The Design and Implementation of a Capacity-Variant Storage System

Ziyang Jiao and Xiangqun Zhang, *Syracuse University;*
Hojin Shin and Jongmoo Choi, *Dankook University;* Bryan S. Kim, *Syracuse University*

https://www.usenix.org/conference/fast24/presentation/jiao

This paper is included in the Proceedings of the
22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings
of the 22nd USENIX Conference on
File and Storage Technologies
is sponsored by

The Design and Implementation of a Capacity-Variant Storage System

Ziyang Jiao
Syracuse University

Xiangqun Zhang
Syracuse University

Hojin Shin
Dankook University

Jongmoo Choi
Dankook University

Bryan S. Kim
Syracuse University

Abstract

We present the design and implementation of a capacity-variant storage system (CVSS) for flash-based solid-state drives (SSDs). CVSS aims to maintain high performance throughout the lifetime of an SSD by allowing storage capacity to gracefully reduce over time, thus preventing fail-slow symptoms. The CVSS comprises three key components: (1) CV-SSD, an SSD that minimizes write amplification and gracefully reduces its exported capacity with age; (2) CV-FS, a log-structured file system for elastic logical partition; and (3) CV-manager, a user-level program that orchestrates system components based on the state of the storage system. We demonstrate the effectiveness of CVSS with synthetic and real workloads, and show its significant improvements in latency, throughput, and lifetime compared to a fixed-capacity storage system. Specifically, under real workloads, CVSS reduces the latency, improves the throughput, and extends the lifetime by 8–53%, 49–316%, and 268–327%, respectively.

1 Introduction

Fail-slow symptoms where components continue to function but experience degraded performance [16, 52] have recently gained significant attention for flash memory-based solid-state drives (SSDs) [40, 41, 66]. In SSDs, such degradation is often caused by the SSD-internal logic's attempts to correct errors [3, 16, 44, 50]. Recent studies have demonstrated that fail-slow drives can cause latency spikes of up to 3.65× [40], and since flash memory's reliability continues to deteriorate over time [25, 40, 66], we expect the impact of fail-slow symptoms on overall system performance to increase.

Figure 1 demonstrates a steady performance degradation for a real enterprise-grade SSD. We age the SSD through random writes by writing about 100 terabytes of data each day, and during morning hours when no other jobs are running, we measure the throughput of the read-only I/O, both sequential and random reads. As shown in Figure 1, the performance of the SSD degrades as the SSD wears out, at a rate of 4.2% and 4.3% of the initial performance for each petabyte written, for random reads and sequential reads, respectively. It is unlikely that the throughput drop is due to garbage collection as (1) this was measured daily over months, and (2) only reads are issued during measurement. By the end, writing a total of 9 petabytes of data to the SSD decreased the throughput by 37% for random reads and 38% for sequential reads.

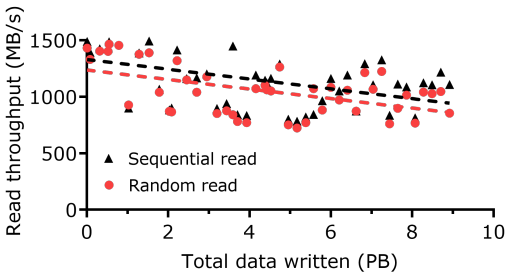


Figure 1: SSD performance degradation due to wear-out. The dashed line represents the linear regression of the daily data points. The throughput decreases by 37% for random reads and 38% for sequential reads after 9 petabytes of data writes.

To address this problem, we start with two key observations. First, flash memory, when it eventually fails, does so in a fail-partial manner. More specifically, an SSD's failure unit is an individual flash memory block [3, 44, 50], and the SSD-internal wear leveling algorithms are artifacts to emulate a hard disk drive-like fail-stop behavior [25, 31]. Second, an SSD has no other choice but to trade performance as flash memory's reliability deteriorates, because a storage device's capacity remains fixed and unchanged from its newly installed state until its retirement. SSD's internal data re-reads [4, 5, 42, 53] or preventive re-writes [6, 18] are such choices that lead to fail-slow symptoms [30, 31].

