Florida State University Libraries

Electronic Theses, Treatises and Dissertations | The Graduate School

2017

I/O Latency in the Linux Storage Stack

Brandon Stephens

Follow this and additional works at the DigiNole: FSU's Digital Repository. For more information, please contact lib-ir@fsu.edu

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

I/O LATENCY IN THE LINUX STORAGE STACK

By

BRANDON STEPHENS

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2017

Copyright © 2017 Brandon Stephens. All Rights Reserved.

Brandon Stephens defended this thesis on November 15, 2017.
The members of the supervisory committee were:

An-I Andy Wang
Professor Directing Thesis

Zhi Wang
Committee Member

David Whalley
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

My advising professor, Dr. Andy Wang, has been impressively patient with me throughout this journey. Without his guidance, I might have spent the last year falling down abyssal rabbit holes of issues only tangentially related to my research scope.

Dr. David Whalley was responsible for obtaining the IRES research grant through the NSF which allowed several students from FSU's computer science department to research at Chalmer's University in Gothenberg, Sweden. Because of my interactions with Dr. Whalley and the experience he offered me as a whole, I discovered a passion for traveling and eating international food.

Dr. Zhi Wang provided me with my first pleasant experience in the operating systems realm when I took his course during my undergraduate career. I distinctly remember Dr. Zhi Wang having a clear and concise answer for every question in that course. During my graduate career, I was fortunate enough to work as a teaching assistant under him for the very same class.

I am grateful to Dr. Mark Stanovich, for whom I acted as a research assistant during my later undergraduate career, and to my friend Ruud, who broadened my perspective about computer science in general. Mark and Ruud both were influential in my decision to pursue graduate studies.

Thanks to the many family members and friends that have stimulated my interest in the sciences over the years, and to my biggest cheerleader, Hyery, whose presence alone eased my stress throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As storage device performance increases, the lifespan of an I/O request becomes throttled more-so by data path traversal than physical disk access. Even though many computer performance analysis tools exist, a surprisingly small amount of research has been published documenting bottlenecks throughout the Linux storage stack. What research has been published focuses on results found through tracing, glossing over how the traces were performed. This work details my process of developing a refined tracing method, what that method is, and how the research can be applied to measure I/O latency at any layer of the Linux storage stack. Sample results are given after examining the filesystem layer, the block layer, and the memory management system. Among these three components of the storage stack, the filesystem layer is responsible for the longest duration of an I/O request's lifespan.

CHAPTER 1

INTRODUCTION

As technology improves, storage devices such as hard-disk drives (HDDs) and solid-state drives (SSDs) are able to operate more quickly and efficiently. The use cases of computers adapt with the times, thus broadening the definition of "general" when talking about the general use cases of machines. Modern operating systems are under constant development to keep up with these evolving use cases. For some operating systems, such as Linux, this can lead to higher complexity within the kernel. The performance increase of hardware, coupled with more complicated kernels, results in an I/O request's lifespan spending a majority of its time traversing down to the physical layer of the storage stack rather than at the physical layer (see Section 1.1).

This work explores the life of a write request through ext4, the memory management subsystem and the block layer of the Linux storage stack, seeking to discover where the write request spends the majority of its time. This is determined by tracing the write request; monitoring all of the functions called throughout the filesystem, memory management, and block layers from the time a write request was issued to the time it is written to the target storage device.

The scope of this work has evolved over the last year, initially seeking to explicitly and definitively state the bottlenecks of I/O requests throughout the Linux storage stack. Although the scope has been reduced over time to exploring write requests within a few layers of the storage stack, the techniques described within could be used for tracing any type of I/O request throughout any layer.

1.1 The Linux Storage Stack

For the purpose of this research, I refer to the Linux storage stack as a sequence of layers that I/O passes through from a user-space application all the way down to a physical storage device, and potentially back up. A simplified version of the Linux storage stack can be see in Fig 1.1. The application layer is where applications built for user interaction live. The virtual filesysem (VFS) layer acts as an abstraction between the application and filesystem layers, providing a consistent user experience despite which filesystem is actually in use. A bit lower down the stack is the SCSI

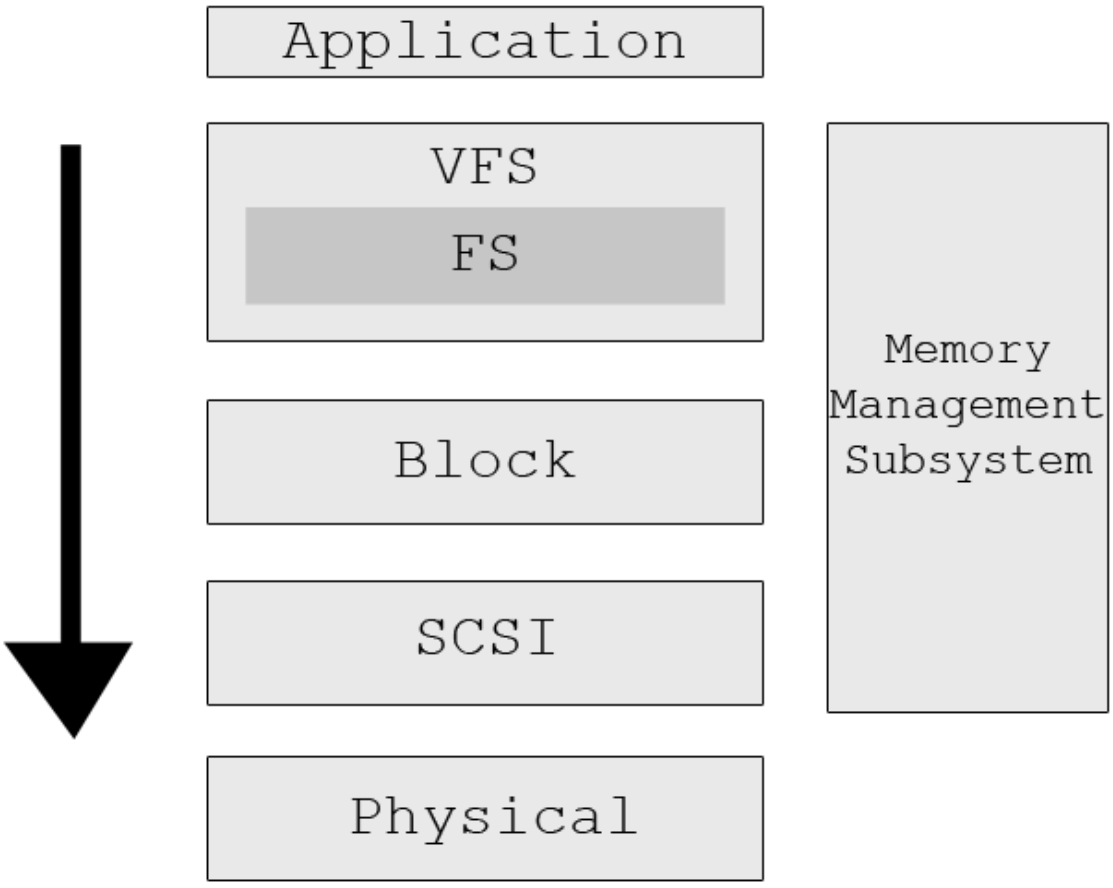

Figure 1.1: A simplified depiction of the Linux storage stack.

layer, responsible for direct communication with SCSI devices (the physical layer). My research focuses on the filesystem layer, the memory management subsystem, and the block layer, each of which are described below.

