Journal of Systems Architecture 164 (2025) 103405

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

μScope: Evaluating storage stack robustness against SSD's latency variation

Linxiao Bai [a,*], Shanshan Li [a], Zhouyang Jia [a], Yu Jiang [b], Yuanliang Zhang [a], Zichen Xu [c], Bin Lin [d], Si Zheng [a], Xiangke Liao [a]

[a] National University of Defence Technology, Changsha, China
[b] TsingHua University, Beijing, China
[c] School of Mathematics and Computer Science, Nanchang University, Jiangxi, China
[d] Center for Strategic Evaluation Consultation, Academy of Military China, Beijing, China

ARTICLE INFO

Dataset link: https://github.com/u-Scope/uScope

Keywords:
Storage stack performance
Latency variation
Fault injection
Performance monitoring

ABSTRACT

The rapid development of Solid State Disks (SSDs) drastically reduces device latency from 100 µs to around 10 µs. However, performance advertised is not always performance delivered. Background operations (e.g., garbage collection and wear leveling) inside the SSDs now may severely influence the performance. In addition, SSDs are also susceptible to fail-slow failures. Traditionally, studying SSD-based stack focuses on understanding the SSD internal behaviors or discussing the impacts of software stack on throughput.

In this paper, we conduct an extensive study on software stack atop the low-latency SSDs, especially under device latency variations. We build μScope to overcome two major challenges, including achieving fine-grained latency injection and low-overhead monitoring, in profiling. Via μScope, we manage to obtain three major lessons in access patterns, consistency trade-offs and consecutive performance variations which shall benefit developers for further optimizations.

1. Introduction

Flash-based Solid State Disk (SSD) is a staple in today's computing on multiple fronts including personal mobile devices to cloud-scale data centers. The competitive performance (e.g., 6 GiB/s throughput with PCIe 4.0 support), the low energy draw (e.g., 10 times less than disks [1]) and the high storage density (e.g., 100 GiB QLC-based SSD) make the SSD a favorable choice.

Nevertheless, to fully exploit the power of SSDs is non-trivial. Previous work [2–4] have shown that the performance, mainly throughput, of a SSD-based storage stack can be greatly influenced by the SSD design, such as internal parallelism and NAND properties. In addition, recent research [5] also focus on studying the throughput variation of SSD-based stack under different file system configurations. These studies have helped developers to further understand the nature of SSDs and motivated subsequent optimizations in development and tuning.

Still, one significant topic has been left untouched, the software stack behaviors (e.g., file system, kernel block layer and driver) under SSD's latency variations. The importance of such a study is two-fold. First, SSD are becoming increasingly faster. For example, the newly emerged Ultra-low Latency (ULL) SSDs have driven the average latency to the sub-10 µs level [6,7]. Such improvement fundamentally changes the landscape as we can no longer treat such SSDs as slow storage devices behind the *IO Wall*. Instead, recent studies [8,9] indicate that software stack processing now take nearly same amount of time as the device during the IO processing. This motivates us to investigate the influence of software stack behaviors on the latency which may be regarded as noises when running on slower devices.

Second, SSD itself can also cause widespread and severe latency variation. It is well-known that SSD's internal activities (e.g., garbage collection, wear leveling and data integrity checks) can severely impact IO processing, leading to high tail latency [10–15]. Moreover, SSDs are also susceptible to a new type of failure, called Fail-slow failure [16–18], where the storage device is still functioning but with much degraded performance. In this case, studying the software handling and reaction can be helpful for stack's end-to-end robustness.

Unfortunately, evaluating the software stack behavior is not straightforward. First, existing analytical framework and methodology (e.g., ltrace [19] and strace [20]) focus on the end-to-end performance, thereby cannot provide an in-depth and fine-grained analysis on each layer of the software stack. In addition, if we simply use instrumentation-based solutions to capture latency variations, it is still

* Corresponding author.
E-mail addresses: linxiao_b@nudt.edu.cn (L. Bai), shanshanli@nudt.edu.cn (S. Li), jiazhouyang@nudt.edu.cn (Z. Jia), jy1989@mail.tsinghua.edu.cn (Y. Jiang), zhangyuanliang13@nudt.edu.cn (Y. Zhang), xuz@ncu.edu.cn (Z. Xu), bingo186@126.com (B. Lin), si.zheng1009@gmail.com (S. Zheng), xkliao@nudt.edu.cn (X. Liao).

https://doi.org/10.1016/j.sysarc.2025.103405
Received 23 December 2024; Received in revised form 18 February 2025; Accepted 28 March 2025
Available online 24 April 2025

inaccurate as monitoring itself can introduce extra latency. Moreover, even if we manage to obtain a low-overhead monitoring method, the SSD can be unstable (e.g., background activities or suffering fail-slow failures) and thus lead to incorrect conclusions.

In this paper, to study the impact of SSD's latency variation in storage stack, we build μScope, an eBPF-based [21,22] fault injection tool. There are two major components. First, we, based on RamDisk [23], build a simulated SSD, to ensure the target device can run with controlled delays in a deterministic fashion. We have included frequency, amplitude and distribution patterns of fault injection to simulate a wide range of SSD behaviors.

Moreover, we devise an eBPF-based monitoring framework to achieve low-overhead and fine-grained latency monitoring across all layers in software stack. Specifically, it performs real-time monitoring of the storage stack related functions in the kernel at the nanosecond level. We have set up interfaces for users, which can easily perform deeper performance monitoring on specified functions.

With μScope, we evaluate three typical benchmarks in Filebench [24] extensively under a wide variety of setups, including file systems and configurations. Through analysis, we identify the following major observations:

- Write-intensive workloads exhibit greater sensitivity to latency variation compared to read-intensive ones. In the presence of latency variation, the slowdown experienced by write-intensive tasks (9.39%) significantly exceeds that of read-intensive tasks (3.34%). This is because write operations involve more disk access than read operations, increasing the chances of latency variation events. Furthermore, the latency of software stack increase in write operations is 11.3% to 29.6% higher compared to read operations under different configuration conditions. Employing a thinner storage stack, like using storage performance development kit(SPDK), can effectively mitigate the impact. The ratio of slowdown decrease from 11.08% to 3.06% after equipped with SPDK.
- Configuration items related to consistency and performance decide the dependence between software executions and underlying I/O. Thus, the dependence can increase the impact of SSD's latency variation(i.e. SSD's latency variation) on software stack. Moreover, different configuration item values may lead to an impact gap of more than 10%. Unlocking read/write order relationships in the kernel helps improve storage stack performance. However, only software-level modifications have not fundamentally avoided the impact of SSD's latency variation. Performance degradation still occurs during the processing the orders.
- Various latency variation patterns result in different impacts on the software stack. As the block layer queue temporarily accumulates SSD's latency, continuous slow I/Os negatively impact the software stack's performance considerably. Furthermore, in situations with the same occurrence ratio, the software stack delay caused by continuous slow I/Os increases by 5% to 10% more than that caused by random slow I/Os. Focusing on the prediction of continuous slow I/Os should become the key of solving the problem. Compared to the node switching commonly used in current distributed systems, the disk switching delay is lower, making it a better solution to mitigate the impacts.

Our paper has three key contributions:

- To the best of our knowledge, we are the first to provide a detailed impact analysis of SSD's latency variation on software stack. We believe our study paves the way towards the better understanding on impact of SSD's latency variation.
- We build an eBPF-based fault injection tool–μScope. It is used to analyze storage stack robustness against SSD's latency variation. We publish μScope on Github: https://github.com/u-Scope/uScope.
- We use μScope to conduct a comprehensive study on the impact of SSD's latency variation on software stack. Also, we discuss the relationship between the impact and workload, file system, and latency variation patterns. After analysis of results, we offer insights into the root causes of SSD's latency variation's impact on software stack and provide suggestions for users.

The rest of the paper is organized as follows: Sections 2 and 3 introduce the background and motivation of our research. Section 4 introduces μScope we develop. Section 5 examines the experimental settings. Then, Section 6 evaluates device latency variation effect, analyzes its root causes and provides suggestions. Next, Section 7 discusses the experimental limitations, while Section 8 explores related work. Finally, we conclude our work in Section 9.

2. Background

2.1. Storage stack

With the iteration of Linux versions, the complexity of storage stacks has also increased. The current Linux storage stack can be roughly divided into the following parts: Virtual File System(VFS), page cache, File System(FS), Block Layer, Drivers, and Device [25,26].

The main function of VFS [27] unifies the interfaces of various file systems. User mode read and write functions include read, write, readv, writev, pread and pwrite [28]. Various file systems have their own implementations, and the role of VFS is to unify them into a single function name and provide it to users.

In contrast, the page cache [29] mechanism is used to improve performance. When memory resources are not sufficient, the data accessed by users will not be discarded, but will be cached in memory. The next time the user utilizes the device, rapid in-memory data can be accessed without the need for slow storage devices. When the page cache is hit, there will be no I/O access to the disk.

Similarly, FS is the file system that manages persistent data. Its primary function is to achieve unified disk space management. On the one hand, the FS plans disk space uniformly, and on the other hand, the FS provides a user-friendly interface for ordinary users.

Additionally, the block layer [30] connects the FS and device driver layers. I/O is the basic bio unit in the block layer. The block layer is responsible for staging, merging, and determining the order in which I/O requests are processed. Hence, the block layer's soft interruption feature handles the work after an I/O request is completed.

Finally, drivers are special programs that enable communication between computers and devices, which can be equivalent to hardware interfaces.

2.2. Kernel performance monitoring

Kernel performance monitoring technology has always garnered attention. Consequently, different kinds of performance monitoring tools for Linux exist. The commonly used existing kernel monitoring tools include strace [20], ltrace [19], Ftrace [31], iostat [32] and perf [33].

First, ltrace tracks a process's library function calls. It lists which library functions were called during the process and the time of the call. Similarly, strace monitors the time consumption of system calls during process execution. It is mainly used to monitor the interaction between user space processes and the kernel.

In contrast, Ftrace is an internal tracer designed to enable system developers and designers to monitor the occurrences within the kernel. It can be used for debugging or analyzing latencies and performance issues that occur outside the user-space. One of the most common uses of Ftrace is event tracing. Numerous static event points exist throughout the kernel that can be enabled via the tracefs file system to monitor the occurrences within certain parts of the kernel [31].

Additionally, iostat is a statistical tool for the I/O system. Iostat monitors disk I/O statistics for all disks and file systems and it outputs values such as CPU and disk utilization, disk read and write speed, I/O request queue length, and waiting time length in the current state.

Finally, perf is a profiling tool built into the Linux kernel source tree. It is based on performance events and supports analysis of processor and operating system related indicators. Perf can be used for finding performance bottlenecks and locating hot codes.

2.3. Fail-slow devices

Fail-slow device is used to describe a hardware that is still running and functional but in a degraded mode [16], slower than its expected performance. The Fail-slow failure is a hardware problem of great concern, especially in SSD. Current research points out that the probability of Fail-slow in SSD is 6.05× higher than that in HDD [17].