Based on the two key observations, we propose a capacity-variant storage system (CVSS) that maintains high performance even as SSD reliability deteriorates. In CVSS, the logical capacity of an SSD is not fixed; instead, it gracefully reduces the number of exported blocks below the original

capacity by mapping out error-prone blocks that would exhibit fail-slow behavior and hiding them from the host. This approach is enabled by the SSD's ability to update data out-of-place. Surprisingly, we find that maintaining a fixed-capacity interface comes at a heavy cost, and reducing capacity counterintuitively extends the lifetime of the device. Our experiments show that, compared to traditional storage systems, the capacity-variant approach of CVSS outperforms by 49–316% and outlasts by 268–327% under real-world workloads.

We enable capacity variance by designing kernel-level, device-level, and user-level components. The first component is a file system (CV-FS) that dynamically tunes the logical partition size based on the aged state of the storage device. CV-FS is designed to reduce capacity in an online, fine-grained manner and carefully manage user data to avoid data loss. The device-level component, CV-SSD, maintains its performance and reliability by mapping out aged and poor-performing blocks. Without needing to maintain fixed capacity, CV-SSD simplifies flash management firmware, avoids fail-slow symptoms, and extends its lifetime. Lastly, the user-level component, CV-manager, provides necessary interfaces to the host for capacity variance. Users can set performance and reliability requirements for the device through commands, and the CV-manager then adaptively orchestrates CV-FS and the underlying CV-SSD.

The contributions of this paper are as follows.

- We present the design of a capacity-variant storage system that relaxes the fixed-capacity abstraction of the storage device. Our design consists of user-level, kernel-level, and device-level components that collectively allow the system to maintain performance and extend its lifetime. (§ 3)
- We develop a framework that allows for a full-stack study on fail-slow symptoms in SSDs over a long time, from start to failure. This framework provides a comprehensive model of SSD internals and aging behavior over the entire lifetime of SSDs[1]. (§ 4)
- We evaluate and quantitatively demonstrate the benefits of capacity variance using a set of synthetic and real-world I/O workloads throughout the SSD's *entire* lifetime. Capacity variance avoids the fail-slow symptoms and can significantly extend the SSD's lifetime. (§ 5)

2 Background and Motivation

We first show the increasing trend of flash memory errors in SSDs and describe how flash cells wear out. We then explain how the current storage system abstraction exacerbates reliability-related performance degradation, and summarize prior work for addressing these fail-slow symptoms.

Flash memory errors and wear-out. The rapid increase in NAND flash memory density has come at the cost of reduced reliability and exacerbated fail-slow symptoms. Figure 2 shows the reported flash raw bit error rates (RBERs) in recent publications [3, 4, 14, 30, 42, 55, 57, 65], and this trend indicates that flash memory errors are already a common case.

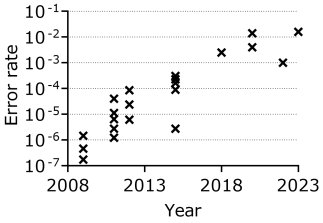


Figure 2: Flash memory error rates have increased significantly over the past years.

One of the significant flash memory error mechanisms is wear-out, where flash cells are gradually damaged with repeated programs and erases [44, 50]. Because wear-outs are irreversible, once a flash block reaches its endurance limit or returns an operation failure, it is marked as bad by the SSD-internal flash translation layer (FTL) and taken out of circulation. To replace these unusable blocks, SSDs are often over-provisioned with more physical capacity than the logically exported capacity.

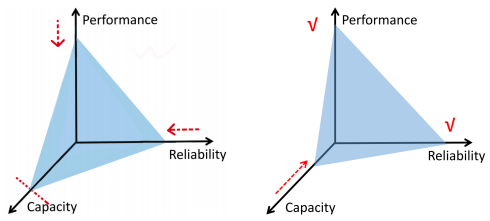
SSD's wear-outs are caused not only by write I/Os, but also by SSD-internal management such as garbage collection, reliability management, and wear leveling (WL). Although much of the literature has emphasized the role of garbage collection in the SSD's internal writes, studies have revealed that SSD's reliability management and WL also significantly impact the lifetime [25, 27, 30, 43]. WL, in particular, is revealed to be far from perfect, wearing out some of the blocks 6× faster [43] and often leads to counterintuitive acceleration of wear-outs, increasing the write amplification factor as high as 11.49 [25].