1.1.1 Filesystem Layer

A filesystem (FS) is usually thought of as two separate pieces; a collection of files and a directory structure. Files typically contain some content and attributes related to that content (often called metadata). Directory structures provide ways to traverse paths between files or retrieve information about groups of files on a system. How and what information is stored and retrieved is dependent on a filesystem's implementation. My research currently only considers the well-known ext4 filesystem (which is the default filesystem of many Linux distributions). Because I/O requests originate from

the application layer and must pass through the filesystem layer, the filesystem layer was chosen as a tracing candidate.

ext4. Ext4 is the fourth iteration of the extended filesystem, which was a created specifically for the Linux kernel. It is a type of journaling filesystem, meaning it maintains a log (journal) of committed metadata changes. Such a feature is useful in reducing corruption in the event of power loss, though it also affects performance (which is considered in this work as "interference", described in Section 2.3.4).

Although this work is not a study on ext4, our results partially rely on the fact that this filesystem implements delayed allocation (also called "allocate-on-flush"). This means that instead of allocating blocks per write request, ext4 will wait on a group of writes to arrive before it begins any block allocation. How long this delay takes is dependent on memory pressure-related system thrashing; the more memory pressure, the longer write requests have to wait for block allocation.

1.1.2 Memory Management Subsystem

The memory management (MM) subsystem is in charge of handling address spaces, memory mapping, physical and virtual memory allocation, and some protection services of such allocated memory. After a write or read request has traversed the filesystem layer, FS-agnostic generic routines (literally, routines that begin with `generic_`) are called. Such generic routines are implemented within `/mm/filemap.c`, a component of the memory management subsystem. These routines ultimately call functions which live at the block layer, making the memory management subsystem a good candidate to be traced alongside the filesystem and block layers.

Typically, the memory management subsystem is referred to as a "subsystem" rather than a "layer" due to its functionality being spread across multiple layers. In this work, however, "memory management subsystem" and "memory management layer" are used interchangeably, as it is useful to think of the `generic_` interface as a separate layer between the filesystem and block layers.

1.1.3 Block Layer

The block layer cannot be understood without first having an understanding of block devices. We can consider traditional storage mediums (HDDs, SSDs, RAM) as being hardware devices. A block device should be thought of more like a file or software device (be it physical or logical) which allows the Linux kernel to interact with hardware devices in an abstract way; acting as a general

mechanism for translating kernel requests into hardware specific instructions. Importantly, block devices allow for this interaction to be buffered/cached, leaving users with no way to determine how long it will take for a write request to physically be written to the intended storage medium. With that in mind, the block layer can be understood as the layer in charge of managing the kernel's interactions with block devices.

Functionally, the block layer manages multiple I/O queues. If a write request is generated in user space, it will eventually have to hit the block layer (whether using direct I/O or not), where the request will be stored as a series of fixed-size blocks. Any I/O request that ends up at the block layer will be submitted to block devices through a scheduler. As such, I/O schedulers are considered part of the block layer. Due to the block layer's location among the I/O path (namely, being directly above the SCSI layer), it was chosen as a candidate to be traced.

1.2 Storage Mediums

Hard-disk Drives (HDDs, which are based on rotating disks) have been the most common type of storage medium for decades. They are cheaper than other storage solutions, though they also have high latency costs. Solid State Drives (SSDs, a form of electronic storage) operate with a much higher efficiency than HDDs, offering a 20x (or more) speedup, though at a higher price. In the context of this work, SSDs refer to the NAND flash non-volatile memory (NVM) variants. RAM Disks (RDs) are only as expensive as RAM and operate at an extremely high efficiency. RDs are a type of volatile storage, meaning that they cannot maintain data without power. We later show that RDs can reduce latency across certain storage layers by over 5000x when compared to HDDs and over 200x when compared to SSDs (Table 4.2).

1.2.1 FPGA Boards

SSDs are proprietary by nature. This means that users have no control over certain aspects of these devices, such as configuring the flash translation layer (FTL) of a device. Seeking to take advantages of the lessons learned by He et al's proposed five critical rules for using SSDs [1], we looked into using various flash-programmable gate array (FPGA) boards.

One such board was the Cosmos OpenSSD, which is based on the HYU Tiger3 controller (implemented by a Zynq-7000 FPGA). This board would have suited our needs, but we quickly

learned that FPGA setup is not always quick or easy. In this particular case, we had to interact with the company responsible for creating the software suite that would allow us to configure and modify our OpenSSD. The software would not interact without first owning the proper license. Luckily, the company offered licenses for academic purposes. Unluckily, the process of obtaining an academic license took several months. Once the license had been approved and delivered, we noticed that it did not unlock features within the software which were necessary for our board development. Communications with the company's support team yielded no solution, so work on the OpenSSD had to be scrapped.

1.3 Related Work

Part of the importance of this paper is that there is a surprisingly small amount of published research information available on layer-specific latency. There was one set of slides (without a corresponding published paper) presented at the Linux Storage & Filesystem Workshop of 2008, where Accardi et al use `iolat` to show that the SCSI layer is the most time consuming layer for direct and cached I/O on non-RAM disk storage mediums [13]. On a RAM disk with direct I/O, they state that the FS and MM layers make up more than half of the lifespan of I/O. The dominance of these layers within an I/O's lifespan is why I have chosen to focus on them. `iolat` is a tool lacking useful documentation, though it appears to be a workload generator and IOPS measuring tool.

Swanson and Caulfield suggest that the Linux storage stack accounts for 0.3% of the latency cost when dealing with 4KB accesses to hard disk drives, but that it accounts for up to 70% of latency for SSDs [2]. The authors look into ways of maximizing NVM performance based on techniques of refactoring (spreading storage system functionality across multiple layers), reducing (minimizing software operations needed to perform certain tasks), and recycling (re-using existing software components). The authors do not report layer-specific latency issues.

CHAPTER 2

POSSIBLE APPROACHES

Computer performance analysis is not a new field. As such, there are many tools available to assist analysts in studying the various behavioral characteristics of their machines as a whole, as well as the individual components of a machine, software, etc. The need to debug, monitor, assess, and improve performance of the Linux kernel has resulted in numerous frameworks to assist analysts with their work. Perhaps the two most popular in tracing the Linux kernel are tracepoints (static kernel tracing) [4] and kprobes (dynamic, "on-the-fly" kernel tracing) [5]. Modern Linux kernels can easily take advantage of tracepoints using the built-in framework, ftrace.

2.1 Ftrace

Ftrace (or "ftrace"), short for "function tracer," is a framework of different tools and utilities that operate within kernel space [10], which has been built into the kernel since version 2.6.27. Ftrace can be used for latency tracing, static event tracing, interrupt tracing, generating call graphs, and for many other purposes. With its suite of available tools and ease of access (being built into modern kernels), ftrace was a good candidate to apply towards the research goal.