There are various reasons for SSD Fail-slow. On one hand, there are internal root causes, such as firmware bugs, heavy garbage collection, and suboptimal wear-leveling. On the other hand, hot temperature, power and so on can be attributed to external causes. The detailed reasons are listed in the previous work [16].

The appearance of tail latency is mostly caused by the Fail-slow in the SSD. In the test of a real SSD, the latency will reach 2.5× of the average latency at the 99% percentile, even 20× at the 99.9% percentile [11]. Such high tail latency is unacceptable to users. Besides, Fail-slow is likely to cause cascading reaction. Existing works offers anecdotes about the serious results caused by Fail-slow, which motivates us to carry out this work. There are practical examples to prove that a Fail-slow drive can significantly degrade the performance of the entire RAID based on SSDs [10]. Some work has also proposed the possibility of Fail-slow to fail-stop, which can bring unpredictable consequences such as system collapse [16]. As a kind of severe latency variation within hardware, Fail-slow may seriously affect the performance of the software or damage the normal operation of the software.

3. Motivation

Obtaining a stable storage device. Prior work [16,17] points out that compared to hard disk drive (HDD), latency variation in NVMe SSD is much more widespread and frequent, and can significantly degrade performance. These problems motivate us to ask a question: Are the software stacks resilient to device-level severe latency variations (i.e., Fail-slow failures)?

We attempt to directly study the software stack under SSD performance. We encounter difficulties in monitoring the real SSD. First, SSD is a black box. By nature, its latency is beyond our control due to its internal activities (e.g., garbage collection). Note that, host-aware SSDs (e.g., open-channel SSD) enables users with better management but can suffer from excessive monitoring overhead, thereby yielding untrustworthy results.

Therefore, we choose to use simulated SSDs as the testing carrier and conduct fault injection to simulate performance fluctuations in SSDs. Initially, we attempt to use existing SSD simulation tools such as SSDSim [34], SimpleSSD [35] and FEMU [36].

We use Fio [37] to compare and test real and simulated disks. To test the latency and bandwidth of random reads and writes on each simulated disk, we conduct 10 repeated experiments on each device and calculate the average value of each test item. The result is as shown in the Table 1. The term "CPU%+" represents the incremental increase in CPU utilization after initiating the simulation, compared to its utilization prior to the simulation's start.

These tools all simulate the functions of SSDs very well. However, the implementation of SSDSim is in the user mode. The access path to it is different from the actual disk access. It cannot help study the impact of SSD performance on the software stack. In contrast, SimpleSSD focuses on implementing the internal logical structure of SSDs, thus introducing a delay of 100 μs during operation, which exceeds the actual disk latency significantly. So SimpleSSD is not the optimal choice for simulating SSDs either. FEMU is a simulation tool capable of emulating various operations in SSDs, such as garbage collection and wear leveling. It also offers latency and bandwidth characteristics comparable to those of real SSDs. However, due to its significant additional CPU overhead, it cannot be considered our preferred simulation tool.

Table 1
Performance results of real and simulated disks.

Device	Fio results			CPU%+
	lat (μs)	99.9th lat	Bandwidth	
Samsung 980PRO	37.55	112.96	6370 MB/s	–
SSDsim	45.59	89.07	6099 MB/s	+5.2%
SimpleSSD	60.90	189.43	5976 MB/s	+9.3%
FEMU	40.02	135.88	6023 MB/s	+9.2%
μScope-RamDisk	34.50	93.01	6219 MB/s	+5.1%

Table 2
Performance results of different profiling tools.

Tools	Precision	Object	Latency (μs)	
			AVG	+
baseline	–	–	45.91	0
strace	μs	System calls	–	–
ltrace	μs	Library function calls	–	–
iostat	–	CPU and disks	–	–
Ftrace	μs	Storage stack functions	48.89	6.49%
perf	ns	Storage stack functions	51.73	12.7%
μScope	ns	Storage stack functions	49.03	6.79%

Low-overhead profiling. We have also made several preliminary attempts in performance monitoring. There exist a rich set of performance monitoring tools for the kernel (introduced in 2.2).

To compare different performance monitoring tools, we use Filebench [24] to test the commonly used tools in the current kernel. The baseline is the performance result of Filebench without using any tools. We use the default parameters of Filebench for testing and give the results in Table 2. More testing details are explained in Section 6.6.

Strace and ltrace can be used to monitor system calls and library function calls during software operation. However, as user monitoring tools, they can only provide end-to-end performance and their performance monitoring in the kernel storage stack is not sufficient. The main objects monitored by iostat are the disk and CPU, which lacks fine-grained I/O process monitoring. Although Ftrace can track and observe functions in the kernel, its accuracy is subtle and insufficient to support our performance analysis of low latency software stacks. Specifically, Ftrace provides performance measurement accuracy at the microsecond level. However, the latency of some functions within the storage stack has reached the 100-nanosecond level. Therefore, while Ftrace is useful for observing general trends, it is not sufficient for a fine-grained analysis of the software stack's response to SSD's latency variation. Perf is a profiling tool based on eBPF which we expect to use. It meets all our requirements for monitoring functionality. Unfortunately, it introduces too much overhead because of recording redundant information. This is against our purpose of achieving microsecond-level performance variations.

4. μScope design

In this section, first, we present the design goals and introduce the workflow of μScope (Section 4.1), and then describe the detailed technology including SSD simulation (Section 4.2), slow I/O injection (Section 4.3), and the eBPF monitoring system (Section 4.4).

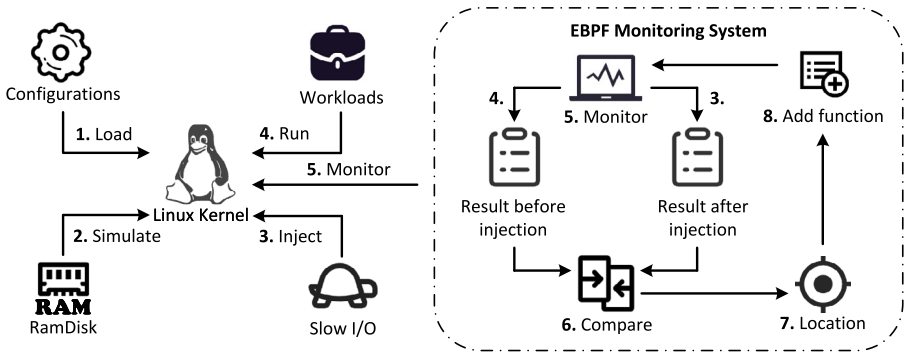


Fig. 1. μScope workflow: evaluate storage stack robustness against SSD's latency variation. There are eight steps: 1. Load the experimental environment. 2. Simulate devices using a RamDisk. 3. Inject slow I/Os as SSD's latency variation. 4. Choose and run workloads on this disk. 5. Monitor kernel function latency. 6. Compare the results to find performance anomalies. 7. Analyze and locate the function with abnormal performance. 8. Add the monitored function to find the root cause.

4.1. Overview

In this study, we aim to analyze the impact of SSD's latency variation on storage stack. To achieve this, we have three goals.

Goal I: Obtaining a stable device: Previous studies [16] have shown that latency variation problems exists on a large scale in SSDs. However, using real devices may not meet the testing requirements. In real devices, latency variation is unstable and unmanageable, making it difficult for reproduction. Therefore, the optimal choice is simulating a stable device.

Goal II: Injecting realistic and controllable slow I/Os: We inject slow I/Os into the test device to simulate the occurrence of latency variation in real devices. To make the simulation as real as possible, multiple indicators such as time, frequency, and latency variation pattern are evaluated. Therefore, it is crucial for the root cause analysis to generate a controllable and reproducible latency variation.

Goal III: Monitoring kernel storage stack with light overhead: Monitoring the performance of the storage stack during testing is essential. There are currently many tools for kernel monitoring, but not under low latency conditions. Also, existing tools have low accuracy or heavy load (as introduced in Section 3). Therefore, it is necessary to build a high-precision kernel storage stack monitoring with light overhead.

In order to achieve these goals, we devise μScope. The μScope workflow can be divided into two parts and eight steps, which are displayed in Fig. 1 as follows: The first part is the preparation work before testing, including steps 1–3: **1.** Load the experimental environment (file systems) according to the selected parameters. **2.** Simulate devices using a RamDisk. **3.** Inject slow I/Os as SSD's latency variation. The second part is testing and performance monitoring, including steps 4–8: **4.** Choose and run the workloads on this disk. **5.** Monitor kernel function latency. **6.** Compare the results before and after injection to find performance anomalies (i.e., performance anomalies refer to changes other than an increase in the underlying latency. For example, the latency in the ext4 file system layer may increase by 13% after the injection delay). **7.** Analyze and locate the function with the abnormal performance. **8.** Add the monitored function.

After the new monitored function is added, the system will monitor the kernel function delay in a more fine-grained manner. Similarly, reducing the number of monitored functions is also allowed. Repeat steps **5.** to **8.** in the monitoring system until the root cause of the abnormal performance is revealed. Moreover, adding monitored functions step by step minimizes the performance noise caused by monitoring.

4.2. PART I : SSD simulation

According to our previous research (in Table 1), the vast majority of simulators are unable to perfectly simulate the operation of real disk SSDs. Existing simulators do not meet our demands as one important goal of theirs is to mimic the internal behaviors of SSDs including garbage collection, wear leveling and data scrubbing. We, however, intend to explore the behaviors of the software stack under SSDs' latency variations. While such latency variations can be caused by the disk internal behaviors, they are uncontrollable. Thus, we chose to build a stable simulated SSD and apply artificial latency upon it to study the software behaviors under various types of latency. So, we expect to obtain a stable storage device, which we can treat as a pure SSD that does not involve any interfere from garbage collection, wear leveling and such intrinsic mechanisms. The problems exposed on such devices must also exist in real disks, and maybe more complex. Meanwhile, more attention should be paid to observing the impact of software stacks instead of the internal implementation of device.

Therefore, we use RamDisk for simulation. We generate a file to simulate the device in memory, and then format and mount it like normal disks. In this way, read and write operations on the simulated device can pass through the same storage stack like real disks.

To achieve a similar effect to the real disk in our simulation disk, we used the Fio [37] tool to compare and test the simulation disk with the SAMSUNG 980PRO 250G. We applied different workload types in Fio, and then performed 1000 runs on the simulated and real disks to compare the output results. The result showed that the simulation disk runs are faster than those of the actual one under the default parameters. This is reasonable because the memory runs faster than the SSD disks.

To make our simulation more realistic, we added a small delay to the simulation disk to match the speed of the two. After the delay injection we performed 1000 runs again. The comparison between the simulated disk with added delay and the actual one shows that the average delay is almost identical, and the throughput difference is within 9%. Notably, the delay percentiles P95 and P99 in the simulated disk were reduced by 45%–80% than those in the actual disk. Our tests require the artificial injection of the slow I/Os into the test disk. Therefore, we must avoid the inherent latency variation's impact in the test disk as much as possible. Thus, Ramdisk is a good simulator. The detailed validation of Ramdisk simulation will be presented in Section 6.5.