Fixed capacity abstraction. Unfortunately, the current storage system abstraction of fixed capacity requires SSDs to implement wear leveling (WL), even if it is imperfect and harmful [25, 43]. Specifically, with the fixed capacity abstraction, the device is not allowed to have part of its capacity fail (i.e., wear out) prematurely, and therefore the SSD has to perform wear leveling to ensure that most, if not all, of its capacity is wearing out at roughly the same rate. If the SSD cannot maintain its original exported capacity when too many blocks become bad, then the entire storage device becomes unusable [50]. This is despite the fact that the SSD internally has a level of indirection and abstracts the physical capacity.

However, the file system provides a file abstraction to the user-level applications, and this abstraction hides the notion of capacity. While utility programs such as `df` and `du` report the storage capacity utilization, file operations such as `open()`, `close()`, `read()`, and `write()` do not expose capacity directly. Instead, the file system manages the storage capacity using persistent data structures such as superblock and allocation maps to track the utilization of the SSD.

The fixed capacity abstraction used between the file system and storage devices necessitates the implementation of WL

[1] Our framework and extensions are available at https://github.com/ZiyangJiao/FAST24_CVSS_FEMU



(a) TrSS (b) CVSS

Figure 3: Comparison between the traditional fixed-capacity storage system (TrSS) and capacity-variant storage system (CVSS). For TrSS (Figure 3a), the performance and reliability degrade as the device ages to maintain a fixed capacity; for CVSS (Figure 3b), the performance and reliability are maintained by trading capacity.

on physical flash memory blocks. However, WL leads to an overall increase in wear on the SSD, resulting in a significantly higher error rate as all the blocks age. This, in turn, manifests into fail-slow symptoms in SSDs.

Fail-slow symptoms. Fail-slow symptoms are caused by the SSD's effort to correct errors [4, 5, 42, 53] and prevent the accumulation of errors [6, 18]. Because SSDs are commonly used as the performance tier in storage systems where the identification and removal of ill-performing drives are critical, fail-slow symptoms in SSDs have gained significant attention recently. Prior research in this area can be categorized into three types. The first group focuses on developing machine learning (ML) models to quickly identify SSDs experiencing fail-slow symptoms [7, 22, 61, 66, 67]. Various models, including neural networks [22], autoencoders [7], LSTM [67], feature ranking [61], and random forest [66], have been explored with varying accuracy and efficacy. The second group aims to isolate and remove ailing drives using mechanisms deployed in large-scale systems, identified through ML [40] or system monitoring [21, 52]. The third group proposes modifications to the interface to reject slow I/O and send hedging requests to a different node [20] or drive [38].

Unfortunately, ML-based learning of SSD failures requires an immense number of data points, is often expensive to train, and is only available in large-scale systems [40, 66]. Furthermore, as SSDs evolve and new error mechanisms emerge (e.g., lateral charge spreading [36] and vertical and horizontal variability [56]), older ML models become obsolete, making it difficult to reap the benefits of fail-slow prediction. Most critically, these prior approaches only treat the symptoms and fail to consider the underlying cause: the flash error mechanism.

3 Design for Capacity Variance

The high-level design principle behind the capacity-variant system is illustrated in Figure 3. This system relaxes the fixed-capacity abstraction of the storage device and enables a better tradeoff between capacity, performance, and reliability. The traditional fixed-capacity interface, which was designed