Because ftrace is operated through the debugfs filesystem (which provides kernel developers a way to access kernel information through the user space [3]), its usage is not completely intuitive. Its interface exists as a collection of files which must be updated (via `write()`, `echo`, or some equivalent) in order to change a trace's configuration. If one were to browse ftrace's home directory, `/sys/kernel/debug/tracing`, they would find multiple configuration options: a file to determine whether tracing is turned on or off, a file listing the current event tracer, a file containing white-listed functions that will appear in the trace, etc. Although it sounds simple, there are ultimately many ways to improperly configure a trace without realizing it. With that in mind, one should try not to touch any file inside of this directory without knowledge of the file's purpose and the consequences associated with modifying its contents.

2.1.1 Applying Ftrace

During this research's infancy, I wrote a script which interacted with ftrace for me, being careful to touch only the minimal number of ftrace configuration files. The script would first ensure that tracing was disabled before proceeding. This meant that any other number of initialization operations could be performed without generating any extra trace data. Next, the script removed any previously existing data stored in ftrace's output file to ensure that the next set of trace data did not contain residual content. After configuring the tracer type and enabling any extra options (such as telling ftrace to use a latency output format), I was able to safely turn the trace on and run a workload.

Because the goal of using ftrace was to trace I/O throughout the storage stack, the first chosen workload was a simple write of 1MB to the hard disk using `dd`, a file converter/copier. 1MB (about 250 blocks on a machine with a block size of 4096B) was chosen as it should generate a sufficient amount of traffic at each layer to help get an understanding of how ftrace works (as it turns out, it generates more than enough traffic). This 1MB was generated by copying from `/dev/urandom` (a pseudorandom number generator) to a temporary location on a hard drive, via `dd if=/dev/urandom of=/tmp/gibberish bs=1M count=1`. Immediately after `dd` was called, a sync was issued to ensure that the writes would be written to their destination before the trace was turned off.

The method mentioned above could be a convenient way for a user to begin learning ftrace, though it can produce a nasty bug. If a user is modifying the current tracer (through the file located at `/sys/kernel/debug/tracing/current_tracer`) via the `echo` command, some kernel versions (4.4.0 - 4.5.5 and possibly more) will crash after two such modifications are made. This issue can be worked around by creating a C program which writes to the current trace file (thus avoiding `echo`), or it can be patched [6].

Although ftrace's basic functionality can be used in a controlled manner, more complex tracing requires acting with extreme care in how configuration files are changed. Additionally, ftrace is limited by the set of information that it can provide, in that a tracepoint will only record trace data at a function's entrance and exit. Seeking to avoid the limitations, I began looking into building a simple, custom framework that would allow me to print relevant information from any point within a function.

2.2 In-House Framework

Developing an in-house framework to trace the kernel is not necessarily hard, though there are a few considerations that must be made: you must know the proper locations within the kernel to trace your write request, how to identify which write requests in the storage stack belong to the workload you are trying to trace, what kind of information you want to save at each tracepoint, and an efficient method of saving that information.

2.2.1 Workload Generator

To be compatible with the proposed framework, a workload generator must be capable of creating identifiable write requests at any layer of the storage stack. Although `dd` might at first seem like a fine candidate for our write-heavy workload generator, it cannot be easily configured in a way to achieve quick identification. Instead, it is preferable to create a workload generator which embeds a unique key and a sequence number into a write request. This way, we can look for write requests at any layer of the kernel whose payload begins with our unique key. If we need to know which write request of ours we are looking at, then we need only extract the associated sequence number.

There are other methods that could be used to identify writes in the kernel that were generated by a specific process. Namely, the process identifier (PID) of your workload generator can be searched for within a kernel using `task_tgid_nr()` (rather than `task_pid_nr()`, as PIDs seen by users are different from PIDs as seen the kernel). It may also be possible to run a workload using a specific user identifier (UID) and matching that in the kernel with `get_current_user()->uid.val`, which may not work at all layers. A less effective approach would be to generate write requests of an uncommon ("never" used) size, though this method may fail at the block layer where writes are stuffed inside of blocks of uniform size. It may be possible to iterate over the block layer's various I/O vectors and manually calculate the size of a write embedded within blocks, but this is potentially too time consuming at a layer where inefficiency can cause a system hang.

Despite the method for identifying whether a write request is one of interest, the best way to determine the order in which you are processing the write requests is by using the embedded sequence number technique.

2.2.2 Logging Implementation

One benefit to the in-house tracing framework is that you get to choose what information is logged, the format of the log, and the logging method itself. For a simple case which mimics a subset of default ftrace output, one could choose to log a timestamp (using `struct timespec` with `getnstimeofday()`) and the current function's name. If the workload generator embeds a sequence number into the write payload, then that value is also a likely candidate to be stored.

Logging can be unnecessarily expensive depending on implementation. Some in-house tracing frameworks, such as that of Jun Yan et al [12], rely on `printk()` to print to one of the kernel's logs. While this method works well at some higher layers of the storage stack (such as the filesystem layer), it can be troublesome at lower layers where costly functions such as `printk()` can have serious performance impacts.

To avoid the high costs associated with `printk()`, one could instead create and write to a log in memory. When a trace has completed, this log can be flushed somewhere on disk to be analyzed.

2.2.3 Controlling the Trace with a Kernel Module

Without a way to turn a trace on or off, our in-house framework would useless. One such way to do this is by creating a kernel module whose sole purpose is to maintain variables associated with our tracing framework, such as an on/off switch and a log buffer. The kernel module I have written uses the procfs filesystem, although I have recently discovered that kernel developer and maintainer Greg Kroah-Hartman has discouraged this, suggesting that sysfs should be used instead [7].

It is relatively easy to write a limited capability module (such as a toggle) for the procfs filesystem, as only the read and write file operations need to be implemented. Much like ftrace, the user needs only to write a 1 or 0 (via `echo` or otherwise) to the `/proc` module to indicate whether tracing should be turned on or off, respectively. When a user tries to read from the `/proc` module (via `cat /path/to/proc/entry` or similar), it should report whether or not tracing is turned on (again, by displaying a 1 or 0) and, optionally, the path to the trace log (or any other possibly relevant information, to taste).

2.2.4 Problems with This Approach

Although the in-house framework gives an analyst an easily customizable, low-overhead tool without any additional features that may never be used (which could potentially add extra overhead), it is not perfect; this method can be time consuming and have too limited a scope.

Learning how to interact with the kernel is not an easy task. For starters, `/kernel` contains hundreds of thousands of lines of code, `/fs` has over a million lines of code, and `/drivers` has over 10 million lines of code. Even without considering the complexity of kernel data structures, gaining an intimate knowledge of each aspect of the kernel is a process that could very possibly take decades. As such, potential developers or kernel hackers instead need to be able to develop the skill to identify only the parts of the kernel with which they need to handle, a process coupled with a steep learning curve.

Apart from learning how to interact with the kernel, this method has a scoping issue. Specifically, because the kernel is so large and there are potentially thousands of functions of interest, there would have to be a tracepoint entered in one or more locations within each of those functions. Although it seems like this process could be automated, the automation would be complex. Different layers need to be interacted with in different ways (i.e., you cannot extract a write request's sequence number in the block layer the same way you would in the filesystem layer), and different functions at the same layer do not always have access to the same structures (which is largely dependent on parameters).