4.3. PART II : Slow I/O injection

To analyze the impact of SSD's latency variation on the software stack in multiple aspects and scenarios, the injection of slow I/Os, which is used to simulate tail latency in the device, is quite crucial. In this job, fault means performance fault. Therefore, fault injection is equivalent to slow I/O injection. Few previous work on slow I/O injection in RamDisk. We adopt the following methods to inject slow I/O. First, the specified CPU takes over running the simulated RamDisk. Next, we isolated the CPU to ensure that it does not perform other test related operations. Then we modified the RamDisk's source code by

adding *udelay()* to the I/O's bottom operations (i.e., read from RamDisk to buffers or write from buffers to RamDisk). Meanwhile, the CPU is in a busy waiting state during the *udelay()* operation, meaning the CPU does not perform any other operations. This indicates that the slow I/Os have been injected into the underlying device.

After that, the RamDisk source code is compiled and loaded into the Linux system; as a result, it must be reloaded for each injection method. Slow I/O injection is controlled by the three parameters listed below:

- *Ratio* determines the injection proportion.
- *Latency* determines the injection duration.
- *Type* determines the injection type.

We first collected data on the latency distribution of SSD disks under busy conditions over a 72-h period, observing the characteristics of latency during SSD's performance variation. Based on the frequency of these fluctuations, we designed three delay patterns for injection to simulate various scenarios as comprehensively as possible. Additionally, we identified corresponding instances in the PERSEUS [14] latency fluctuation database, providing evidence that our injection models are plausible and may occur in real-world situations. Three delay patterns are listed below:

- *Random injection*. The delay injection is based on a random number, and each one is independent. (e.g. host18, host15 in cluster-D in PERSEUS.)
- *Continuous injection*. This involves concentrating the slow I/Os at the bottom, keeping the underlying equipment at low speed for a certain period. For example, if the ratio is 1%, we injected the delay to 1000 continuous I/O per 100,000 I/Os. (e.g. host1, host3 in cluster-G in PERSEUS.)
- *Interval injection*. This was designed to disperse the slow I/Os in the underlying devices as much as possible. This injection type is controlled using an *interval* instead of a *ratio*. If the *interval* is 100, we injected the first I/O after every 100 I/Os. (e.g. host45 in cluster-E in PERSEUS.)

These types of injection covers the characteristics of various delay distributions. After the delay injection, we treated the RamDisk as a black box and used it as an SSD with latency variation in our experiments. In subsequent experiments, unless otherwise specified, a random injection mode will be used, with a 10% probability of injecting a 10-µs delay. This configuration is chosen because it more accurately reflects the potential performance variation of an SSD under busy conditions [14].

4.4. PART III : eBPF monitoring system

To accomplish our observation goal, we required a tool that could monitor the kernel functions with nanosecond accuracy. We initially inserted the instrumentation directly into the kernel due to the simplicity of this method. However, we encountered the following challenges: first, direct instrumentation generates an additional delay in the kernel that cannot be observed. Therefore, we cannot ignore the impact of the direct instrumentation delay on our experimental results. Besides, monitoring the observation function iteratively during the experiment is necessary for finding the root cause of the delay. Consequently, using the direct instrumentation method means that every time we changed the monitored function, we must recompile the kernel, which takes 10 to 30 min each time. This significantly reduces experiments efficiency.

Accordingly, we developed a system monitoring tool—μScope with eBPF [8,21,22,38–41] technology. We utilized the BPF compiler collection (BCC [42]) tool to write the eBPF programs, which can be used to hook kernel functions in a user-defined way. We mainly used kprobe and kretprobe [43] modules to obtain time information during I/O execution (i.e., the time when the selected function is executed). Then, we processed and analyzed the acquired data in the background. Moreover, the data collection process is quite sensitive to delay. As result, minimizing the additional performance observation burden is challenging. In μScope, adding each observed function generates additional costs. Therefore, the process of selecting the observation function should be thorough and efficient.

The I/O stack has a clear hierarchical relationship in the Linux kernel [44]. As a result, I/O processes have relatively fixed execution paths in the kernel. We have placed the majority of software stack functions and their call relationships in the monitoring system. The original μScope is constructed with the entry function and exit function of each layer of the I/O stack as the observation points. And then we increase the monitored functions during the step-by-step analysis process. For example, whenever we discovered that the block layer had a delay exception during the monitoring process, we made further observations on the block layer. The μScope adds the functions in a deeper level in the block layer to the observation points and so on. In most cases, after three to five iterations, we identified the root cause of the performance issues at the function level. When there were no performance exceptions at each layer under the original μScope, we inferred that there were no performance problems and did not add additional analysis.

Furthermore, before our experimental, we compared the system performance when μScope was turned on and off. We discovered that the performance impact was relatively stable in approximately 20,000 comparative experiments, covering various configurations and workload settings. In other words, the latency fluctuation caused by μScope is much smaller than that of the software stack. Therefore, the delay impact introduced by the detection tool is negligible. In addition, it enabled monitoring various target functions without recompiling the kernel. It is flexible and very easy to apply to other scenarios. By modifying a few code lines for the monitored function, we can monitor the target function and even adapt to any version of the kernel.

5. Experimental setup

5.1. Experimental parameters

Our research objective is to investigate the impact of the SSD's latency variation on the performance of the software storage stack in the kernel. Our experiment focuses on local stand-alone storage systems (e.g., Ext4 [45], F2FS [46], Btrfs [47], XFS [48], SPDK [49]). While selecting configuration items, we observed that our objective aligns with Cao's [5] goal of observing performance variation in modern storage stacks. Therefore, our experimental parameters were inspired by Cao's guidance, which was recommended by several storage experts. During our experiment, we tested five file systems: EXT4, Btrfs, XFS, F2FS and SPDK. They are all widely used in modern systems and cover a variety of current designs and functions.

EXT4 is the abbreviation of the fourth extended file system which is a journaling file system for Linux. It is the default file system for many Linux distributions including Debian and Ubuntu. Our experiment tests three configuration items of EXT4. **Block size.** This is a configuration item of EXT4 and XFS. Block represents a group of continuous sectors and is the basic unit for accessing data in the file system. Too large block size will lead to a lot of waste of space, and a small block size will lead to a reduction in the speed of reading and writing large files. Incorrect block size selection can degrade file system performance by several orders of magnitude [50]. **Inode size.** This is a configuration item of both EXT4, Btrfs and XFS. Inode is one of the most basic disk structures in the file system [26]. It is used to store the metadata of file on disk. Almost all I/O operations are related to inode. Therefore, this configuration item plays a crucial role in performance. **Journal mode**. This configuration is special for EXT4. It determines the journal management mode of the journaling file system. Journaling is a pre-write logging of the file system to recover from power failure or crash. In Ext4, there are three types of journaling modes: *writeback*, *ordered*, and *journal* [45]. Different journaling modes generally bring

Table 3
List of file systems and corresponding parameters.

FS	Parameter	Default	Value range
EXT4	Block size	4096	1024, 2048
	Inode size	256	128, 512, 1024
	Journal mode	Journal	Ordered, writeback
XFS	Block size	4096	1024, 2048
	Inode size	256	128, 512, 2048
	AG count	Based on device size	8, 32, 128
F2FS	gc_idle	Greedy	Cost–Benefit, ATGC
Btrfs	Node size	16 384	4096, 65 536
	Special options	datacow datasum	nodatacow nodatasum
SPDK	Memory zone size	2048 MB	4096, 8192
	I/O model polling	False	True
	Log level	INFO	ERROR, WARN, DEBUG

different I/O overhead, so this configuration item has a great impact on performance.

XFS is a high-performance 64-bit journaling file system created by Silicon Graphics, Inc (SGI) in 1993 and ported to Linux in 2001. It shows high performance and high scalability for large files and large directories on new storage devices. Red Hat Enterprise Linux uses it as default filesystem. Our experiment tests three configuration items of XFS. In addition to **Block size** and **Inode size** introduced earlier, **Allocation group count** is also an important configuration option of the XFS file system. The allocation group enables XFS to have the ability of parallel I/O, which has a great impact on I/O performance.

Btrfs is a modern copy on write (COW) file system for Linux. It has efficient snapshot and cloning technology. It is also strong in fault tolerance, repair and easy administration. Besides node size (similar to inode size), our experiment tests two other configuration items in Btrfs. **Nodatacow.** This is the option to determine whether Btrfs uses COW. When COW is disabled, Btrfs updates in-place when creating new files. Updates in-place improve performance for workloads that do frequent overwrites, at the cost of potential partial writes, in case the write is interrupted due to system crash or device failure. Nodatacow implies nodatasum, and disables compression. **Nodatasum.** This is the option to determine whether Btrfs enables data checksumming. There is a slight performance gain when checksums are turned off, the corresponding metadata blocks holding the checksums do not need to updated.

F2FS (flash-friendly file system) is a kind of new file system designed according to the log-structured file system approach, which adapts to new storage forms. Our experiment tests one key configuration item in F2FS. **gc_idle.**This configuration item determines the algorithm when F2FS performs garbage collection. F2FS is a log-structured file system so that garbage blocks will be generated during operation. It is essential to conduct garbage collection. The algorithms that F2FS can choose during garbage collection include *Cost–Benefit* algorithm, *Greedy* algorithm and *ATGC* (Age Threshold based Garbage Collection) algorithm. Garbage collection generally brings high performance overhead. Therefore, the garbage collection algorithm inevitably has a significant impact on the I/O performance of the file system.

SPDK (Storage Performance Development Kit) is an open-source, high-performance storage framework designed to accelerate storage I/O operations. It is optimized for modern storage hardware, particularly for NVMe (Non-Volatile Memory Express) devices, by bypassing the operating system kernel and interacting directly with hardware, thereby reducing overhead and maximizing performance. **Memory Zone Size.** This configuration specifies the size of the memory zone; larger memory zones can reduce the overhead of memory allocation and deallocation, thus improving performance. **I/O Model Polling.** This configuration determines whether I/O operations are performed using polling or interrupt-driven mode; choosing polling generally improves performance by reducing latency. **Log Level.** This setting controls the verbosity of log outputs; higher log levels can increase overhead and potentially affect performance.

Table 3 summarizes all the parameters and values used in our experiments.

Table 4
Filebench workload settings in our experiments.

