,aio=threads,format=raw,id=id0:-
```

this option appears as the main file "/dev/nvme0n1" on executing 'lsblk'.
`file` defines the disk images to be used with the drive. "$NVMEIMGF" is
stored at "$HOME/images/vssd1.raw". The raw file is created using:

```bash
./qemu-img create -f raw $NVMEIMGF 1G
```

It would create a image of 1GB.

`if` is none that means no interface.
'aio' is threads, that selects pthread based disk I/O.
'format' is raw.
'id' is matched with the -device drive name. It is meant for nvme device.
-device nvme,femu_mode=1,drive=id0,serial=serial0,id=nvme0:- Add the device
nvme with its driver properties mentioned in -drive with identifier id0.
'femu_mode' 1 defines that FEMU would run as a blackbox

-net user,hostfwd=tcp::8080-:22:- It has mapped host port 8080 to guest VM
port 22.

-fsdev local,id=shr1,path=$HOME/test,security_model=passthrough,writeout=
writeout:-
'fsdev' defines the properties of the exported file system and security
model.
'local' defines that accesses to the filesystem are done by QEMU.
'id' uses the same -device fsdev id as identifier.
'security_model' as passthrough means that files are stored using
the same
credentials as they are created on the guest. It needs QEMU to run as root
.
'writeout' This is an optional argument. The only supported value is
immediate . It means that host page cache will be used to read and
write
data but write notification will be sent to the guest only when the data
has been reported as written by the storage subsystem.

-device virtio-9p-pci,fsdev=shr1,mount_tag=hostshare:- virtio-9p-pci
transports
protocol messages and data between the host and the guest.
we are using 'pci' type with virtio-9p device.
'fsdev' specifies the id value "shr1" used along with -fsdev option.
'mount_tag' specifies the tag name to be used by the guest to mount this
-net nic, model=virtio: - Defines a paravirtualized (model=virtio) network adapter.

-nographic: - This option will turn QEMU into command line application. However, while installing it starts with graphical output. The emulated serial port is redirected to console and muxed with the monitor.

-qmp unix:/qmp-sock, server, nowait 2>&1 | tee stream_output.txt: - It invoke the qemu binary in control mode using qmp, and create a unix socket. qmp allows application like libvirt to communicate with QEMU instance.

Figure 3.2 List of block device as part of QEMU with command line shown in section 3.2.1. Drives in the QEMU command line hd0, hd1, and id0 represents 'sda', 'sdb' and 'nvme0n1'.
CHAPTER 4. EXPERIMENT AND RESULT

This section contains the Experiment and the results obtained by running the benchmark. In this section, the device tracer is built using the FEMU [13], which is acting like SSD emulator for NVMe commands and default scsi device of QEMU.

4.1 System Configuration

The experiment was conducted on the machine with below configuration:

- **CPU**: Intel Xeon E-2174G CPU@3.80GHz.
- **Hard disk**: 2 "WDC WD20SPZX-75UA7T0".
- **RAM**: 64GB.
- **Operating System**: Ubuntu 16.04.1 with kernel v4.15.0-64-generic.

4.2 Code Changes

To trace the command of NVMe-based and SCSI-based devices, both the drivers are changed and includes the QEMU provided API "qemu_log" for logging the commands into a file. The changes are included in files "scsi-disk.c" and "nvme.c" for scsi-commands and nvme-commands respectively. Changes included for both the cases are:

As we already know that there are two types of command handled in the nvme, viz, admin commands and IO commands and therefore, the changes were needed at two places.