2.3 trace-cmd

trace-cmd is a command line front-end for ftrace (written by the ftrace developer Steven Rostedt) which helps overcome the awkwardness associated with ftrace interaction. The tool allows users to specify any options they wish to set for a function trace; which tracer to use, any desired filter functions, and even a specific program to trace. trace-cmd will then flip the appropriate switches within the ftrace framework appropriately, reducing the possibility of error associated with manual ftrace interactions. We immediately see major benefits to this approach. Not only do we achieve a useful granularity not provided by an in-house framework or many other non-tracepoint-based tools, but its ease of use reduces the potential of error. Additionally, trace-cmd has a feature built in by default which some aforementioned off-the-shelf tools do not have; it monitors its own effect on a trace.

2.3.1 Record

There are two main phases of trace-cmd: the record phase and the report phase. In the record phase (`trace-cmd record`), trace-cmd enables tracing on a target program with whatever configuration options it was given. After being invoked, trace-cmd will launch one tracing process per available CPU that pulls information from the kernel's ring buffer into temporary files [11]. At the conclusion of a trace, these temporary files are combined into a single .dat file and are then deleted.

2.3.2 Report

The .dat file created in trace-cmd's record phase is hardly useful by itself. The report phase (`trace-cmd report`) creates an ASCII version of the .dat file, making it human-readable, thus easier to parse. To collect function latency information, a user can either supply `trace-cmd report` the `--profile` option or they can build their own report parsing script. The former option has benefits of reducing the amount of time spent programming to create a parser, efficiently parsing and displaying latency information to the user, and being easy to use. This method suffers from one potential drawback, being that it is unclear whether or not the function latencies displayed per process are inclusive of trace-cmd's overhead.

Although more time consuming, constructing a custom parser to extract function latencies from a trace report provides a few advantages over using the `--profile` option. First, the `--profile` option only displays an event name and frequency along with its total, average, maximum, and minimum durations across all CPUs for non-trace-cmd tasks. A custom parser would instead be able to extract information that the `--profile` flag does not provide, which is inclusive of per-CPU information and latency experienced (or interference generated) by trace-cmd itself.

2.3.3 Events Lost

When performing traces at large scopes (i.e., without any filter functions), trace-cmd is prone to dropping events. This can be hazardous for traces, as the number of events lost on any given run can exceed 50% of the total events generated during the trace.

Given that I was experiencing a large number of events dropped due to the CPU generating events more quickly than trace-cmd could record, I thought of a few possible ways to mitigate the issue: reducing the total number of usable CPU cores to reduce the amount of work being done,

increasing the size of the I/O being traced, changing the scope of a trace, and slowing the CPU clock speed.

After reducing the number of cores proved to be unsuccessful (Table 2.1), I moved on to reducing the tracing scope. trace-cmd makes this process rather easy, allowing you to specify a subset of functions to be traced ("filter functions"). If I ran trace-cmd specifying that it only trace functions at the filesystem layer using the function graph tracer (e.g., `trace-cmd record -p function_graph -l ext4* <process to trace>`), I noticed that there were 0 lost events. The same held true at other layers of the filesystem, indicating that this method was superior in terms of lost events. The downside to this method is that there is a lot of context lost when tracing only certain functions; you do not get to see a complete call-graph of events which took place between any chosen filter functions.

Table 2.1: Average Events Lost Across 100 Runs

Write Configuration	8 CPUs @ 3.4GHz	1 CPU @ 3.4GHz
1KB, 1 write	38.6%	48.5%
1KB, 10 writes	47.3%	38.9%
4KB, 1 write	41.9%	48.2%
4KB, 10 write	50.4%	36.9%
1MB, 1 write	29.8%	37.9%
1MB, 10 write	47.3%	32.6%

Average events lost across 100 runs. The results of this experiment show that there is no substantial difference in events lost despite write size or number of CPU cores.

Having decided that limiting the scope of trace-cmd to specific filter functions suited my needs, I did not spend time pursuing whether a slowed clock would effect the amount of events lost for large-scope traces. For those interested, a simple way to slow down a CPU is to use the built-in CPU frequency `userspace` governor [8]. This may not be available by default on systems which use the `intel_pstate` driver, but one can simply add `intel_pstate=disable` to their kernel bootline to get around this issue.

2.3.4 Interference

For small write requests, trace interference can be quite high. The largest amount of interference my experiments experienced was after 100 runs of 100 4KB writes on a hard drive, where my workload generator was responsible for only 13% of total events logged by trace-cmd. Conversely,

trace-cmd alone comprised 46% of total events. The remaining 41% of events were attributed to other processes (such as those involved in journaling or task-scheduling). The memory management and block layers see little interference from trace-cmd for small writes, (6% and $< 1\%$, respectively), but still suffer from non-loadgen processes making up 90% of total traced events.

The data from experiments for 100 runs of 100 1MB writes shows that my workload generator's events comprise a greater portion of the total events than was the case in smaller writes at the ext4 and memory management layers, with an overall decrease at the block layer. In general, the faster the storage medium, the less interference is encountered. These results are summarized in Table 2.2 and Fig 2.3.4.

Table 2.2: Loadgen Total Event Makeup

	HDD	**SSD**	**RD**
ext4, 4KB	12%	18%	38%
ext4, 1MB	37%	37%,	39%
mm, 4KB	12%	46%	69%
mm, 1MB	73%	83%	85%
block, 4KB	10%	25%,	83%
block, 1MB	10%	10%,	37%

Percentage of events created by the workload generator. A corresponding interference graph can be seen at Fig 2.3.4.

2.3.5 Other Considerations

If you are interested only in measuring the frequency of functions and their corresponding durations, I recommend using `trace-cmd record` alongside `trace-cmd report --profile` instead of writing your own report parser in the interest of saving time programming and error checking results. In the case of traces which run for a long time or monitor particular large I/O requests, trace-cmd can create very large datafiles on disk. If you do not care for disk interaction, you can instead use `trace-cmd profile`, which combines `trace-cmd record` and `trace-cmd report --profile` without ever writing to the disk.

Log Size. The logs generated by trace-cmd can be overwhelmingly large dependent on the number and size of write requests being traced. While an experiment for tracing 100 1MB writes at the ext4 layer on a hard drive generate a 26MB .dat and 40MB report file. 100 10MB writes