Workload	Read/Write	#Files		Test time (s)	
		Default	Actual	Default	Actual
Fileserver	1:2	10,000	80,000	60	800
Webserver	10:1	1000	80,000	60	800
Varmail	1:1	1000	80,000	60	800

5.2. Workload settings

We used the Filebench [24] tool to generate the experiment workload. Compared with other test tools such as FIO [37], Filebench uses real environment trace to simulate a more realistic storage stack situation. Our experiments used the following three pre-configured Filebench workloads listed in Table 4: **Fileserver** writes intensive workloads, and it can be used to simulate most common storage system in the real environment. **Varmail** simulates the I/O operations on the mail servers. It simulates the user's reading, writing, and deleting emails behavior. It also uses a flat directory structure close to the actual mailbox, which tests the I/O's file system capacity within the large directory. Finally, **Webserver** simulates I/O operations on web servers. When users browse web pages, the number of read operations is greater than that of write operations, so the R/W ratio is 10:1 in this workload. In addition, the concurrent nature of multi-threading and the fast reading of small files are important elements of workload testing. The Webserver workload covers typical read-intensive loads. Furthermore, for experimental purposes, we adjusted the file number and test time for each workload. In Table 4, 'Actual' means the configuration actually used in our experiment

During the experiment, we repeated the test 20 times for each configuration item and delay injection method to ensure the result accuracy. We performed approximately 10,020 h of experimental testing. Even though we tested similar servers simultaneously, the experiment enabled eight servers to run for nearly 60 days.

5.3. Hardware setup and Ramdisk configurations

Our experiments were conducted on eight identical Alibaba Cloud servers equipped with the third generation Intel Xeon Scalable processor (i.e., Ice Lake). Memory space is essential because we needed to simulate devices in RAM. Our server was equipped with 64G of memory space and one CPU socket with eight physical cores. Due to the large scale of testing required, we also purchased eight servers with identical configurations and conducted sample experiments before the actual testing to ensure that there were no significant differences in server performance. Furthermore, we installed the Ubuntu 20.04 system on each server, with the kernel upgraded to version 5.8.0.

The Ramdisk in our experiments serves as a simulated SSD within the kernel. Here, we specify the configuration settings of the SSD used in our experiments. Page caching is deliberately disabled during the tests because the extensive I/O operations to SSDs are required for tests. If page caching is enabled, reading and writing to the page would result in significantly reduced testing efficiency. The SSD operates with the default noop I/O scheduler. The driver used by the Ramdisk is provided by the Ramdisk module within the Linux kernel. However, to enhance the accuracy and realism of the tests, we modify the Ramdisk driver to implement a multi-queue mechanism with a queue count of 16 and a queue depth of 64. All settings not explicitly mentioned in the experiments are configured to their default values.

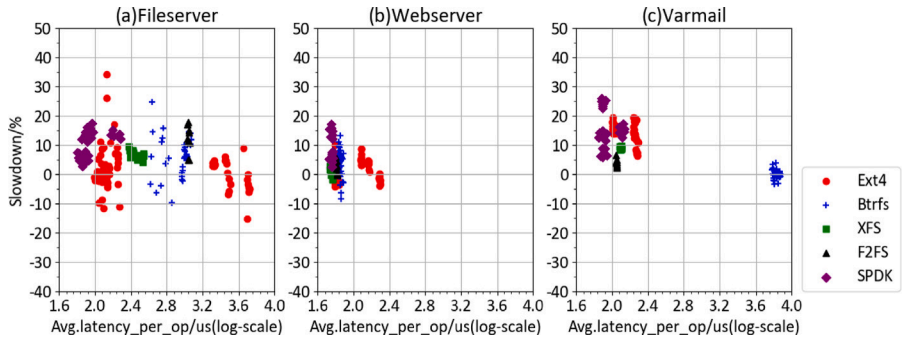


Fig. 2. Overview of latency per operation and performance degradation after injecting slow I/Os with different storage stack configurations under three workloads: (a) Fileserver, (b) Webserver, and (c) Varmail. The x axis represents the mean latency per operation in each workload under normal operation; the y axis shows the percentage of increase after injection. Ext4 configurations are represented with circles, XFS with squares, Btrfs with pluses, F2FS with triangles and SPDK with diamonds. Each scatter point represents a combination of a file system configuration and an injection method.

6. Evaluation and analysis

This study investigates the effect of the SSD's latency variation on the system software stack under different conditions. In the following sections, we explain the experimental results in detail.

Section 6.1 provides an overview of the performance impact in various storage stack configurations and workloads. Sections 6.2–6.4 discuss and make further analysis from three aspects.

6.1. Influence at a glance

We first summarize the performance impact of the underlying latency variation on the system storage stack. Then, we explore the experiment design method introduced in Section 4. We use three typical workloads in Filebench as the benchmarks, whose parameters are shown in Table 4. The experiment's file systems and configurations are listed in Table 3.

Fig. 2 shows the results as scatter plots broken into the three workloads: Fileserver (in Fig. 1(a)), Webserver (in Fig. 1(b)), and Varmail (in Fig. 1(c)) Each symbol represents one storage stack configuration. We use circles for Ext4, squares for XFS, pluses for Btrfs and triangles for F2FS. Moreover, we record the I/O latency in each run and calculated the average. The x-axis represents the average latency per operation in each workload during normal operation. The y-axis shows the average latency's percentage increase per operation (i.e., the slowdown ratio) after the slow I/O injection. Slowdown refers to the ratio between the increase in average latency after injection and the average latency before injection. We observe that a smaller ratio increase results in a reduced bottom-tail latency impact under the current configuration. Each scatter point represents the combination of a file system configuration and injection method.

Ext4's performance varies greatly with the configurations, especially under the Fileserver workload. The small difference between the scatter points on the x-axis suggests that these configurations have similar average latency, while the difference on the y-axis represents the varying configuration sensitivity to the SSD's latency variation. Ext4 had a minimum slowdown ratio of 34% on some of the configurations without excluding the software stack performance fluctuations' impact. In addition, the Ext4 configurations are generally sensitive under the Varmail workload, and can reach a maximum of 19.5%. Moreover, webserver's configuration slowdowns are less than 10.1%.

Compared with Ext4, the scatter points of F2FS is relatively concentrated, indicating that F2FS performance has little impact with a change in configuration. It has a low average latency and is less affected by SSD's latency variation under the Varmail and Webserver. However, under the write-intensive Fileserver workload, the average F2FS latency is quite high, only being lower than that of some Ext4 configurations. At the same time, F2FS is significantly affected by the underlying latency variation, with a slowdown rate of 5.0%–15.1%, which exceeds the configurations of other file systems.

Compared to other file systems, SPDK achieves significantly lower I/O latency by bypassing the kernel. Consequently, along the x-axis, SPDK demonstrates superior performance over all other file systems. However, due to its inherently low latency, it is more sensitive to performance fluctuations. As a result, when slow I/O is introduced, SPDK generally experiences a more pronounced slowdown.

6.2. Access pattern

Symptom. *Write operations involve more disk access than read operations, increasing the chances of latency variation events. The latency increase in write operations is 10.9% to 33.7% higher than the read operations under different configuration conditions.*

Analysis. In general, each file system is more sensitive to the underlying latency variation under the Fileserver and Varmail workloads, which arouse our discussion on R/W ratio.

We compare each operation's delay separately listed in Fig. 3. The figure shows the latency increase ratio of each operation under each workload with default Ext4 configurations. Each column represents one operation under the workload. We observe that the *writfile* operation which means continuous writing, has the highest slowdown ratio up to 13.3%. The slowdown ratio of write operations like *appendfilerand*, *createfile* and *deletefile* operations are 4.8%-8.9%. In contrast, the slowdown ratio of *readfile* operation is about 3.6%-4.3%. The performance of operations unrelated to read and write, such as *openfile* and *closefile*, is not affected after injecting slow I/Os.

Next, we use μScope to monitor the number of system access disks under each workload, which can be determined by setting an observation window and counting the underlying function runs detected by eBPF during the window period. We randomly select 20 observation windows in 5 s units from the test results, which do not overlap. Then, we count the runs and calculate the average value. Next, we perform 10 repeated experiments for each configuration item. We observed that, within the same time frame, the frequency of disk access operations initiated by write requests exceeded that of read operations by 30.9% to 49.7%. In a complete write operation, the interaction with the disk includes reading and writing back, while the read operation does not require writing back. Even though writing backs may be clustered or delayed, write operations bring more disk access in general.

To validate our experiments, we utilize the fio [37] tool to generate workloads with varying Read/Write ratios. The experiments are conducted on an ext4 file system using its default configuration settings. For each read-to-write ratio, we record the average latency from the fio tests and calculate the slowdown ratios of the workloads after injecting latency variations.

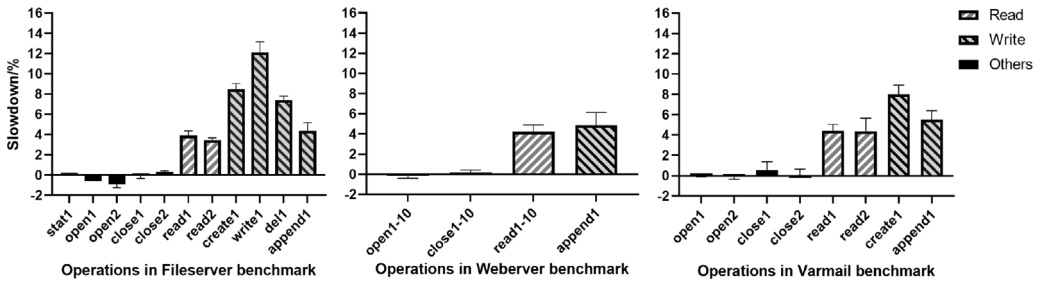


Fig. 3. Each operation's slowdown/% in each workload with the default Ext4 configurations. The white-gray bars represent read related operations and black-gray bars represent write related operations.

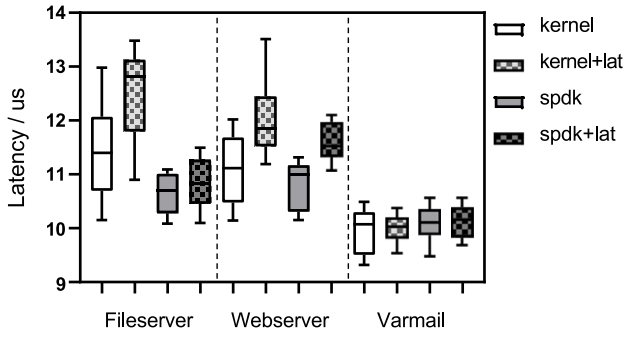


Fig. 4. The latency per operation and performance degradation after injecting latency variation in three workloads based on kernel storage stack and SPDK framework.

The results are illustrated in Fig. 5. As shown, there is no significant trend in the average latency across workloads with different Read/Write ratios. However, as the proportion of write operations increases, the performance degradation becomes more pronounced. This observation further confirms that write operations are more adversely affected by latency spikes at SSDs.

Root Cause. Due to more disk access, write operations are more likely to be affected by the underlying latency variation, so in write-intensive workloads, the potential performance impact of latency variation is more significant.

Lessons. The results indicate that, compared to read requests, writes can suffer more performance impacts under an unstable device. To mitigate this issue, one solution is to employ a thinner storage stack, for example a user-space storage engine. To verify this idea, we further conduct a comparative experiment using storage performance development kit(SPDK) [49] as an example. We build the SPDK architecture on the same system and enabled it to be loaded onto our simulated hard drive. Similarly, we conduct experiments using the workload from Filebench and inject latency variation. The performance results under the SPDK framework and the kernel software stack are compared in Fig. 4.