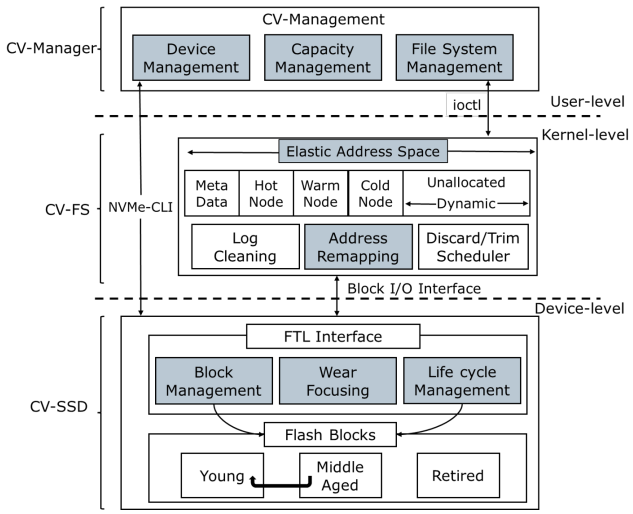
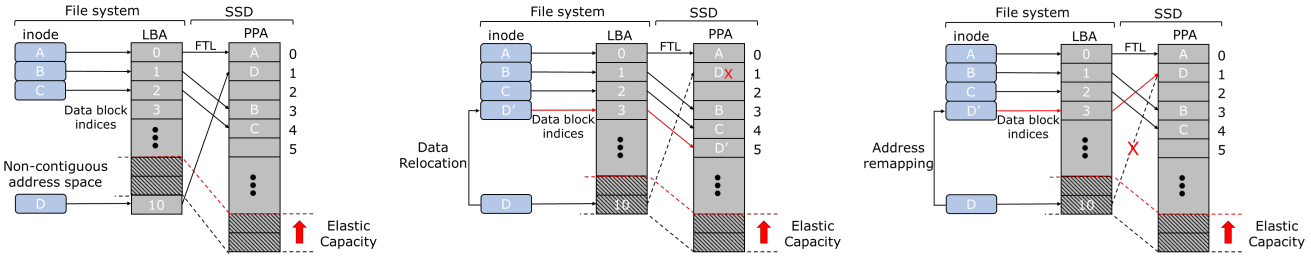


Figure 4: An overview of the capacity-variant system: (1) CV-FS exports an elastic logical space based on CV-SSD's aged state; (2) CV-SSD retires error-prone blocks to maintain device performance and reliability; and (3) CV-manager provides user-level interfaces and orchestrates CV-SSD and CV-FS. The highlighted components are discussed in detail.

for HDDs, assumes a fail-stop behavior where all storage components either work or fail at the same time. However, this assumption is not accurate for SSDs since flash memory blocks are the basic unit of failure, and it is the responsibility of the FTL to map out failed, bad, and aged blocks [31, 50].

By allowing a flexible capacity-variant interface, an SSD can gracefully reduce its exported capacity, and the storage system as a whole would reap the following three benefits.

- **Performant SSD even when aged.** An SSD can avoid fail-slow symptoms by gracefully reducing its number of exported blocks. Error management techniques such as data re-reads [4, 5, 42, 53] and data re-writes [6, 18] would be performed less frequently as blocks with high error rates can be mapped out earlier. This, in turn, reduces the tail latency and lowers the write amplification, making it easier to achieve consistent storage performance.
- **Extended lifetime for SSD-based storage.** An SSD's lifetime is typically defined with a conditional warranty restriction under *DWPD (drive writes per day)*, *TBW (terabytes written)*, or *GB/day (gigabytes written per day)* [58]. With the fixed capacity abstraction, the SSD reaches the end of its lifetime when the physical capacity becomes less than the original logical capacity. Instead, with capacity variance, the lifetime of an SSD would be extended significantly, as it would be defined by the amount of data stored in the SSD, not by the initial logical capacity.
- **Streamlined SSD design.** By adopting the approach of allowing the logical capacity to drop below the initial value, SSD vendors can design smaller and more efficient error correction hardware and their SSD-internal firmware: There



(a) Non-contiguous address space (b) Data relocation (c) Address remapping

Figure 5: Design options for capacity variance. In Figure 5a, the FS internally maps out a range of free LBA from the user, causing address space fragmentation. In Figure 5b, the data block is physically relocated to lower LBA. This approach maintains the contiguity of the entire address space but exerts additional write pressure on the SSD. Lastly, in Figure 5c, the data block can be logically remapped to lower LBA. This approach incurs negligible system overhead by introducing a special SSD command to associate data with a new LBA.

is no need to overprovision the SSD's error handling logic or to ensure that all blocks wear out evenly.