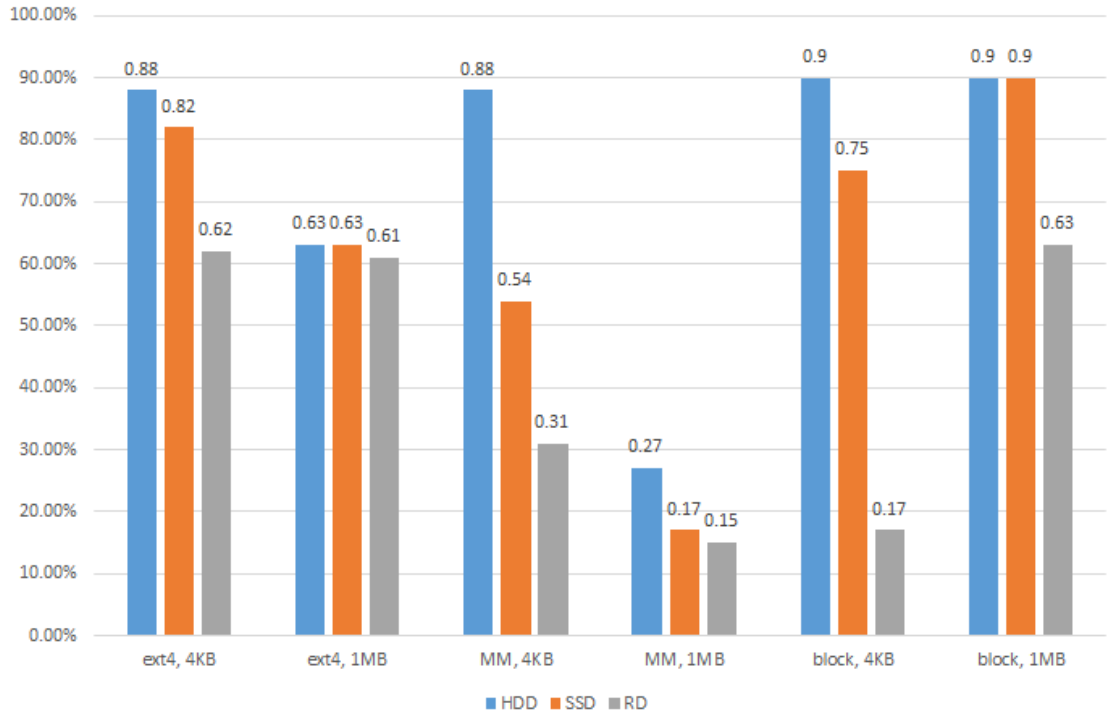

Figure 2.1: Percentage of interference encountered during traces for each device, per layer configuration.

generate a 196MB .dat file and a 362MB report file. Smart post-processing can generate a file with relevant information from this large report at a size of 1.5MB or less.

Experiment Durations. The duration of experiments for 100 runs of 100 1MB writes at the ext4 layer can exceed 15 minutes on a hard drive, which is exclusive of post-trace processing system. This amount of time is of course dependent on the speed of the device you are writing to. A 10MB write size can incur up to 4x the amount of time spent writing, giving about an hour per experiment on slower storage mediums.

2.4 perf_events

perf_events is a user space utility for monitoring workloads of applications whose subsystem was merged into the Linux kernel in version 2.6.31. The tool itself is readily available in some Linux distributions (such as Scientific Linux), and is downloadable through packages on most Linux distributions where it is not already installed. It can be used to profile disk I/O using various Linux tracing features, though it only offers a subset of the ftrace framework's functionality.

2.5 eBPF

eBPF, or "enhanced Berkeley Packet Filter" is a powerful tracing tool with a multitude of capabilities. Unlike the original BPF, eBPF is not limited to packet filtering. Instead, it can generate I/O latency heatmaps, hook into kprobes, access the entire BPF compiler collection (BCC), and more. This functionality would allow eBPF to create trace data very similar to that of trace-cmd and the underlying ftrace subsystem. By default, eBPF does not record its own overhead like trace-cmd does, and its usage is far more complex.

CHAPTER 3

EXPERIMENTS

Through a long process of refining the scope and execution of experiments, the goal of this work has become profiling the life of a write request through the Linux storage stack. Specifically, this work monitors functions called and their corresponding latencies from the time a write request has been issued from userspace to the time when it has been written to the intended storage medium. We look only into the filesystem (ext4), memory management, and block layers.

3.1 System Specs

All experiments were performed on the 4.12.9 Linux kernel. Hardware Specifications can be found in Table 3.1. The SSD used in our machine manages up to 550 MB/s seq. read, 520 MB/s seq. write, 90,000 IOPS random read, and 90,000 IOPS random write. The RAM disk (RD) used was made according to kernel specifications [9]. The HDD, SSD, and RD were each formatted with ext4.

Table 3.1: Hardware Specs

Component	Model	Model #
HDD	Seagate BarraCuda 1TB 7200RPM SATA	ST31000524AS
SSD	PNY CS1311 120GB SATA III	SSD7CS1311-120-RB
RAM	CORSAIR Vengeance 8GB SDRAM DDR3 1600	CMZ8GX3M2A1600C9B
Processor	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz	BX80623I72600K
Motherboard	ASUS P8Z68-V PRO/GEN3	-

Hardware specifications of the machine used to run experiments.

3.2 Workload Generator

In order to generate a controlled sequence of writes, I wrote a workload generator (henceforth "loadgen") in C capable of writing some given number of bytes to a target location. loadgen is only a slightly modified version of the workload generator detailed in Section 2.2.1, being updated for efficiency and given functionality to toggle whether trace-cmd should currently be tracing or not.

loadgen first ensures that trace-cmd is not currently recording trace data (see Section 3.3.1), opens/creates a file residing on a target storage medium, and then creates a buffer of some specified amount of bytes. A four-byte integer containing a sequence number (starting from 0) is written at the beginning of the buffer. Any remaining space of the buffer is filled with random integers.

Once buffer initialization has completed, loadgen begins `write()`ing to the target location. loadgen instructs trace-cmd to only record trace data from before `write()` is called until just after the subsequent `fsync()` call is made. This granularity ensures that trace-cmd is performing the minimal amount of work possible, and only records when we want it to, thus lowering the amount of trace data generated. We trace the `fsync()` call because write requests can be buffered for a non-deterministic amount of time, but `fsync()` ensures that the write requests will be written to their intended storage medium before any other instructions of a program can be executed.

3.3 Tracing

I have chosen to use the ftrace framework due to its proficiency in live tracing (Section 2.1). I do not manually interact with the framework, but instead use trace-cmd (Section 2.3) since it facilitates interaction with ftrace. trace-cmd has the added benefit of reporting its own interference with traces, if any, which allows me to quantify its overhead.

3.3.1 Trace Scope

By default, trace-cmd will start tracing a given process as soon as that process begins execution. Because of this, trace-cmd can gather trace data that a user may not intend to capture (such as function calls made during the program's initialization phase). There are some options which trace-cmd provides to start with tracing disabled, but I found they do not work as expected. The solution I opted for is to create a small interface which allows me to control whether tracing is active by `write()`ing a 0 or 1 to ftrace's on/off toggle (as described in section 2.1). Using this interface, I was able to trace whatever portions of loadgen I saw fit (namely, the `write()` and `fsync()` calls).

3.3.2 Invocation

In my use cases, trace-cmd is invoked using `trace-cmd record -p function_graph -l <function class>* <loadgen path and args>`. The function_graph tracer is chosen because it displays the duration of each function call when a funcgraph_exit point is reached, rather than timestamps for

funcgraph_entry points (as does the default function tracer). The `*` seen after `<function class>` acts as a wildcard, specifying that trace-cmd should only consider the functions which begin with `<function class>` (i.e., those beginning with ext4 or blk).

When a trace has completed, I run `trace-cmd record` without any options passed to it. Notably, `--profile` is not used because it would interfere with my own post-processing tools.