We can observe from the results that in the Fileserver and Webserver workload, SPDK performs more stably in the face of underlying latency variation compared to the kernel storage stack. SPDK is more resilient to underlying latency variation and has better performance. In Fileserver workload, the slowdown of average latency is 3.06% for SPDK, much smaller than that for kernel stacks(11.08%).

Thinner storage stack effectively reduces the impact of underlying latency variation. However, the polling mechanism causes the underlying slow I/Os to be more directly reflected in the application layer, and does not respond to slow I/Os. If slow fault prediction and processing with light overhead can be added to the existing foundation of SPDK, it will greatly improve performance in low latency hardware environments.

6.3. Configurations

Symptom. *The configurations related to consistency and performance in the file system may lead to the dependence between software execution functions, increasing the impact of SSD's latency variation on software stack by more than 10%.*

Analysis. The scatter points in Fig. 2 indicate that file systems under different configurations can have different sensitivity to SSD's latency variation. We use the Ext4 system as a representative case to further study the relationship between configurations and such sensitivity. Fig. 6 compares the average latency of Ext4 under different configurations and the slowdown ratios after the injection of slow I/Os. We rerun the same experiments shown in Fig. 2. While, due to space limit, we only demonstrate Ext4, we reach similar conclusions for the other file systems. The bars represent average latency, and correspond to the left y-axis. The slowdown ratio for each configuration is shown as symbols and corresponds to the right y-axis. The x-axis lists the configuration details, and is formatted as the three-part tuple < Block Size — Inode Size — Journal Mode>.

Fig. 6 shows, in most workloads, the Ext4 configuration with *ordered* often has a larger slowdown than *writeback*. When Journal Mode is *journal*, there is no necessary relationship. We use μScope to monitor the journal-related functions and the performance exception of *journal_commit_transaction()* is the most noticeable. Also, *journal_commit_transaction()* is the commit function of journal transactions, which is the main function to execute journal functions. As shown in Fig. 7-T, in the *ordered* mode, after the bottom slow I/O injection, the execution time of the function has changed significantly. Its average latency has increased by 53.2% after the injection.

We perform a more fine-grained monitoring within the function. The commit of journal transactions is divided into the following six steps: **1.** Pre-processing of transaction information. **2.** Submission of data buffers. If Journal Mode = *ordered*, kernel must wait for the write operation to be completed. If Journal Mode = *writeback* or *journal*, it will be executed directly after submission. The function of this step is *journal_submit_data_buffers()*. **3.** Writing the metadata block buffer to journal. The main monitored function of this step is *journal_write_metadata_buffer()*. In all the modes, this step requires waiting for the metadata to be written before proceeding. The main monitored function of this step is *journal_write_metadata_buffer()*. If Journal Mode = *journal*, after step three, there is an additional step to write the data blocks to the journal. **4.** Writing journal transaction control part, including descriptor, etc. This step needs to be executed in every mode. **5.** Writing the commit block. After ensuring that all are correctly written, the kernel writes the journal's commit block synchronously. The main monitored function of this step is *journal_write_commit_record()*. **6.** Completing the transaction commit. Finally, this transaction is added to the checkpoint queue, indicating that the transaction submission has been completed.

However, *journal_submit_data_buffers()* only adds the data block buffer to the write disk queue. There is no specific function for the operation waiting to be written. Therefore, we calculate the gap of

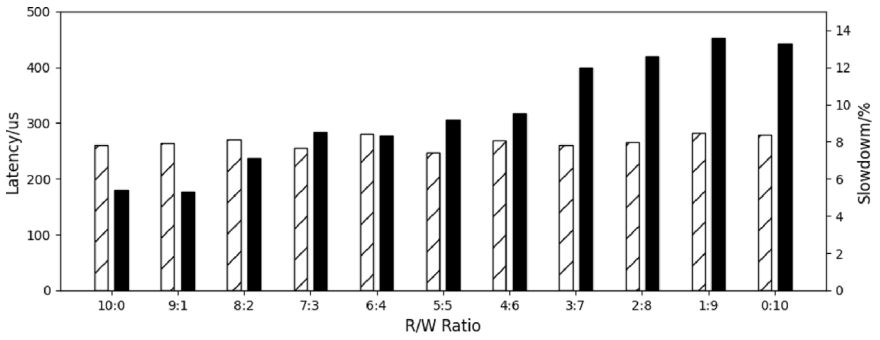


Fig. 5. The average latency and performance degradation after injecting latency variations in workloads with different Read/Write Ratio. The average latency is represented by white bars with diagonal stripes, with the left *y*-axis used for scaling. The slowdown is represented by black bars, with the right *y*-axis used for scaling.

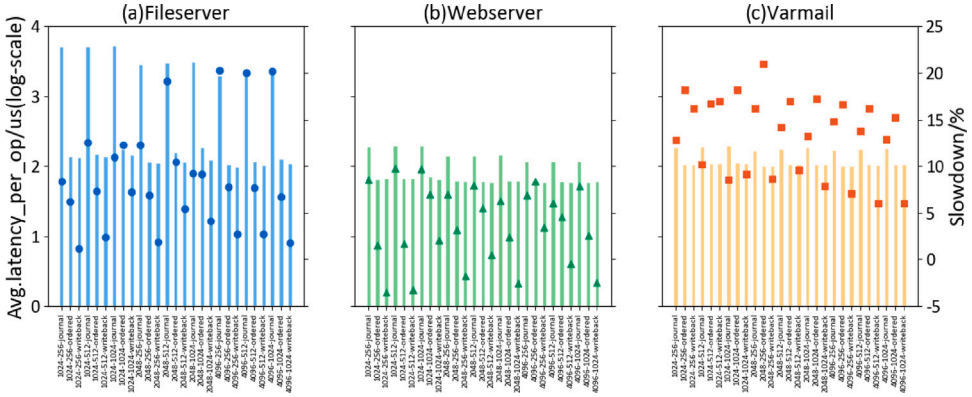


Fig. 6. The average latency of Ext4 and the slowdown ratio after the injection of slow I/Os under different configurations. The bars represent the average latency of operations, and correspond to the left *y*-axis. The slowdown ratio for each configuration is shown as symbols, and corresponds to the right *y*-axis. The *x*-axis consists of configuration details, and is formatted as the three-part tuple < Block Size — Inode Size — Journal Mode>.

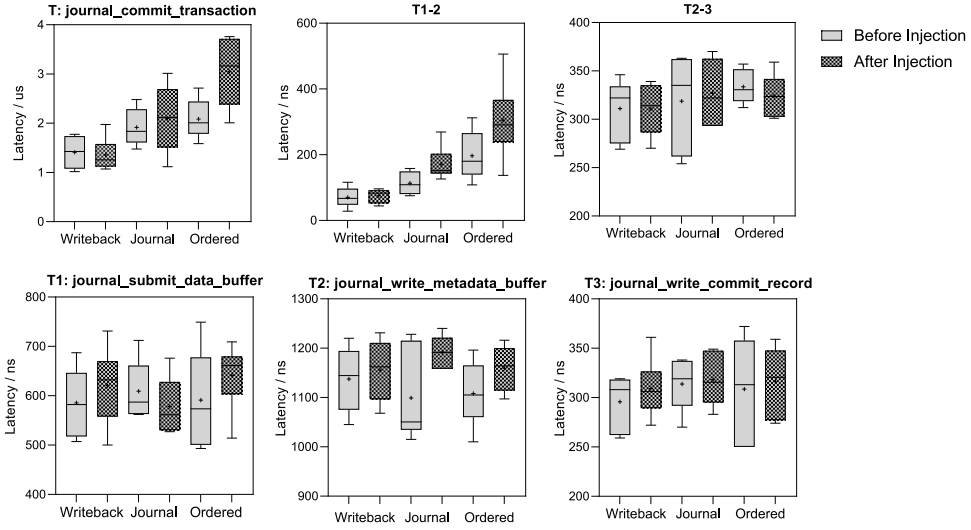


Fig. 7. The latency of *journal_commit_transaction()* and lower-level functions before and after the injection of slow I/Os (monitored by eBPF). T1–2: the waiting time between T1 and T2. T2–3: the waiting time between T2 and T3.

the end time of *journal_submit_data_buffers()* and the begin time of *journal_write_metadata_buffer()*, which is almost equivalent to waiting time(T1–2). We carry out our experiments with the performance monitoring of the three sub-functions below *journal_commit_transaction()* and the gap between them.

Thus, we use μScope to monitor these functions in detail and results are shown in Fig. 7. T1–2 and T2–3 means the waiting time between T1, T2 and T3. The results show that although the latency of the three sub-functions increases after the injection of slow I/Os, they are mild and not related to the journal mode. In contrast, the latency growth of waiting time T1–2 is quite noticeable in the *ordered* mode, with the average latency increasing by 51.0% and the 95th percentile rising by 239.2%. In the *writeback/journal* mode, the waiting time is approximately 0. Journal Mode configuration is just one example. We have observed similar results on other configurations.

Root Cause. The different Journal Modes result in various execution processes. The dependence during software execution, in other words, the execution sequence of functions, can easily amplify the

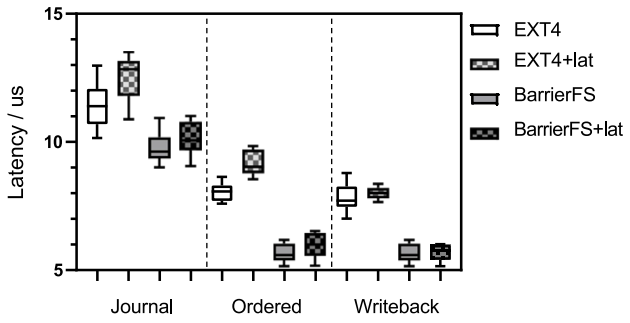


Fig. 8. The latency per operation and performance degradation after injecting latency variation in Fileserver workload based on EXT4 and BarrierFS.

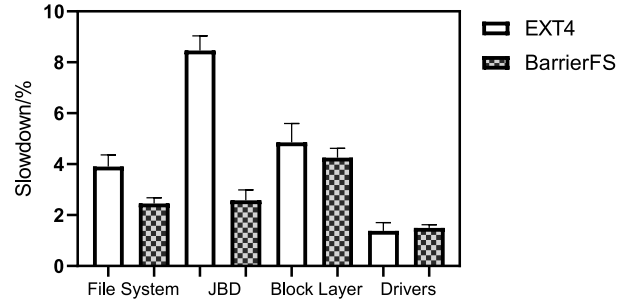


Fig. 9. The slowdown of different layers after injecting latency variation in Fileserver workload based on EXT4 and BarrierFS.

impact of SSD's latency variation. The greater the dependence between functions and bottom I/Os during software execution, the more significant the impact on the storage stack against SSD's latency variation.