```c
/* Change #1 Starts */
#define ADMIN_CMD 0
#define IO_CMD 1
```
static uint64_t get_timestamp_micro() {
    struct timeval tv;
    gettimeofday(&tv, NULL);

    return (tv.tv_sec *(uint64_t)1000000+tv.tv_usec);
}

static uint64_t * get_local_time() {
    time_t rawtime;
    struct tm * timeinfo;
    char * fin_time;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    fin_time = asctime(timeinfo);
    fin_time[strlen(fin_time)-1] = 0;
    return fin_time;
}

/* Change #1 Ends */

static void nvme_process_sq_admin(void *opaque) {
    :
    :
    /* Change #1 Starts */
    uint64_t ts = get_timestamp_micro();
    uint64_t *lt = get_local_time();
    qemu_log("%s,%lld,%d,%d,%ld,%lld,%lld,%lld,%lld,%ld,%ld,%ld,%ld,%ld,%ld,%ld,%ld,%d\n",lt,ts,cmd.opcode,cmd.fuse,cmd.cid,cmd.nsids,cmd.res1,cmd.mptr,cmd.prp1,
    cmd.prp2,cmd.cdw10,cmd.cdw11,cmd.cdw12,cmd.cdw13,cmd.cdw14,cmd.cdw15,ADMIN_CMD);
    /* Change #1 Ends */
    :
}

static void nvme_process_sq_io(void *opaque)
{

    /* Change #1 Starts */
    uint64_t ts = get_timestamp_micro();
    uint64_t *lt = get_local_time();
    qemu_log("%s,%lld,%d,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld,%lld
    ,%d\n",lt,ts,cmd.opcode,cmd.fuse,cmd.cid,cmd.nsid,cmd.res1,cmd.mptr,cmd.prp1,cmd.prp2,cmd.cdw10,cmd.cdw11,cmd.cdw12,cmd.cdw13,cmd.cdw14,cmd.cdw15,IO_CMD);
    //qemu_log("%lld,%d,%lld,%lld,%d\n",ts,cmd.opcode,cmd.cdw10,cmd.cdw10<<12,IO_CMD);
    /* Change #1 Ends */

}

Here the structure NVMeCmd that has been captured is of 64-byte (512 bits).

typedef struct NvmeCmd {
    uint8_t opcode;
    uint8_t fuse;
    uint16_t cid;
    uint32_t nsid;
    uint64_t res1;
    uint64_t mptr;
    uint64_t prp1;
    uint64_t prp2;
    uint32_t cdw10;
    uint32_t cdw11;
    uint32_t cdw12;
    uint32_t cdw13;
    uint32_t cdw14;
}
Similarly, these changes were included in the scsi device. In case of scsi, we have emulated commands and the dma (direct memory access) commands. The changes are part of scsi-disk.c

```c
/* Change to Add Timestamp #1 -- Begins */
static uint64_t get_timestamp_micro () {
    struct timeval tv;
    gettimeofday (&tv , NULL);
    return (tv.tv_sec*(uint64_t)1000000+tv.tv_usec);
}

static uint64_t * get_local_time (){ 
    time_t rawtime;
    struct tm * timeinfo;
    char *fin_time;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    fin_time = asctime(timeinfo);
    fin_time[strlen(fin_time)-1] = 0;
    return fin_time;
}
/* Change to Add Timestamp #1 -- Ends */

static int32_t scsi_disk_emulate_command(SCSIRequest *req , uint8_t *buf) {
    
    /* Change #1 Starts */
    /* Device id has been included because there are two */
    /* scsi devices "sda" and "sdb" and as we are performing *
```
* all operations on sdb for tracing the commands, it is *
* important to include dev id otherwise, it would keep *
* printing the commands from the sda as well.           */
if(req->dev->id == 1){
    uint64_t ts = get_timestamp_micro();
    uint64_t *lt = get_local_time();
    qemu_log("%s,%lld,%d,%d,%lld,%d,%d\n",lt,ts, req->cmd.buf[0], req->
        cmd.len, req->cmd.xfer, req->cmd.lba, req->cmd.mode,0);
}
/* Change #1 ends */
:
:
}

static int32_t scsi_disk_dma_command(SCSIRequest *req, uint8_t *buf)
{
    :
    :
    /* Change #1 Starts */
if(req->dev->id == 1){
    uint64_t ts = get_timestamp_micro();
    uint64_t *lt = get_local_time();
    qemu_log("%s,%lld,%d,%d,%lld,%d,%d\n",lt,ts, req->cmd.buf[0], req->
        cmd.len, req->cmd.xfer, req->cmd.lba, req->cmd.mode,1);
}
/* Change #1 ends */
:
:
}

The structure SCSICommand is recorded that is part of SCSIRequest. The "buf" contains all the information as described in section 2.3.2. However, the other fields shown in this structure
contains the already calculated values from the buffer and thus it is not needed to compute the value of buffer explicitly.