Figure 4 illustrates the main components of a capacity-variant system. Enabling the capacity variance feature is achieved by designing the following three components: (1) CV-FS, a log-structured file system for supporting elastic logical partition; (2) CV-SSD, a capacity-variant SSD that maintains device performance and reliability by effectively mapping out aged blocks; and (3) CV-manager, a capacity management scheme that provides the interface for adaptively managing the capacity-variant system.

3.1 Capacity-Variant FS

The higher-level storage interfaces, such as the POSIX file system interface, allow multiple applications to access storage using common file semantics. However, to support capacity variance, the file system needs to be modified. In this section, we discuss the feasibility of an elastic logical capacity based on existing storage abstractions and then investigate different approaches for supporting capacity variance, Lastly, we describe our new interface for capacity-variant storage systems based on the selected approach.

3.1.1 Feasibility of Elastic Capacity

Current file systems assume that the capacity of the storage device does not change and tightly couple the size of the logical partition to the size of the associated storage device. To overcome this limitation, the CV-FS file system declares the entire address space for use at first and then dynamically adjusts the declared space as the storage device ages in an online manner. This is achieved by defining a variable logical partition that is independent of the physical storage capacity.

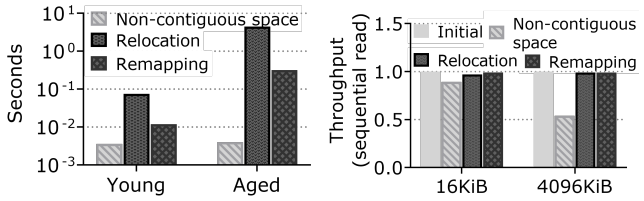
Thankfully, this transition is feasible for three reasons. First, the TRIM [47] command, which is widely supported by interface standards such as NVMe, enables the file system to explicitly declare the data that is no longer in use. This allows the SSD to discard the data safely, making it possible to reduce the exported capacity gracefully. Second, modern file systems can safely compact their content so that the data in use are contiguous in the logical address space. Log-structured file systems [54] support this more readily, but file system defragmentation [59] techniques can be used to achieve the same effect in in-place update file systems. Lastly, the file abstraction to the applications hides the remaining space left on storage. A file is simply a sequence of bytes, and file system metadata such as utilization and remaining space is readily available to the system administrator.

3.1.2 File System Designs for Capacity Variance

Shrinking the logical capacity of a file system can be a complex procedure that may result in data loss if not done carefully [31]. Most importantly, any valid data within the to-be-shrunk space must be relocated and the process must be coordinated with underlying storage accordingly. Moreover, to ensure users do not need to unmount and remount the device, the logical capacity should be reduced in an online manner, and the time it takes to reduce capacity should be minimal with low overhead.

Figure 5 depicts three approaches to performing online address space reduction: (1) through a non-contiguous address space; (2) through data relocation; and (3) through address remapping. We describe each approach and our rationale for choosing the address remapping (Figure 5c).

- **Non-contiguous address space** (Figure 5a). The file system internally decouples the space exported to users from the LBA. When logical capacity should be reduced, the file system identifies an available range of free space from the end of the logical partition and then restricts the user from using it, for example, by marking that as allocated. With this approach, the adjustment of logical capacity can be efficiently achieved with minimal upfront costs, as the primary task involved is allocating the readily available free space. However, this approach increases the file system cleaning overhead and fragments the file system address



(a) Time elapsed for shrinking 1 GiB logical capacity.

(b) Performance comparison before and after shrinking 60 GiB logical capacity.

Figure 6: Performance results for three capacity variance approaches. The address remapping approach introduces lower overhead (Figure 6a) and does not incur fragmentation after shrinking the address space (Figure 6b).

space. Due to the negative effect of address fragmentation [12, 13, 19, 24], we avoid this approach despite the lowest upfront cost.