3.4 Post-Processing Tools

There are two primary tools that I use for post-processing; a report parser and a statistic calculator. The report parser looks at the output of `trace-cmd report`, separating information about function calls on a per-process basis so that it can identify which processes called which functions. Each time a function is encountered, it is added to a dictionary. If a function is found in the report which has already been encountered, we update the function's frequency count and increase its total duration by the newly encountered duration. Because the function_graph tracer lists funcgraph_entry and funcgraph_exit points, the report parser only updates the frequency when a funcgraph_entry point is seen, and only updates the total duration of a function when a funcgraph_exit is seen.

The statistic calculator looks at numerous statistics from each run of an experiment, keeping track of totals and averages of these statistics across all runs within an experiment. These statistics include the number of events generated during a trace (keeping track of events per process of interest, such as loadgen and trace-cmd, as well as a total number of events from all processes), the total number of events lost during a trace, the number of runs that are currently being totaled and averaged, and some useful percentages from the information listed above.

All data was stored in .csv format and imported into Microsoft Excel to generate graphs.

3.5 Experiment Configuration

Individual experiments consist of 100 runs (101 total, with the first run expelled), where each run traces 100 `write()`s of some specified amount of bytes at a given layer. After each individual run, post-processing tools are used to collect and update relevant statistics. The experiment configurations used for this work are detailed in Table 3.2. Each experiment was run after a reboot of the machine to help ensure a more uniform initial state.

Table 3.2: Traced Layers

Layer	**4KB**	**8KB**	**1MB**
Application			
VFS			
FS	X	X	X
MM	X	X	X
block	X	X	X

Which layers above SCSI were traced, and with what size writes. All listed write sizes are aligned (i.e., 4KB = 4096B).

At the filesystem layer, all functions beginning with `ext4_` were traced. This provides complete coverage of all FS-related functions called on device formatted with ext4. Choosing functions for the MM and block layers was only slightly more difficult, as not all of the potentially interesting functions begin with the same extension. Furthermore, tracing every possible function at these layers can result in potentially too many events being generated for trace-cmd to record, thus contributing to an overall loss of data (for example, trying to capture each time a `page_` function is called can result in a 20% or greater loss of events). As such, this work focuses on functions called that are near the top of the MM and block layers (in terms of call order), those beginning with `generic_` and `blk_`, respectively. Because the functions are near the top of their respective layers, their timing encompasses many functions which live underneath them.

CHAPTER 4

RESULTS

Results from our various experiments (described in Table 3.2) suggest that the vast majority of time spent during a write is within the ext4 layer despite which device was written to and the size of writes made. Due to similarities between the 4KB and 8KB experiments (and therefore a lack of interesting differences between the two), this section will focus on the results seen in experiments using 4KB and 1MB write sizes.

First, we look at the total amount of time spent per layer for a 4KB and 1MB write to get an idea for which layers need to be examined in depth for different storage devices. Table 4.1 shows that a write spends far more of its time traversing through ext4 than it does the memory management or block layers no matter which storage device is used. Unsurprisingly, the memory management layer incurs a greater cost for the RAM disk (a memory dependent medium) than it does for any other storage device. A sample visual representation of this data is seen in Fig 4.

Table 4.1: Percentage of Time Spent for 4KB and 1MB writes

Device	**ext4, 4KB**	**MM, 4KB**	**block, 4KB**	**ext4, 1MB**	**MM, 1MB**	**block, 1MB**
HDD	99%	$< 1\%$	$< 1\%$	99%	$< 1\%$	$< 1\%$
SSD	95%	1%	3%	96%	$\sim$4%	$< 1\%$
RD	88%	$\sim$12%	$< 1\%$	81%	$\sim$19%	$< 1\%$

Percentage of time spent per layer for each device on a 4KB and 1MB write, averaged across 10,000 writes of each size.

4.1 Functions of Interest

This section lists the most time consuming functions from each layer. Due to the recent completion of experiments at the memory management and block layers, their results have yet to be thoroughly examined. However, a brief inspection notes that the data between the 4KB and 1MB experiments across devices is mostly similar. Because the memory management and block

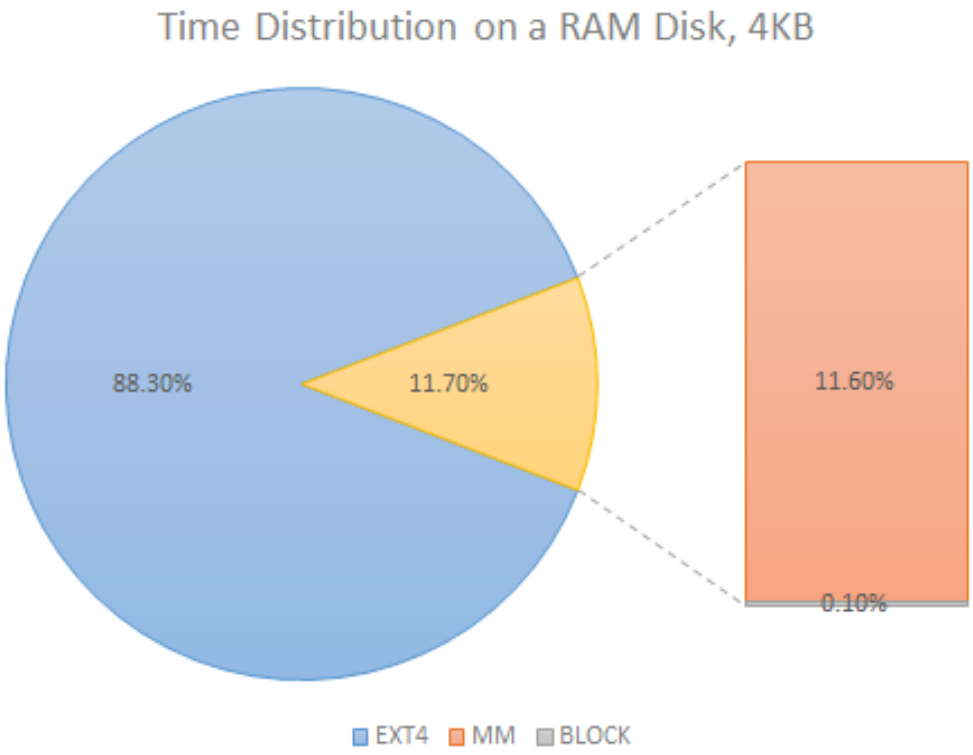

Figure 4.1: Percentage of time spent per layer on a RAM disk averaged across 10,000 4KB writes.