```c
#define SCSI_CMD_BUF_SIZE 16

struct SCSICommand {
    uint8_t buf[SCSI_CMD_BUF_SIZE];
    int len;
    size_t xfer;
    uint64_t lba;
    enum SCSIXferMode mode;
};
```

4.3 Diagnosis

In the diagnosis, it is observed that how tracing commands could help us diagnose the problem, the experiment was conducted for the buggy kernel and the non-buggy kernel. With the help of small workload shown in appendix B, the experiment with file synchronizing command `fsync` was conducted. It was observed that the IO command `SYNC\_CACHE`, in case of SCSI, generated through fsync does not reach the device when ran with buggy kernel. On checking with the patched kernel, the `SYNC\_CACHE` command was observed. It is shown in the figure-4.1 that with the buggy kernel the command `SYNC\_CACHE` didn’t reach the device. However, with the patched kernel the sync command was observed, shown in figure-4.2.

4.4 Overhead

For our experiment to show the overhead, Filebench [43] suite has been used. It is a file system and storage benchmark which can generate various workloads that makes it flexible. Its extensive Workload Model Language helps to understand the the I/O behavior on an application. It is good for micro and macro bench-marking, easy to setup and use. It provides different workloads such
Figure 4.1 Shows that the command \texttt{SYNC\_CACHE }”53” didn’t reach to the device in case of buggy kernel. command 42 is ”WRITE\_10”, 40 is ”READ\_10”.

as varmail, webproxy, fileserver, database-server, etc. The instruction to install and use can be found at [43]. For this project, standard workloads varmail, fileserver and webproxy has been used whose parameters are shown in Table-4.1 that are similar to the ones used in previous work. The Filebench workload description [43] is:

- **Varmail**: This workload emulates the simple mail server IO activities with 16 threads by default. It consists of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory.

- **Fileserver**: This workload emulates fileserver IO activity and performs a sequence of creates, deletes, appends, reads, writes and attribute operations on a directory tree.

- **Webproxy**: This workload emulates a simple web proxy server IO activity. It provides create-write-close, open-read-close, and delete operations of multiple files in a directory tree and a file append to simulate proxy log.

<table>
<thead>
<tr>
<th># of files</th>
<th>Mean dir. width</th>
<th>Mean file size</th>
<th># of threads</th>
<th>Mean IO size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10</td>
<td>16 KB</td>
<td>1</td>
<td>16 KB</td>
</tr>
</tbody>
</table>

Table 4.1 Filebench Parameters
Figure 4.2  Shows that the command SYNC_CACHE ”53” reaches to the device in case of patched kernel. command 42 is ”WRITE_10”, 40 is ”READ_10”, 53 is SYNCHRONIZE_CACHE.

4.4.1  Runtime Overhead

The runtime of the DevT is negligible in comparison to the native run which is .01% of it. As we are only tracing and appending the 64 byte of NVMe command and CDB buffer of SCSI separately, it hardly generates a big difference. The runtime overhead can be seen in Figure-4.4 and Table-4.3 for NVMe and SCSI respectively.

4.4.2  Space Overhead

As it has been described in the previous section regarding the amount of data it is storing. It does not make big difference in the space even. Among all the three workloads the varmail has emulated most I/O activity of a simple mail-server which shows in Table-4.2 that also shows amount of bytes taken by other workloads.
Figure 4.3 **Runtime overhead for collecting SCSI commands.** This figure shows the normalized runtime under three workloads with native and DevT enabled.

Table 4.2 **Space Overhead** It shows the space overhead of the device trace.

<table>
<thead>
<tr>
<th>Workload</th>
<th>SCSI</th>
<th>NVMe</th>
</tr>
</thead>
<tbody>
<tr>
<td>varmail</td>
<td>1.69 MB</td>
<td>94 MB</td>
</tr>
<tr>
<td>webproxy</td>
<td>7.7 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>fileserver</td>
<td>5.4 KB</td>
<td>136.8 KB</td>
</tr>
</tbody>
</table>

Figure 4.4 **Runtime overhead for collecting NVMe commands.** This figure shows the normalized runtime under three workloads with native and DevT enabled.
CHAPTER 5. CHALLENGES AND CONCLUSION

In this section, I will be discussing about the challenges that has been encountered in this project followed by the overall conclusion.

5.1 Introduction

As we have traced all the I/O commands in this project and logged them on the host side. It was also needed to link these commands to the kernel traces in order to understand the problem area. The chances are that due to improper functioning of the Linux Kernel device drivers, the underlying storage device may not receive the required opcode for the operation. It may cause ambiguity in finding where the problem lies and give immense trouble in troubleshooting. By linking the commands it would narrow down the specific area for diagnosing reducing the number of man-hours. To synchronize the events of commands in this scenario an existing technique was tried called "Network Time protocol" or ntp. The problem that was encountered through ntp was that it is not so accurate at the micro-second level which causes issues with the synchronization of events. Therefore, to understand the time sharing between host and guest further research has been done that is described in the next section.

5.1.1 Time Sharing & Synchronization between guest and host

To synchronize events based on the timestamps, it is very important to understand what is the source of the time? How does it flow throughout the full system? There are many API that provides the timestamp to be used in applications. One of the widely used API is gettimeofday(). The tracing of this API has been done in Linux Kernel version 4.4.107.

The function gettimeofday() can get the time of the day into the data structure “struct timeval *tv” and gives the number of seconds and microseconds since the Epoch.
We called this API from the user application and a user space application does not call a system call directly from the kernel space. The glibc, which is a standard library provided by GNU project [44] provides API to connect with the system calls. The implementation of the gettimeofday is in archx86_entryvdsovclock_gettime.c source code file. It is not a usual system call and is located in the special area called vDSO. The vDSO (virtual dynamic shared object) is a shared library that kernel automatically maps into address space of all user-space processes as shown in Figure-5.1.

Why is vDSO even needed? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. This is due both to the frequency of the call as well as the context-switch overhead that results from exiting user space and entering the kernel. The gettimeofday internally calls to __vdso_gettimeofday which further calls do_gettime function. In this function, the struct timeval is populated with the values