- **Data relocation** (Figure 5b). Similar to segment cleaning or defragmentation, the file system relocates valid data within the to-be-shrunk space to a lower LBA region before reducing the capacity from the higher end of the logical partition. This approach maintains the contiguity of the entire address space. Nevertheless, it is essential to note that data relocation exerts additional write pressure on the SSD and the overhead is proportional to the amount of valid data copied. Moreover, user requests are potentially stalled during the relocation process.
- **Address remapping** (Figure 5c). Data is relocated logically at the file system level without data relocation at the SSD level by taking advantage of the already existing SSD-internal mapping table [49, 68]. While this approach necessitates the introduction of a new SSD command to associate data with a new LBA, it effectively mitigates address space fragmentation and incurs negligible system overhead, as no physical data is actually written.

We implement the three approaches above on F2FS and measure the elapsed time for reducing capacity by 1 GiB. The reported results represent an average of 60 measurements. On average, each measurement resulted in the relocation or remapping of 0.5 GiB of data for the aged file system case and 0.05 GiB of data for the young case. We further compare the performance under the sequential read workload with two I/O sizes (i.e., 16 KiB and 4096 KiB) before and after capacity is reduced. As depicted in Figure 6, the elapsed time required to shrink 1 GiB of logical space on an aged file system is 0.317 seconds when employing the address remapping approach. In contrast, the data relocation approach takes approximately 4.5 seconds. Notably, while the non-contiguous address space approach only takes 0.004 seconds, it exhibits significant performance degradation after the capacity reduction, for example, 13% for 16 KiB read and 50% for 4096 KiB read, due to increased fragmentation. We next present

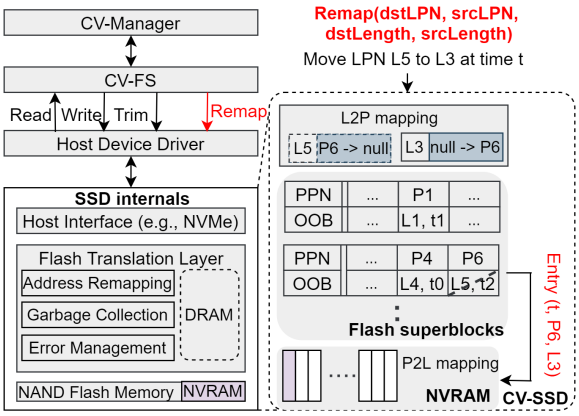


Figure 7: The `REMAP` command workflow for capacity variance: data in the range between `srcLPN` and `srcLPN + srcLength -1` are remapped to logical address starting from `dstLPN`. The third argument, `dstLength`, is optionally used for the file system to ensure I/O alignment.

the design details of the proposed remapping interface and the capacity reduction process with that.

3.1.3 Interface Changes for Capacity Variance

To integrate the address remapping approach into CV-FS, we revise the interface proposed by prior works [49, 68], `REMAP(dstLPN, srcLPN, dstLength, srcLength)`, and tailor it for capacity-variant storage systems. Our modified command enables file systems to safely shrink the logical capacity with minimal overhead by remapping valid data from their old LPNs to new LPNs without the need for actual data rewriting [49, 68]. We extend the current NVMe interface to include remap as a vendor-unique command.

Figure 7 shows an example of the remap command used for shrinking capacity. Assuming the file system address space ranges from LBA0 – LBA47 at the beginning (i.e., LPN0 – LPN5 with 512 bytes sector and 4 KiB page size) and LPN5 is mapped to PPN6 within the device. At time *t*, CV-FS initiates the capacity reduction and identifies that LBA40 – LBA47 (or LPN5) contains valid data. It then issues the remap command to move LPN5 data to LPN3 (i.e., `remap(LPN3, LPN5, 1, 1)`. Upon receiving the remap command, the FTL first finds the PPN associated with LPN5 (PPN6 in our case) and updates the logical-to-physical (L2P) mapping of L3 to PPN6. Finally, the old L2P mapping of L5 is invalidated, and the new physical-to-logical (P2L) mapping of PPN6 is recorded in the NVRAM of the SSD. Once the to-be-shrunk space is free, the file system states are validated and a new logical capacity size is updated.