Lessons. Dependable I/Os are inevitable as they are driven by the applications (i.e., workloads). However, due to the chaotic nature of multiple layers across the stack, the order of persisting I/Os can be altered by the file system, block layer (i.e., IO scheduling), driver and the device itself (e.g., write retry). Traditionally, enforcing the order to maintain consistency demands the each dependable IO to be persisted by the device before issuing the next one, thereby yielding such high overhead as shown in our experiments.

To reduce the impacts from dependable I/Os, one effective solution is to separate the order logic from the durability. Previous work [51] shows that one can leverage a specific set of APIs (i.e., `barrier`) to explicitly maintain the order across all layers. We attempt to use `barrier` technology to address the cascade influence caused by the order logic. `Barrier` technology changes the read and write mode of the kernel stack on the basis of existing systems, separating data read and write from sequence. Due to the fact that BarrierFS is developed based on EXT4, we conduct comparative experiments using BarrierFS and EXT4. We use Fileserver as workload and adjust the log-related configuration of the file system. All other configurations are tested using the default value. The result is shown in Fig. 8.

We can observe from the results that `barrier` greatly reduces the impact of underlying latency variation on the software stack while improving file system performance. This is particularly significant in Journal and Ordered modes. After injecting latency variation, the slowdown in BarrierFS is 7.6% and 3.4% less than that in EXT4 in Journal and ordered modes.

However, BarrierFS did not completely solve the problem. We use μscope for further analysis. We divide the main processes of the I/O stack in the kernel into File System, Journaling Block Device (JBD), Block Layer, and Drivers. And then we measure the slowdown in latency of these parts after injection latency variation, as shown in Fig. 9.

As shown in the figure, the mechanism of BarrierFS efficiently improves the performance Robustness in File System and JBD. But in Block Layer, it does not have a very good effect. This is mainly because BarrierFS mainly implements software level separation of sequence and data, which achieves performance improvement when sending requests downwards in the kernel. However, when it comes to submission to hardware, there is still a situation of queuing and waiting, which has not fundamentally avoided the impact of underlying latency. If such ideas can be applied to the entire storage stack, it will help minimize the impact of latency variation.

6.4. Pattern of latency variation

Symptom. *Continuous slow I/Os can exert worse impacts on the performance of the software stack. In situations with the same occurrence ratio, the software stack delay caused by continuous slow I/Os increases by 14.7% more than that caused by random slow I/Os.*

Analysis. There are various reasons for the SSD's latency variation, and the patterns are also different. To ensure the comprehensiveness of the experiment, we design three different injection methods for injecting slow I/Os, which are introduced in Section 4.3.

When we inject a 10% delay of 10 µs, the multi-queue I/O characteristics of the SSD mitigate the impact of consecutive injections. This is because, regardless of whether the I/O operations are completed, as long as there are available queues, the software stack can continue to send I/O requests downstream, preventing cascading effects even in the case of consecutive I/Os. However, as we increase the injection ratio, this signifies a more significant performance fluctuation in the simulated SSD, and the impact of consecutive slow I/O injections gradually becomes apparent. When we inject a 20% delay of 10 µs, the consecutive slow I/O injections lead to more significant delay slowdown compared to random slow I/O injections.

Fig. 10 illustrates this result, showing the each file system's slowdown ratio with the default configuration generated using different injection methods. From this, we can also infer that as the injection ratio increases further, the cascading effects caused by consecutive slow I/O operations will become more pronounced. We examine the underlying cause of this phenomenon using the example of injecting 20%–10 µs delays. show the each file system's slowdown ratio with the default configuration generated using different injection methods. We use μScope to monitor functions to find the root cause of this phenomenon. The results indicate that different patterns have varying performance impacts, and this occurs in the Generic Block Layer and lower levels. Therefore, we speculate that the root cause occurred in the Generic Block Layer.

As shown in Fig. 11, the delay of the Generic Block Layer changes over time. Since there were no performance anomalies during the stage where the latency variation did not occur, we refined the observation range to two cycles of slow I/O injection (i.e., one cycle per 1000). We observe that the delay of the Generic Block Layer shows a trend over time, slowly increasing and then decreasing, in a cyclic manner. It is most evident in continuous injection.

Furthermore, we conduct monitoring on low-level functions for root cause analysis. An in-depth analysis of the functions in the Generic Block Layer reveals that the queue issue increased the latency. Although multi-queue mechanisms alleviate the blocking issues within the software stack to some extent, when slow failures become more severe, the latency at the generic block layer continues to rise progressively. When the underlying latency variation occurs, the request's processing delay becomes longer and the bio queue becomes blocked, resulting in bio accumulation in the waiting queue. For functions that block sending bio, their latency also increases accordingly. Once the potential variations in latency subside and the Ramdisk is able to efficiently handle I/O requests, the blocking issues gradually dissipate within a short period of time.

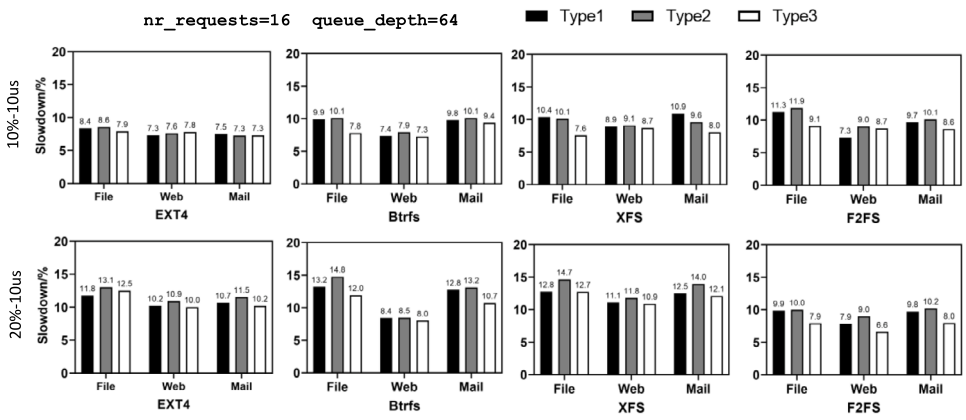


Fig. 10. The slowdown ratio of each file system generated by different injection methods. There are three types of slow I/O injection. As the injection ratio increases (due to the deterioration of hardware latency variation), the cascading effects resulting from consecutive slow I/Os become more pronounced. Type1: Random Injection; Type2: Continuous Injection; Type3: Interval Injection.

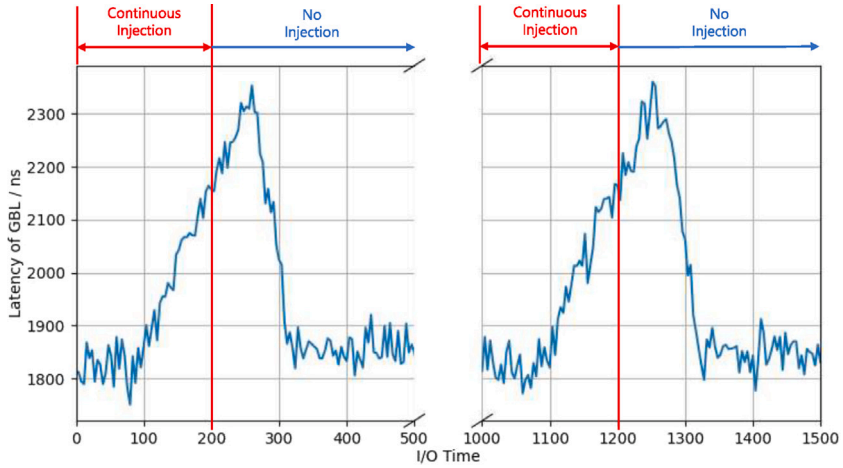


Fig. 11. The Latency (ns) of Generic Block Layer when encountering continuous SSD's latency variation (in two cycles). The red periods represent intervals during which continuous latency variation are injected, while the blue periods indicate intervals without any injection. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Therefore, if the latency variation occurs through continuous slow I/O, the bio sent by the Generic Block Layer to the waiting queue will also accumulate, significantly impacting the performance. Hence, based on results of different patterns, we discover that the more dispersed slow I/O is, the less impact it has on function latency.

Root Cause. As the block layer queue temporarily accumulates SSD's latency, continuous slow I/Os can negatively impact the performance of the software stack considerably. This phenomenon becomes more pronounced when there are significant bottom latency variations, as the multi-queue mechanism helps mitigate some of the issues related to continuous slow I/Os.

Lessons. First of all, there are many instances of continuous slow I/Os in latency variations (e.g. fail-slow [10]). It cannot be avoided in low-latency SSDs. Based on the current experimental results, one apparent solution is to increase the queue depth and the number of queues. However, this is also dependent on the inherent parallelism capabilities of the SSD. This approach, however, is not universally applicable to all users. If this issue cannot be effectively resolved on a single SSD, it appears that we could address it within a multi-disk array or a distributed system. Forwarding I/O requests from slow disks is an effective method for mitigating the impact of slow failures.

Currently, some work has been done to avoid the influence of Fail-slow, such as LinnOS [52] predicting the slowdown of underlying I/O in the kernel. To encounter the performance degradation caused by continuous slow I/Os, we try to reproduce LinnOS and mount it in our system. However, LinnOS only predicts individual slowness. The accuracy decreases significantly when it predicts continuous slow I/Os. From the comparison of our experimental results, it can be seen that individual slowness has little impact, while continuous ones can lead to more serious performance impacts. It is important and urgent to implement effective continuous slow I/O prediction.

When continuous slow I/Os occurs, the commonly used distributed approach is to switch nodes. Switching nodes usually takes around 3 ms–10 ms by testing. This is an expensive expense for low latency systems. According to previous research [10,17], the latency variation of the disk is generally not widespread. In the vast majority of cases, only a single disk is slow. So the Fail-slow granularity is likely to be the disk rather than the node. We simulate the time required for switching disks, which only requires 5 μs-17 μs. In high-performance scenarios, switching disks is a promising choice to mitigate impacts from SSD's latency variations.

6.5. Validation of Ramdisk

In this section, we evaluate the effectiveness of using Ramdisk to simulate SSDs. Given that FEMU [36] exhibits performance characteristics comparable to real SSDs and is capable of simulating SSD activities (although excluded due to high CPU overhead), we conduct latency tests by comparing real SSDs, FEMU-based SSD simulations, and Ramdisk-based SSD simulations. The results of the average latency test

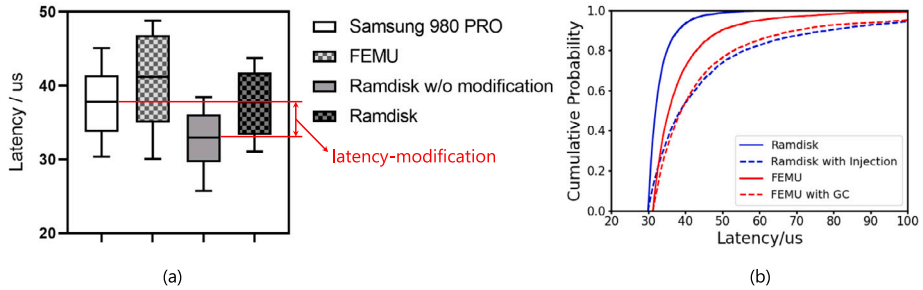


Fig. 12. (a) Average Latency of the SSD and Three Simulated SSDs. (b) The cumulative distribution probability (CDF) plots of the Ramdisk before and after the injection of performance fluctuations (10% with a 10 μs delay), as well as those of FEMU before and after simulated garbage collection.

are presented in Fig. 12(a). As mentioned in Section 4.2, the unmodified Ramdisk exhibits lower operational latency compared to a physical SSD. Therefore, we first introduce a fixed delay in Ramdisk to ensure that its average latency aligns with that of the physical SSD.