```c
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```
of the gtod data structure and if do_realtime fails then it falls back on vdso_fallback_gtod() which then calls the __kernel_vsyscall.

```c
struct vsyscall_gtod_data {
  unsigned seq;
  int vclock_mode;
  cycle_t cycle_last;
  cycle_t mask;
  u32 mult;
  u32 shift;

  /* open coded 'struct timespec' */
  u64 wall_time_nsec;
  gtod_long_t wall_time_sec;
  gtod_long_t monotonic_time_sec;
  u64 monotonic_time_nsec;
  gtod_long_t wall_time_coarse_sec;
  gtod_long_t wall_time_coarse_nsec;
  gtod_long_t monotonic_time_coarse_sec;
  gtod_long_t monotonic_time_coarse_nsec;

  int tz_minuteswest;
  int tz_dsttime;
};
```

Figure 5.2  Shows the gtod data structure.

The gtod data structure vsyscall_gtod_data shown in Figure-5.2 is populated by update_vsyscall function. The updation flow is shown below. Figure-?? process address space showing the location of the vdso. Here the tick_set_periodic_handler function is an event handler and handles the interrupt generated by the standard jiffies in the system. As per the system on which these findings were done provides the CLK_TCK of 100 which is number of jiffies per sec and thus provides the interrupt with 10ms duration. It further calls the function update_wall_time which updates the wall time based on the type of clocksource. There are multiple types of clocksource in the linux kernel system such as tsc, hpet, acpi_pm. With the involvement of the the virtual environment one more clock source is added to the available list of the clock source i.e. kvm-clock as shown in the Figure-5.3. Kvm-clock
provides the interface between the guest and the host machine to share the time where it lets the
guest to read the wall clock of the host. The following command would help in finding the current
clocksource of the system.

```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource".
```

The kvm-clock or the paravirtual clock in the guest sets aside a page of its RAM and asks
the host to write time into that page (using an MSR). The host writes a structure containing the
current time to this page — in theory the host updates this page constantly, but in reality that
would be wasteful and the structure is only updated just before reentering the guest after some
VM event.

![Figure 5.3 Snippet of the clocksource shown in the dmesg.](image)

The MSR or model status register are present at the specific address location in memory and two
of these kinds are "MSR_KVM_WALL_CLOCK_NEW" and "MSR_KVM_SYSTEM_TIME_NEW". Although these MSRs are per CPU entity but the effect of this MSR is global. The memory is
expected to hold the copy of following structures:

- For "MSR_KVM_WALL_CLOCK_NEW"

```c
struct pvclock_wall_clock {  
```
The fields have their meaning:

- **version**: guest must check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.
- **sec**: number of seconds for wallclock at time of boot.
- **nsec**: number of nanoseconds for wallclock at time of boot.

**For "MSR\_KVM\_WALL\_CLOCK\_NEW"**

```c
struct pvclock_vcpu_time_info {
    u32 version;
    u32 pad0;
    u64 tsc_timestamp;
    u64 system_time;
    u32 tsc_to_system_mul;
    s8 tsc_shift;
    u8 flags;
    u8 pad[2];
} __attribute__((__packed__)); /* 32 bytes */
```

The fields have their meaning:

- **version**: guest has to check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.
– tsc_timestamp: the tsc value at the current VCPU at the time of the update of this structure. Guests can subtract this value from current tsc to derive a notion of elapsed time since the structure update.

– system_time: a host notion of monotonic time, including sleep time at the time this structure was last updated. Unit is nanoseconds.

– tsc_to_system_mul: multiplier to be used when converting tsc-related quantity to nanoseconds.

– tsc_shift: shift to be used when converting tsc-related quantity to nanoseconds. This shift will ensure that multiplication with tsc_to_system_mul does not overflow. A positive value denotes a left shift, a negative value a right shift. The conversion from tsc to nanoseconds involves an additional right shift by 32 bits. With this information, guests can derive per-CPU time by doing:

\[
\text{time} = (\text{current\_tsc} - \text{tsc\_timestamp})
\]

\[
\text{if (tsc\_shift} \geq 0)
\]

\[
\text{time} \leftarrow= \text{tsc\_shift}
\]

\[
\text{else}
\]

\[
\text{time} \leftarrow= -\text{tsc\_shift};
\]

\[
\text{time} = (\text{time} * \text{tsc\_to\_system\_mul}) \gg 32
\]

\[
\text{time} = \text{time} + \text{system\_time}
\]

– flags: bits in this field indicate extended capabilities coordinated between the guest and the hypervisor. Flag bit= 0, time measures taken across multiple cpus are guaranteed to be monotonic. Flag bit = 1, guest vcpu has been paused by the host.