In particular, the required size for NVRAM is small (for example, 1 MiB for a 1 TiB drive as suggested by the prior work [49]), as it is only used to maintain a log of the remapping metadata. Assuming the SSD capacity and page size are 1TB and 4KB, respectively, a single remapping entry requires

no more than 8B space. The 1 MiB NVRAM would be sufficient to hold entries for 512 MiB remapped data during a capacity reduction event. Between capacity reduction events, the reclamation of a flash block/page will cause a passive recycle on the associated remapping entries. However, in cases where a larger buffer is needed, the log can perform an active cleaning or write part of the mappings to flash because the space allocated for internal metadata will be conserved with a smaller device capacity [17]. Alternatively, the need for NVRAM can be eliminated by switching to the data relocation method, but at the cost of a higher overhead for logical capacity adjustment.

As a result, this new interface does not require taking the file system and device offline to adjust address space since data are managed logically. Moreover, it does not complicate the existing file system consistency management scheme. Similar to other events such as `discard`, the file system periodically performs checkpoints to provide a consistent state. The crash consistency is examined by manually crashing the system after initiating the remapping command and CrashMonkey [46] with its pre-defined workloads [45].

3.2 Capacity-Variant SSD

In this section, we outline design decisions and their leading benefits for building a capacity-variant SSD. We first discuss the necessity to forgo wear leveling in CV-SSD, and then describe its block management and life cycle management for extending lifetime and maintaining performance. Lastly, we introduce a degraded mode to handle the case where the remaining physical capacity becomes low.

Note that blocks in this subsection refer to physical flash memory blocks, different from the logical blocks managed by the file system. Furthermore, the flash memory blocks are grouped and managed as superblocks (again, different from the file system's superblock) to exploit the SSD's parallelism.

3.2.1 Wear Focusing

The goal of a capacity-variant SSD is to keep as much flash as possible at peak performance and mitigate the impact of underperforming and aged blocks. A capacity-variant SSD would maintain both performance and reliability by gracefully reducing its exported capacity so aged blocks can be mapped out earlier. Therefore, a capacity-variant SSD does not perform wear leveling (WL), as it degrades all of the blocks over time. WL is an artifact designed to maintain an illusion of a fixed-capacity device wherein its underlying storage components (i.e., flash memory blocks) either all work or fail, opposing our goal of allowing partial failure.

Moreover, static WL [8, 9, 15] incurs additional write amplification due to data relocation within an SSD. Dynamic WL [10, 11], on the other hand, typically combines with SSD internal tasks such as garbage collection, reducing the overall cleaning efficiency as its victim selection considers both the valid ratio and wear state. A recent large-scale field study on millions of SSDs reveals that the WL techniques in modern SSDs present limited effectiveness [43] and an analysis study demonstrates that WL algorithms can even exhibit unintended behaviors by misjudging the lifetime of data in a block [25]. Such counter-productive results are avoided by forgoing WL and adopting capacity variance.

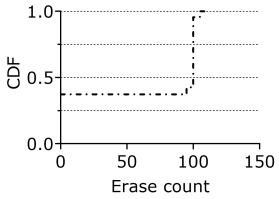


Figure 8: The wear distribution for a 256 GiB SSD under 100 iterations of MS-DTRS workload [29]. Traditional GC and block allocation policies cause a sudden capacity loss as too many blocks are equally aged.

3.2.2 Block Management

A capacity-variant SSD exploits the characteristics of flash memory blocks to extend its lifetime and meet different performance and reliability requirements. Flash memory blocks in SSDs wear out at different rates and are marked as bad blocks by the bad block manager when they are no longer usable [26, 50]. This means that the physical capacity of the SSD naturally reduces over time, and for a fixed-capacity SSD, the entire storage device is considered to have reached the end of its life when the number of bad blocks exceeds its reserved space. On the other hand, the capacity-variant SSD's lifetime is defined by the amount of data stored in the SSD, rather than the initial logical capacity, making it a more reliable and efficient option.