Next, we address the issue of performance fluctuations. We utilize FEMU to simulate garbage collection actions in SSDs and plot the corresponding latency distribution curve. Subsequently, we adjust the latency injection parameters. When injecting a continuous 10%–10 μs delay into the Ramdisk, the latency distribution curve of the Ramdisk aligns closely with that of FEMU with GC, as shown in Fig. 12(b). This shows that the flexible adjustment of latency injection parameters in the Ramdisk can effectively replicate nearly all latency fluctuations caused by internal SSD mechanisms.

This indicates that most latency reductions observable in SSD operations can be effectively simulated using Ramdisk. Despite the internal complexity of SSD mechanisms, e.g., garbage collection and wear leveling, these devices are essentially "black boxes" to the kernel software stack. Their external behavior is reflected mainly in the latency variation [4]. Thus, Ramdisk-based simulations provide a broader range of latency variation modeling.

To further validate this hypothesis, we conduct real-SSD testing. We perform high-load I/O operations on a Samsung 980 PRO and record latency within a one-minute time window. If the average latency within one minute exceeds 10% of the previously tested value, it is considered a warning of severe performance degradation. We collect 1200 data samples, resulting in 27 warnings. We then plot the latency distribution curves for each of these one-minute intervals. Subsequently, we try to adjust the latency injection parameters (including injection *ratio*, *latency* and *type* in Section 4.3), all of which lead to the Ramdisk's latency distribution curve closely resembling the latency distribution curve of the actual SSD. In addition, we use average latency, P50, P90, and P95 as validation points. The first three latency metrics of the Ramdisk differ from the actual drive by no more than 5%, and P95 differs by no more than 10%. This suggests that most of the complex mechanisms within the actual SSD, which contribute to latency variations, can be effectively replicated by the Ramdisk simulation.

Overall, the Ramdisk simulation provides a broader range of performance fluctuation modeling compared to simulations focused on specific mechanisms (such as FEMU). By utilizing Ramdisk, we gain finer control over latency distribution, allowing us to simulate a wider array of potential real-world drive latencies. This enables the observation of the impact on the software stack across a broader spectrum of possible scenarios.

6.6. Overhead

To test the load of μScope, we conduct the following experiments. Firstly, we use the Filebench test without any monitoring tools as the baseline. Based on it, the observation functions are set to the entry function and exit function of each layer of the I/O stack (VFS, File System, Block Layer, Drivers and Device). We run 10 repeated experiments on different benchmarks and the result is shown in Fig. 13. The average overload added by μScope (4.77%) is similar to Ftrace (6.49%), far lower than that brought by Perf (12.7%). But the monitoring granularity of Ftrace is at the microsecond level, while the monitoring granularity of μScope is at the nanosecond level. Overall, the μScope can add less monitoring load while ensuring monitoring accuracy.

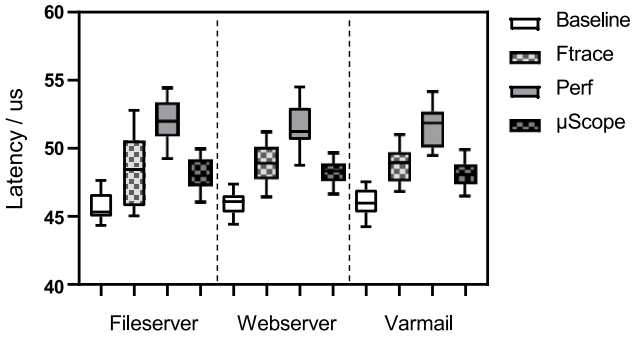


Fig. 13. The average latency (μs) in Filebench result using different monitoring tools under three benchmarks.

7. Limitations and discussion

In μScope, we use a RamDisk-based SSD as the media to inject delay. In practice, real SSDs may suffer from complicated or alternative failure modes that are not covered by our testing framework. As a result, our analysis may not reflect these cases and thus limits the thoroughness of our findings. However, we would argue that, by simulating a wide variety of delay patterns, μScope is capable of covering most scenarios.

Moreover, we use the purest device–RamDisk for the experiment, and it definitely expose the simplest problems. If it involves actual SSDs, there will definitely be more complex situations. Considering the various mechanisms in FTL within the actual SSDs, including garbage collection, wear level, etc., problems may be more interesting. Therefore, further observation and research are worth researching, but more fine-grained monitoring and analysis are also necessary conditions for deeper research.

Another potential limitation comes from our selection on the workloads. In this paper, we use popular benchmarks from the field, such as Fileserver and Varmail. Consequently, certain access patterns or specific attributes may not be included in our study. Yet, we believe that, with our μScope open-sourced, interested parties can easily reuse our toolkit to examine their particular workloads and subsequently identify the suboptimal performance issues.

8. Related work

SSD's Fail-slow. To the best of our knowledge, there is no systematic research on the impact on the software stack against SSD's

latency variation. However, the Fail-slow problems, a typical latency problems in hardware, have always been a concern. Specifically, Hao et al. [11] studied storage performance in over 450,000 disks and 4000 SSDs over 87 days. They proved that Fail-slow problems are common. Furthermore, the examples of SSD tail latency motivate us to start this work. Gunawi et al. [16] studied over 100 reports of Fail-slow hardware incidents from large-scale cluster deployments, which showed that all hardware types such as SSD can exhibit Fail-slow problems. Lu et al. [17] collected logs from over one million NVMe SSDs deployed at Alibaba to study the characteristics of Fail-slow in NVMe SSDs.

Stack Performance. With the rapid decline of hardware latency, the performance of software stack has gradually attracted people's attention. Cao et al. [5] provided the first systematic study on performance variation in modern storage stacks. Zhong et al. [8] pointed out that software was now the storage bottleneck. They presented a framework by to accelerating R/W by safely bypassing most of the kernel's storage stack with eBPF. Liao et al. [53] have noticed the performance issue with software stack and proposed some improvement methods for taking full advantage of high-performance drives.

9. Conclusion

In this work, we provide the first systematic study on the impact of SSD's latency variation on software stack performance. We have designed and implemented μScope to evaluating storage stack robustness against SSD's Latency Variation. Although most of our observations are executed in the experimental environment, we believe that they provide valuable insight into the impact of latency variation in real industrial systems.

Based on our experimental results, we have listed three practices for minimizing the impact of latency variation when using new low-latency hardware devices:

(1) Write operations involve more disk accesses than read operations, increasing the chances of latency variation events. Therefore, by leveraging persistent memory or user-space stacks, the impact can be efficiently mitigated.

(2) The configuration item that establishes dependence between functions, is easy to cause sensitivity to the SSD's latency variation. A promising suggestion is to separate the order logic from the durability, which help improve the OS stack to adapt to new devices.

(3) As the block layer queue accumulates SSD's latency temporarily, continuous slow I/Os can negatively impact the performance of the software stack considerably. It is suggested to reasonably set up the algorithm for redirected write in distributed systems.

CRediT authorship contribution statement

Linxiao Bai: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Data curation. **Shanshan Li:** Writing – review & editing, Supervision, Methodology, Funding acquisition. **Zhouyang Jia:** Writing – review & editing, Software. **Yu Jiang:** Writing – review & editing, Methodology. **Yuanliang Zhang:** Writing – review & editing, Software, Methodology. **Zichen Xu:** Writing – review & editing. **Bin Lin:** Writing – review & editing. **Si Zheng:** Writing – review & editing. **Xiangke Liao:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared our data and code on Github: https://github.com/uScope/uScope.

References

[1] D.G. Andersen, S. Swanson, Rethinking flash in the data center, IEEE Micro 30 (4) (2010) 52–54, http://dx.doi.org/10.1109/MM.2010.71.

[2] M. Mesnier, F. Chen, T. Luo, J.B. Akers, Differentiated storage services, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 57–70, http://dx.doi.org/10.1145/2043556.2043563.

[3] K. Wu, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Towards an unwritten contract of intel optane SSD, in: G. Yadgar D. Peek (Ed.), 11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019, USENIX Association, 2019, pp. 124–156, URL https://www.usenix.org/conference/hotstorage19/presentation/wu-kan.

[4] J. He, S. Kannan, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, The unwritten contract of solid state drives, in: G. Alonso, R. Bianchini, M. Vukolic (Eds.), Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, ACM, 2017, pp. 127–144, http://dx.doi.org/10.1145/3064176.3064187.

[5] Z. Cao, V. Tarasov, H.P. Raman, D. Hildebrand, E. Zadok, On the performance variation in modern storage stacks, in: Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST'17, USENIX Association, USA, 2017, pp. 329–343.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key–value store, in: ACM Symposium on Operating System Principles, 2007, pp. 127–144, URL https://www.amazon.science/publications/dynamo-amazons-highly-available-key–value-store.

[7] B. Trushkowsky, P. Bodík, A. Fox, M. Franklin, M. Jordan, D. Patterson, The scads director: Scaling a distributed storage system under stringent performance requirements, in: ACM Symposium on Operating System Principles, 2011, pp. 163–176.

[8] Y. Zhong, H. Li, Y.J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, A. Cidon, XRP: In-Kernel storage functions with eBPF, in: 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 22, USENIX Association, Carlsbad, CA, 2022, pp. 375–393, URL https://www.usenix.org/conference/osdi22/presentation/zhong.

[9] A. Yoo, Y. Wang, R. Sinha, S. Mu, T. Xu, Fail-slow fault tolerance needs programming support, in: Proceedings of the Workshop on Hot Topics in Operating Systems, 2021, URL https://api.semanticscholar.org/CorpusID:234775284.

[10] J. Dean, L.A. Barroso, The tail at scale, Commun. ACM 56 (2013) 74–80, URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[11] M. Hao, G. Soundararajan, D.R. Kenchammanahosekote, A.A. Chien, H.S. Gunawi, The tail at store: A revelation from millions of hours of disk and ssd deployments, Comput. Math. Appl. 22 (7) (2016) 37–42.

[12] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, H.S. Gunawi, Iaso: A fail-slow detection and mitigation framework for distributed storage services, in: USENIX Annual Technical Conference, 2019, pp. 111–130, URL https://api.semanticscholar.org/CorpusID:195781797.