These structures will be populated by the hypervisor to the 4-byte aligned physical address of these memories. On the guest side these kvm-clock are read during the bootup as shown in the Figure-5.4.
In the whole architecture setup during the flow of the `kvmclock_init`, the function `kvm_get_wallclock` is being called where timespec structure is populated with the wall_clock parameters. In addition to this time since system boot is calculated and added to the timespec structure parameter.

```plaintext
delta = pvclock_clocksource_read(vcpu_time); /* time since system boot */
delta += now.tv_sec * (u64)NSEC_PER_SEC + now.tv_nsec;
```

Figure 5.4  Flow of the kvm-clock during guest bootup process.
Till now the most of the gettimeofday has been traced and some connections are still pending to make such as after the initial setup of the kvm-clock on guest side, how does the guest updates the gtod data structure for gettimeofday API with the kvm-clock value.

5.1.2 Conclusion

DevT is a tracer that traces the I/O commands and helps in diagnosing the storage system devices. It leverages the virtualization to collect the device commands without any dependency on the kernel or any special hardware. The overall runtime overhead that comes with it is very less and is almost equal to 0.01% of the native run and will consume small fraction of the overall space. This will help to know the current issue with the storage device by checking the IO commands received at the device side.

There are some challenges that needs to be done in the future are like having proper synchronization between the host and the guest as described in the section of challenges. Along with this, the things that need focus in future is that in case of NVMe protocol, the sync commands does not reach the nvme driver in FEMU which needs to be fixed.
REFERENCES