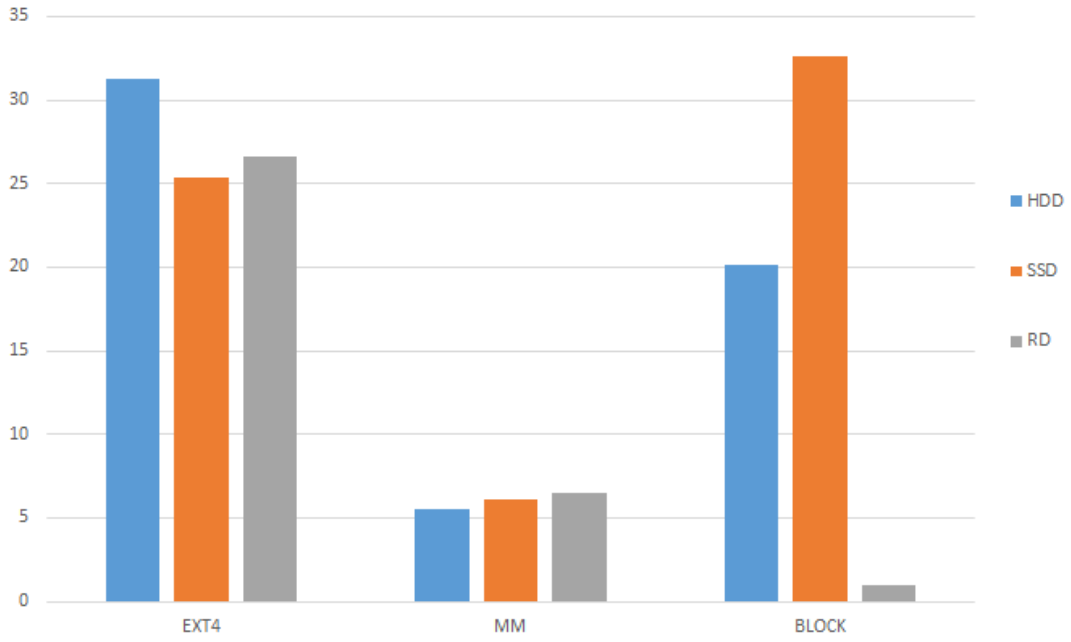

Figure 4.2: Function call frequency across the ext4, memory management, and block layers for a 4KB write.

layers have a limited set of functions being traced, data for the top 10 functions of interest at those layers have been listed in full (Sections 4.1.2 and 4.1.3).

4.1.1 ext4

ext4_sync_file(). When dealing with 4KB writes, `ext4_sync_file()` overwhelmingly dominates the lifespan of a write request for HDDs and SSDs, as shown in Table 4.2. Although Table 4.2 shows the average durations of functions for each device, `ext4_sync_file()` also consumed the most *total* time for a write request's lifespan. For 1MB writes, `ext4_sync_file()` is still a top-3 performer across all three storage mediums (Table 4.3).

`ext4_sync_file()` being called is a direct result of loadgen's `fsync()` call. In its implementation, a commit is initiated and the process waits until it completes. Because we know that writing to a storage medium can take a non-deterministic (but long) amount of time, we expect this function to be a top contender.

ext4_file_write_iter(). If there were any function that we would expect to see when performing `write()`s on ext4, it would be `ext4_file_write_iter()` – the function in charge of getting a write request ready for lower layers. Surely enough, it is a top-3 performer for 4KB and 1MB writes across all devices. What is interesting about this function is that we see it call two other top 10 performing functions in terms of total duration, `ext4_mark_inode_dirty()` and `ext4_mark_iloc_dirty()`. `ext4_mark_inode_dirty()` only has one potentially time-consuming function call, `smp_rmb()`, which is a barrier incurred on machines with symmetric multi-processing (SMP) enabled. `ext4_mark_iloc_dirty()` on the other hand suffers from lock contention and an ext4 checksum which calls an expensive cryptohash.

In an effort to reduce the relative amount of time spent in `ext4_file_write_iter()`, I disabled symmetric multi-processing on my machine. The hope was that the `smp_rmb()` barrier would not incur a penalty, as it relies on SMP to be enabled. This attempt did not pan out, however, as `ext4_file_write_iter()` still appeared in the top 3 performing functions across all devices with no discernible decrease in its relative total time.

ext4_da_write_begin(). As mentioned in Section 1.1.1, ext4 implements delayed block allocation, which all `ext4_da_` functions are part of. Within this function's code, we see three potential sources of time consumption. The first is a section of code responsible for calling

Table 4.2: Average Durations by Function, 4KB Write, ext4

Function	HDD	SSD	RD
ext4_sync_file()	38,687	1,429	7
ext4_file_write_iter()	835	53	4
ext4_writepages()	18	21	5
ext4_dirty_inode()	6	9	3
ext4_da_write_begin()	5	4	3

Average duration (us) of the HDD's top five longest average duration functions from an experiment of 10,000 4KB writes through ext4, rounded to the nearest whole number. Although these functions were chosen based off of the HDD results, the first three listed functions are the top three across all devices and the remaining two functions are within the top ten across all devices.

Table 4.3: Total Durations by Function, 1MB Write, ext4

Function	HDD	SSD	RD
ext4_file_write_iter()	381,684,136	50,234,280	3,902,591
ext4_sync_file()	249,996,165	35,215,351	1,041,199
ext4_da_write_begin()	3,045,175	2,797,478	1,840,435

Total duration (us) of the HDD, SSD, and RD's top three functions from an experiment of 10,000 1MB writes through ext4, rounded to the nearest whole number.

`grab_cache_page_write_begin()` (potentially in a loop), with a note that it could take "a long time" if there is a lot of memory pressure or if the page is being written back. Another area of interest is the `retry_journal` label, which continuously attempts to start journaling some updates or loops us back up to calling `grab_cache_page_write_begin()`.

The final area of interest is perhaps the most interesting section of `ext4_da_write_begin()`, as it is a call to `ext4_block_write_begin()` surrounded by `#ifdef CONFIG_EXT4_FS_ENCRYPTION`. Without `CONFIG_EXT4_FS_ENCRYPTION` enabled, a more generic block write function is called. As the first two time consuming sections require strong knowledge of the memory management subsystem, I instead looked into the effects of `CONFIG_EXT4_FS_ENCRYPTION`. If it were disabled, quite a few delayed allocation functions would potentially have their durations reduced. However, this compilation flag is dependent on many other (mostly encryption related) filesystem flags, and is probably best left alone unless a user wants to completely disable delayed allocation.

4.1.2 mm

The top 10 functions traced on a HDD for 4KB and 1MB writes at the memory management layer have been listed here with their corresponding timing information on other storage mediums for reference. One comment worth noting is that `generic_perform_write()` is called directly by `ext4_file_write_iter()`, a noteworthy function from Section 4.1.1.

Table 4.4: Average Durations by Function, Memory Management

	4KB			1MB		
Function	**HDD**	**SSD**	**RD**	**HDD**	**SSD**	**RD**
`generic_update_time()`	5	4	5	4	5	5
`generic_make_request()`	3	4	1	6	5	72
`generic_perform_write()`	3	5	1	316	287	189
`generic_write_end()`	1	1	0	0	0	0
`generic_file_read_iter()`	1	1	1	1	1	1
`generic_make_request_checks()`	0	0	0	0	0	0
`generic_exec_single()`	0	0	0	0	0	0
`generic_fillattr()`	0	0	0	0	0	0
`generic_permission()`	0	0	0	0	0	0
`generic_file_open()`	0	0	0	0	0	0

Average duration (us) of functions for 10,000 4KB and 1MB writes through the memory management layer, rounded to the nearest whole number. 0's refer to values between 0us and 1us.

4.1.3 block

The top 10 functions traced on a HDD for 4KB and 1MB writes at the block layer have been listed here with their corresponding timing information on other storage mediums for reference.