[13] J. Hao, Y. Li, X. Chen, T. Zhang, Mitigate hdd fail-slow by pro-actively utilizing system-level data redundancy with enhanced hdd controllability and observability, in: 2019 35th Symposium on Mass Storage Systems and Technologies, MSST, 2019, pp. 205–216, URL https://api.semanticscholar.org/CorpusID:207890893.

[14] R. Lu, E. Xu, Y. Zhang, F. Zhu, Z. Zhu, M. Wang, Z. Zhu, G. Xue, J. Shu, M. Li, J. Wu, Perseus: A fail-slow detection framework for cloud storage systems, in: USENIX Conference on File and Storage Technologies, 2023, pp. 238–256, URL https://api.semanticscholar.org/CorpusID:257285445.

[15] T. Fujimori, S. Nomura, BOOSTER: rethinking the erase operation of low-latency ssds to achieve high throughput and less long latency, in: Y. Moatti, O. Biran, Y. Gilad, D. Kostic (Eds.), Proceedings of the 16th ACM International Conference on Systems and Storage, SYSTOR 2023, Haifa, Israel, June 5-7, 2023, ACM, 2023, pp. 94–104, http://dx.doi.org/10.1145/3579370.3594774.

[16] H.S. Gunawi, R.O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, D. Srinivasan, B. Panda, A. Baptist, G. Grider, P.M. Fields, K. Harms, R.B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H.B. Runesha, M. Hao, H. Li, Fail-slow at scale: Evidence of hardware performance faults in large production systems, ACM Trans. Storage 14 (3) (2018) 23, http://dx.doi.org/10.1145/3242086.

[17] R. Lu, E. Xu, Y. Zhang, Z. Zhu, M. Wang, Z. Zhu, G. Xue, M. Li, J. Wu, NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow, in: 2022 USENIX Annual Technical Conference, USENIX ATC 22, USENIX Association, Carlsbad, CA, 2022, pp. 1005–1020, URL https://www.usenix.org/conference/atc22/presentation/lu.

[18] J. Hao, Y. Li, X. Chen, T. Zhang, Mitigate HDD fail-slow by pro-actively utilizing system-level data redundancy with enhanced HDD controllability and observability, in: 35th Symposium on Mass Storage Systems and Technologies, MSST 2019, Santa Clara, CA, USA, May 20-24, 2019, IEEE, 2019, pp. 205–216, http://dx.doi.org/10.1109/MSST.2019.000-2.

[19] Ltrace(1) — linux manual page, 2017, https://man7.org/linux/man-pages/man1/ltrace.1.html.

[20] Strace(1) — linux manual page, 2018, https://man7.org/linux/man-pages/man1/strace.1.html.

[21] A thorough introduction to ebpf, 2017, https://lwn.net/Articles/740157/.

[22] Extending extended bpf, 2014, https://lwn.net/Articles/603983/.

[23] What is a ram disk, 2024, https://www.kingston.com.cn/en/blog/pc-performance/what-is-ram-disk.

[24] Filebench, 2016, https://github.com/filebench/filebench.

[25] Linux storage stack diagram, 2020, https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[26] R.H. Arpaci-Dusseau, A.C. ArpaciDusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Academic, 2015.

[27] Overview of the linux virtual file system, 2002, https://www.kernel.org/doc/html/latest/filesystems/vfs.html.

[28] The linux kernel user-space api guide, 2021, https://www.kernel.org/doc/html/v4.14/userspace-api/index.html.

[29] Linux page cache basics, 2005, https://www.thomas-krenn.com/en/wiki/Linux_Page_Cache_Basicacs.

[30] A block layer introduction, 2017, https://lwn.net/Articles/736534/.

[31] Linux storage stack diagram, 2008, https://docs.kernel.org/trace/ftrace.html.

[32] Iostat, 2016, https://linux.die.net/man/1/iostat.

[33] Linux performance observability tools, 2021, https://www.brendangregg.com/linuxperf.html.

[34] Ssdsim, 2009, https://github.com/xinglin/SSDSim.

[35] Simplessd, 2018, https://docs.simplessd.org/en/v2.0.12/.

[36] H. Li, M. Hao, M.H. Tong, S. Sundararaman, M. Bjørling, H.S. Gunawi, The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator, in: 16th USENIX Conference on File and Storage Technologies, FAST 18, USENIX Association, Oakland, CA, 2018, pp. 83–90, URL https://www.usenix.org/conference/fast18/presentation/li.

[37] Flexible i/o tester, 2022, https://github.com/axboe/fio.

[38] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, G. Carle, Performance implications of packet filtering with linux ebpf, in: 2018 30th International Teletraffic Congress, ITC 30, 2018, pp. 209–217, http://dx.doi.org/10.1109/ITC30.2018.00039.

[39] S. Miano, X. Chen, R.B. Basat, G. Antichi, Fast in-kernel traffic sketching in ebpf, Comput. Commun. Rev. 53 (1) (2023) 3–13, http://dx.doi.org/10.1145/3594255.3594256.

[40] M.H.N. Mohamed, X. Wang, B. Ravindran, Understanding the security of linux ebpf subsystem, in: Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2023, Seoul, Republic of Korea, August 24-25, 2023, ACM, 2023, pp. 87–92, http://dx.doi.org/10.1145/3609510.3609822.

[41] M. Abranches, O. Michel, E. Keller, S. Schmid, Efficient network monitoring applications in the kernel with ebpf and XDP, in: 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2021, Heraklion, Greece, November 9-11, 2021, IEEE, 2021, pp. 28–34, http://dx.doi.org/10.1109/NFV-SDN53031.2021.9665095.

[42] Bcc - tools for bpf-based linux io analysis, networking, monitoring, and more, 2020, https://github.com/iovisor/bcc.

[43] Extracting kprobe parameters in ebpf, 2019, https://eyakubovich.github.io/2022-04-19-ebpf-kprobe-params/.

[44] Linux storage stack diagram, 2018, http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[45] An introduction to linux's ext4 filesystem, 2017, https://opensource.com/article/17/5/introduction-ext4-filesystem.

[46] C. Lee, D. Sim, J. Hwang, S. Cho, F2FS: A new file system for flash storage, in: 13th USENIX Conference on File and Storage Technologies, FAST 15, USENIX Association, Santa Clara, CA, 2015, pp. 273–286, URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[47] O. Rodeh, J. Bacik, C. Mason, Btrfs: The linux b-tree filesystem, ACM Trans. Storage (TOS) 9 (2013) http://dx.doi.org/10.1145/2501620.2501623.

[48] C. Hellwig, Xfs: The Big Storage File System for Linux, vol. 34, login Usenix Mag, 2009.

[49] Z. Yang, J.R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, L.E. Paul, Spdk: A development kit to build high performance storage applications, in: 2017 IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2017, pp. 154–161, http://dx.doi.org/10.1109/CloudCom.2017.14.

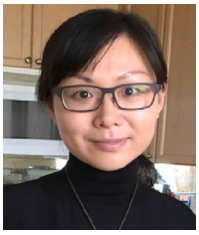
[50] J. He, D. Nguyen, A. Arpaci-Dusseau, R. Arpaci-Dusseau, Reducing file system tail latencies with chopper, in: 13th USENIX Conference on File and Storage Technologies, FAST 15, USENIX Association, Santa Clara, CA, 2015, pp. 119–133, URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/he.

[51] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, S. Cho, Barrier-enabled IO stack for flash storage, in: 16th USENIX Conference on File and Storage Technologies, FAST 18, USENIX Association, Oakland, CA, 2018, pp. 211–226, URL https://www.usenix.org/conference/fast18/presentation/won.

[52] M. Hao, L. Toksoz, N. Li, E.E. Halim, H. Hoffmann, H.S. Gunawi, LinnOS: Predictability on unpredictable flash storage with a light neural network, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, USENIX Association, 2020, pp. 173–190, URL https://www.usenix.org/conference/osdi20/presentation/hao.

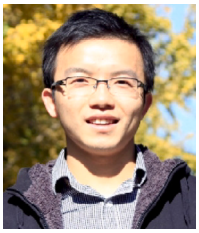
[53] X. Liao, Y. Lu, E. Xu, J. Shu, Write dependency disentanglement with HORAE, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, USENIX Association, 2020, pp. 549–565, URL https://www.usenix.org/conference/osdi20/presentation/liao.

Linxiao Bai received his B.S. degree from the College of Computer Science and Technology at National University of Defense Technology, Changsha, China, in 2020. He is currently pursuing a doctoral degree at the College of Computer Science, National University of Defense Technology. His research focuses on the detection and resolution of performance issues in software stacks within low-latency hardware environments.



Shanshan Li received her Ph.D degrees from the College of Computer Science and Technology at National University of Defense Technology, Changsha, China in 2007. She is currently working as a professor at National University of Defense Technology. Her primary research interests include software defect detection and intelligent software development. She has published multiple papers in top-tier conferences related to software engineering, such as ICSE (Distinguished Paper Award), ISSTA, and ASE.

Zhouyang Jia received his B.S., M.S. and Ph.D degrees from the College of Computer Science and Technology at National University of Defense Technology, Changsha, China, in 2013, 2015 and 2020, respectively. He was a visiting student at University of Kentucky, USA, from 2018 to 2020. He is currently working as an associate professor at National University of Defense Technology. His research interests include software reliability and operating system.



Yu Jiang received his Ph.D degrees from the Department of Computer Science at Tsinghua University in 2015. He is currently an associate professor at the School of Software at Tsinghua University. He focuses on the security of software systems such as databases, kernels, and industrial control. He has published multiple papers in top-tier conferences, such as ACM SOSP, IEEE S&P and USENIX Security.

Yuanliang Zhang is an assistant professor in the College of Computer Science of National University of Defense Technology. His main research interests are system reliability and AI4SE. He has published multiple papers in top-tier conferences related to software engineering, such as ICSE, ISSTA, and ICLR.

Zichen Xu (Senior Member, IEEE) received the Ph.D. degree from the Ohio State University. He is currently a full professor at Nanchang University, China. His research interests include tempo-spatial data-conscious complex system, including performance optimization, analysis, and system design/implementation. He is also a member of ACM and a distinguished member of CCF.

Bin Lin graduated from the College of Computer Science and Technology at National University of Defense Technology, Changsha, China, in 2014. He is current an associate Researcher in Center for Strategic Evaluation Consultation, Academy of Military China. His main research interest is in file systems.

Si Zheng received his Ph.D degrees from the College of Computer Science and Technology at National University of Defense Technology, Changsha, China, in 2014. He is currently a Senior Engineer at the National University of Defense Technology, with his main research fields being Artificial Intelligence and foundational software.

Xiangke Liao graduated from the Department of Computer Science, Tsinghua University in 1985, and obtained his Master's degree in Computer Science from the National University of Defense Technology in 1988. In 2009, he became the Dean of the School of Computer Science, National University of Defense Technology. He was elected as an Academician of the Chinese Academy of Engineering in 2015. He has been engaged in research on high-performance computer systems software and general-purpose operating systems for many years.