APPENDIX A. OPCODES FOR NVME AND SCSI

These are the commands that are part of NVMe.

```c
enum NvmeAdminCommands {
    NVME_ADM_CMD_DELETE_SQ = 0x00,
    NVME_ADM_CMD_CREATE_SQ = 0x01,
    NVME_ADM_CMD_GET_LOG_PAGE = 0x02,
    NVME_ADM_CMD_DELETE_CQ = 0x04,
    NVME_ADM_CMD_CREATE_CQ = 0x05,
    NVME_ADM_CMD_IDENTIFY = 0x06,
    NVME_ADM_CMD_ABORT = 0x08,
    NVME_ADM_CMD_SET_FEATURES = 0x09,
    NVME_ADM_CMD_GET_FEATURES = 0x0a,
    NVME_ADM_CMD_ASYNC_EV_REQ = 0x0c,
    NVME_ADM_CMD_ACTIVATE_FW = 0x10,
    NVME_ADM_CMD_DOWNLOAD_FW = 0x11,
    NVME_ADM_CMD_FORMAT_NVM = 0x80,
    NVME_ADM_CMD_SECURITY_SEND = 0x81,
    NVME_ADM_CMD_SECURITY_RECV = 0x82,
    NVME_ADM_CMD_SET_DB_MEMORY = 0x7c,
    NVME_ADM_CMD_FEMU_DEBUG = 0xee,
};

enum NvmeIoCommands {
    NVME_CMD_FLUSH = 0x00,
    NVME_CMD_WRITE = 0x01,
    NVME_CMD_READ = 0x02,
    NVME_CMD_WRITE_UNCOR = 0x04,
    NVME_CMD_COMPARE = 0x05,
    NVME_CMD_WRITE_ZEROS = 0x08,
    NVME_CMD_DSM = 0x09,
};
```
# define TEST_UNIT_READY 0x00
# define REWIND 0x01
# define REQUEST_SENSE 0x03
# define FORMAT_UNIT 0x04
# define READ_BLOCK_LIMITS 0x05
# define INITIALIZE_ELEMENT_STATUS 0x07
# define REASSIGN_BLOCKS 0x07
# define READ_6 0x08
# define WRITE_6 0x0a
# define SET_CAPACITY 0x0b
# define READ_REVERSE 0x0f
# define WRITE_FILEMARKS 0x10
# define SPACE 0x11
# define INQUIRY 0x12
# define RECOVER_BUFFERED_DATA 0x14
# define MODE_SELECT 0x15
# define RESERVE 0x16
# define RELEASE 0x17
# define COPY 0x18
# define ERASE 0x19
# define MODE_SENSE 0x1a
# define LOAD_UNLOAD 0x1b
# define SCAN 0x1b
# define START_STOP 0x1b
# define RECEIVE_DIAGNOSTIC 0x1c
# define SEND_DIAGNOSTIC 0x1d
# define ALLOW_MEDIUM_REMOVAL 0x1e
# define SET_WINDOW 0x24
# define READ_CAPACITY_10 0x25
# define GET_WINDOW 0x25
# define READ_10 0x28
# define WRITE_10 0x2a
# define SEND 0x2a
# define SEEK_10 0x2b
# define LOCATE_10 0x2b
# define POSITION_TO_ELEMENT 0x2b
# define WRITE_VERIFY_10 0x2e
# define VERIFY_10 0x2f
# define SEARCH_HIGH 0x30
# define SEARCH_EQUAL 0x31
# define OBJECT_POSITION 0x31
# define SEARCH_LOW 0x32
# define SET_LIMITS 0x33
# define PRE_FETCH 0x34
# define READ_POSITION 0x34
# define GET_DATA_BUFFER_STATUS 0x34
# define SYNCHRONIZE_CACHE 