Table 4.5: Average Durations by Function, Block

	4KB			1MB		
Function	**HDD**	**SSD**	**RD**	**HDD**	**SSD**	**RD**
`blk_finish_plug()`	13	11	0	13	12	0
`blk_flush_plug_list()`	13	11	0	12	12	0
`blk_queue_bio()`	5	7	-	7	7	-
`blk_peek_request()`	2	1	-	2	2	-
`blk_queue_start_tag()`	2	2	-	1	1	-
`blk_start_request()`	1	1	-	1	1	-
`blk_init_request_from_bio()`	1	1	-	0	0	-
`blk_rq_map_sg()`	0	-	-	3	2	-
`blk_account_io_start()`	0	-	-	0	0	-
`blk_queue_split()`	0	0	-	3	3	-

Average duration (us) of functions for 10,000 4KB and 1MB writes through the block layer, rounded to the nearest whole number. 0's refer to values that are between 0 and 1us. -'s indicate that the function was not hit for a specified device.

CHAPTER 5

CONCLUSION AND FUTURE WORK

I have detailed an easily reproducible and refined method for measuring I/O request latency at different layers of the storage stack. Although the sample results of this work indicate that optimization efforts need to be directed toward ext4, readers should keep in mind that the results given are for write-heavy workloads and consider only three components of the Linux storage stack.

The remainder of this chapter will describe ways to expand the project to measure different kinds of I/O workloads across more components of the Linux storage stack. Additionally, this chapter will mention some potentially interesting questions worth exploration that are directly related to this research.

5.1 Expanding to Reads

Modifying our methodology to include reads is simple and was only excluded in our experimental results due to time constraints. The only component of our research methodology to change is the workload generator; specifically, `write()` calls would instead change to `read()` calls which read data from a file located on a chosen storage medium. Both sequential and random reads could be used, the latter providing a way to prevent prefetching from impacting results.

5.2 Increasing Scope

With only three components of the storage stack being measured, we cannot claim to have a complete picture of an I/O request's lifespan. For certain types of workloads, SCSI drivers can potentially make up a higher percentage of an I/O request's lifespan than the filesystem layer [13]. Thus, looking into the SCSI-related layers could yield more bottlenecks within the storage stack. Ideally, this same process would be replicated at each layer of the storage stack so that we can definitely state which layers are problematic, and why.

Not only do more layers of the storage stack need to be considered, but some components might benefit from being looked at more in depth. As stated in Section 1.1.2, the memory management

subsystem has a multitude of purposes, yet this work only considers one; the `generic_` interface. There are potentially other aspects of this subsystem which would be of interest to researchers, though some should be considered more volatile than others. For example, trying to measure all functions related to page allocation and management could result in a tremendous amount of trace-events lost (see Section 2.3.3).

As has been previously mentioned, some filesystems employ journaling. This work considered one such filesystem, ext4, without thoroughly analyzing what is going on in the journal. Luckily, the methodology from this work can be easily applied to measure the journaling activities (or, the "journal layer"), as ext4 has a journaling process which can be tracked by trace-cmd (and is, by default).

It may also be worthwhile to look at the filesystem layer across multiple filesystems, not just ext4. After all, it is likely that different filesystems will impact an I/O request's lifespan in different ways; perhaps a filesystem designed with SSDs in mind (such as f2fs) will significantly lower the amount of time a request stays within the filesystem layer.

5.3 Final Thoughts

Tracing frameworks within the kernel seem to suffer from the problem of being unable to exclude the timings measured at lower layers of the Linux storage stack from those found higher within the storage stack. For example, if `ext4_file_write_iter()` is traced, we know that `generic_file_write_iter()` will be called as a result. The first function lives at the filesystem layer, and the second lives within the memory management subsystem. From what I have seen, tracing frameworks will add the time spent within callees to their corresponding callers. In our example, this means that `generic_file_write_iter()` will report latency for itself which will also be added to the latency of `ext4_file_write_iter()`, effectively being recorded twice.

If there is a problem of caller durations being inclusive of callee durations, it is worth noting that the sample results given in Section 4.1.1 would still retain their implications. Specifically, if the memory management subsystem and block layer are both having their latencies recorded twice, the filesystem layer would still account for the majority of an I/O's lifespan (accounting for 85% or more of the total I/O duration).

BIBLIOGRAPHY

[1] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpachi-Dusseau. The Unwritten Contract of Solid State Drives, 2017.

[2] Steven Swanson, Adrian M. Caulfield. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage, 2013.

[3] Jonathan Corbet. debugfs kernel documentation, 2009. https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt.

[4] Mathieu Desnoyerst. Using the Linux Kernel Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.

[5] Jim Keniston, Prasanna S Panchamukhi, Masami Hiramatsu. Kernel probes (kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt.

[6] Masami Hiramatsu. [PATCH] ftrace/x86: Fix function graph tracer reset path. https://lkml.org/lkml/2016/5/16/493.

[7] Greg Kroah-Hartman. Driving Me Nuts - Things You Never Should Do in the Kernel, 2005.

[8] Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar. Linux CPUFreq, CPUFreq Governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[9] James Nelson. Using the ram disk block device with linux, 2004.

[10] Steven Rostedt. ftrace - Function Tracer kernel documentation, 2008. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[11] Steven Rostedt. TRACE-CMD-RECORD(1), 2010. http://man7.org/linux/man-pages/man1/trace-cmd-record.1.html.

[12] Jun Yan, Cejo K. Lonappan, Amir Vajid, Digvijay Sing, and William J. Kaiser. Accurate and Low-Overhead Process-level Energy Estimation for Modern Hard Disk Drives, 2013.

[13] Kristen Carlson Accardi, Matthew Wilcox. Linux Storage Stack Performance, 2008.

BIOGRAPHICAL SKETCH

EDUCATION

Master of Science, Computer Science 2015-2017

Florida State University

After a year of coursework, I was selected by my advising professor (Dr. Andy Wang) to partake in the IRES grant, which sent me to Chalmer's University in Gothenburg, Sweden. During my stay at Chalmer's, I worked on my thesis under the direction of Dr. Wang and Chalmer's faculty. Upon returning to FSU, I revised my thesis a bit and continued researching.

Bachelor of Science, Computer Science 2011-2015

Florida State University

During my undergraduate coursework at FSU, I was fortunate enough to accept a research position under Dr. Andy Wang, assisting Dr. Mark Stanovich (a doctoral student at the time) in his research. Thanks to the pleasant experienced I had working with Dr Stanovich under Dr Wang, I decided to pursue graduate studies at FSU.

Associate of Arts 2009-2011

Tallahassee Community College

A couple of months after I completed high school, I enrolled at Tallahassee Community College. My very first programming classes was taken here, which is what spurred me towards Florida State University's Computer Science program.

WORK EXPERIENCE

Teaching Assistant 2015-2017

Florida State University, part-time

This work included holding recitations (lectures), grading assignments, engaging with students during office hours, and occasionally mentoring students. I have been a TA for Object Oriented Programming (C++), Operating Systems, Unix Tools, and MicroApps.

Systems Group Member 2014-2015

Florida State University, Part-time

As part of the Computer Science System Administration group, I had several achievements; I automated day-to-day tasks, handled help desk tickets, and migrated the CS department's mail server to the maildir format.