0x35
# define LOCK_UNLOCK_CACHE 0x36
# define INITIALIZE_ELEMENT_STATUS_WITH_RANGE 0x37
# define READ_DEFECT_DATA 0x37
# define MEDIUM_SCAN 0x38
# define COMPARE 0x39
# define COPY_VERIFY 0x3a
# define WRITE_BUFFER 0x3b
# define READ_BUFFER 0x3c
# define UPDATE_BLOCK 0x3d
# define READ_LONG_10 0x3e
# define WRITE_LONG_10 0x3f
# define CHANGE_DEFINITION 0x40
# define WRITE_SAME_10 0x41
# define UNMAP 0x42
# define READ_TOC 0x43
# define REPORT_DENSITY_SUPPORT 0x44
# define GET_CONFIGURATION 0x46
# define SANITIZE 0x48
# define GET_EVENT_STATUS_NOTIFICATION 0x4a
# define LOG_SELECT 0x4c
# define LOGSENSE 0x4d
# define READ_DISC_INFORMATION 0x51
# define RESERVE_TRACK 0x53
# define MODE_SELECT_10 0x55
# define RESERVE_10 0x56
# define RELEASE_10 0x57
# define MODESENSE_10 0x5a
# define SEND_CUE_SHEET 0x5d
# define PERSISTENT_RESERVE_IN 0x5e
# define PERSISTENT_RESERVE_OUT 0x5f
# define VARLENGTH_CDB 0x7f
# define WRITE_FILEMARKS_16 0x80
# define READ_REVERSE_16 0x81
# define ALLOW_OVERWRITE 0x82
# define EXTENDED_COPY 0x83
# define ATA_PASSTHROUGH_16 0x85
# define ACCESS_CONTROL_IN 0x86
# define ACCESS_CONTROL_OUT 0x87
# define READ_16 0x88
# define COMPARE_AND_WRITE 0x89
# define WRITE_16 0x8a
# define WRITE_VERIFY_16 0x8e
# define VERIFY_16 0x8f
# define PRE_FETCH_16 0x90
# define SPACE_16 0x91
# define SYNCHRONIZE_CACHE_16 0x91
# define LOCATE_16 0x92
# define WRITE_SAME_16 0x93
# define ERASE_16 0x93
# define SERVICE_ACTION_IN_16 0xe
# define WRITE_LONG_16 0xf
# define REPORT_LUNS 0xa0
# define ATA_PASSTHROUGH_12 0xa1
# define MAINTENANCE_IN 0xa3
# define MAINTENANCE_OUT 0xa4
# define MOVE_MEDIUM 0xa5
# define EXCHANGE_MEDIUM 0xa6
# define SET_READ_AHEAD 0xa7
# define READ_12 0xa8
# define WRITE_12 0xaa
# define SERVICE_ACTION_IN_12 0xab
# define ERASE_12 0xac
# define READ_DVD_STRUCTURE 0xad
# define WRITEVERIFY_12 0xae
# define VERIFY_12 0xaf
# define SEARCH_HIGH_12 0xb0
# define SEARCH_EQUAL_12 0xb1
# define SEARCH_LOW_12 0xb2
# define READ_ELEMENT_STATUS 0xb8
# define SEND_VOLUME_TAG 0xb6
# define READ_DEFECT_DATA_12 0xb7
# define SET_CD_SPEED 0xbb
# define MECHANISM_STATUS 0xbd
# define READ_CD 0xbe
# define SEND_DVD_STRUCTURE 0xbf
APPENDIX B. WORKLOADS

B.1 SCSI-test

```c
#define _XOPEN_SOURCE 500
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>

int main() {
    int fd, ret, offset, i;
    char buf[5], read_buf[5];
    //struct stat st;
    printf("pid = %d\n", getpid());
    sleep(30);
    fd = open("/dev/sdb", O_RDWR);
    if(fd < 0){
        perror("open: ");
        return 1;
    }

    for(i = 0; i<5; i++)
    {
```
buf[i] = 'a';

// pwrite and fsync
offset = 0;
ret = pwrite(fd, &buf, 5, offset);
printf("The written value id %d\n", ret);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("write fsync done\n");
}
ret = 0;
ret = pread(fd, &read_buf, 5, offset);
printf("The read value is %d\n", ret);
if(ret < 0) {
    perror("pread: ");
    return 1;
} else {
    printf("pread [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("read fsync done\n");
}
offset = 200;
ret = pwrite(fd, &buf, 5, offset);
printf("The written value id %d\n", ret);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("write fsync done\n");
}
offset = 400;
ret = pwrite(fd, &buf, 5, offset);
printf("The written value id %d",ret);
    if(ret < 0) {
        perror("pwrite: ");
        return 1;
    } else {
        printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
        fsync(fd);
        printf("write fsync done\n");
    }
}
offset = 600;
ret = pwrite(fd, &buf, 5, offset);
printf("The written value id %d",ret);
    if(ret < 0) {
        perror("pwrite: ");
        return 1;
    } else {
        printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
        fsync(fd);
        printf("write fsync done\n");
    }
}
offset = 800;
ret = pwrite(fd, &buf, 5, offset);
printf("The written value id %d",ret);
    if(ret < 0) {
        perror("pwrite: ");
        return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("write fsync done\n");
}

// cleanup
close(fd);
return 0;
}

B.2 NVMe-test

#define _POSIX_SOURCE
#include <sys/stat.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>

int main() {
    int fd, ret, offset;
    char buf[4] = "abc";
    char read_buf[4];
    int x;
    scanf("%d", &x);
    fd = open("/dev/nvme0n1", O_RDWR);
    if(fd < 0){
        perror("open: ");
        return 1;
    }
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("write fsync done\n");
}

// cleanup
close(fd);
return 0;
// pwrite and fsync
offset = 0;
ret = pwrite(fd, &buf, 4, offset);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]
", ret, offset);
    fsync(fd);
    printf("fsync done\n");
}
ret = pread(fd, &read_buf, 4, offset);
if(ret < 0) {
    perror("pread: ");
    return 1;
} else {
    printf("pread [%d] bytes at offset [%d] is %s
", ret, offset,
read_buf);
    fsync(fd);
    printf("fsync done\n");
}
ret = pwrite(fd, &buf, 4, offset);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("fsync done\n");
}
ret = pwrite(fd, &buf, 4, offset);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("fsync done\n");
}
ret = pwrite(fd, &buf, 4, offset);
if(ret < 0) {
    perror("pwrite: ");
    return 1;
} else {
    printf("pwrite [%d] bytes at offset [%d]\n", ret, offset);
    fsync(fd);
    printf("fsync done\n");
}
close(fd);
return 0;