The fail-slow symptoms and performance degradation in SSDs are caused by aged blocks with high error rates [4, 5, 42, 53]. Traditional SSDs consider blocks as either good or bad and such coarse-grained management fails to meet different performance and reliability requirements. On the other hand, the capacity-variant SSD defines three states of blocks: young, middle-aged, and retired, based on their operational characteristics. Young blocks have a relatively low erase count and a low RBER, while middle-aged blocks have higher errors and require advanced techniques to recover data. Retired blocks that are worn out or have a higher RBER than the configured threshold (5% by default) are excluded from storing data.

This block management scheme allows the capacity-variant SSD to map out underperforming and unreliable blocks earlier, effectively trading capacity for performance and reliability. In general, blocks start from a young state and transition to middle-aged and retired states. However, a block can also transition from a middle-aged state back to a young state since transient errors (i.e., retention and disturbance) are reset once the block is erased [30].

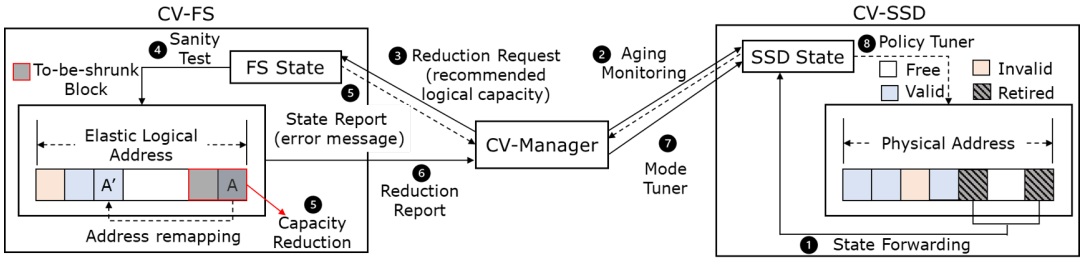


Figure 9: CV-manager design diagram. CV-manager monitors CV-SSD's aged state (Steps 1 and 2) and provides a recommended logical capacity to CV-FS (Step 3). After capacity reduction (Steps 4–6), CV-manager notifies CV-SSD (Step 7). The $CV_{degraded}$ mode will be triggered if the reduction fails (Step 8).

3.2.3 Life Cycle Management

A capacity-variant SSD requires wear focusing to mitigate the impact of aged flash memory blocks. However, simply avoiding wear leveling is insufficient as there are two processes affecting the life cycle of a flash memory block: **block allocation** and **garbage collection (GC)**. Traditional policies such as youngest-block-first for allocation and cost-benefit for GC work well on a traditional SSD, but are not suitable for CV-SSDs, since they aim to achieve a uniform wear state among blocks. Implementing these policies can cause a large number of blocks with the same erase count to map out simultaneously, leading to excessive capacity loss, and the device may suddenly fail. Figure 8 demonstrates this issue, where over 60% of the blocks aggregate to a particular wear state. Excessive capacity loss can increase the write amplification factor (WAF), particularly when the device utilization rate is high.

Allocation policy. In order to make wear accumulate in a small subset of blocks and allow capacity to shrink gradually, CV-SSD will prioritize middle-aged blocks to accommodate host writes and young blocks for GC writes. Since retired blocks are not used, there are four scenarios when considering data characteristics.

I. Write-intensive data are written to a middle-aged block

II. Write-intensive data are written to a young block

III. Read-intensive data are written to a middle-aged block

IV. Read-intensive data are written to a young block

Type I and type IV are ideal cases as they help to converge the wear among blocks without affecting the performance. Type II will also not affect the performance when data are fetched by the host because of the low RBER of young blocks. Moreover, with CV-SSD's allocation policy, such write-intensive data are inevitably re-written by the file system to the middle-aged blocks and the type II blocks will be GC-ed due to their low valid ratio. This type of scenario also happens under the early stages of CV-SSD, in which most blocks are young. Lastly, type III is the case where we need to pay more attention: read-intensive data should be stored in young blocks; otherwise expensive error correction techniques are triggered more often.

Garbage collection. We modify the garbage collection policy to consider (in)valid ratio, aging status, and data characteristics to handle type III cases. The block with the highest score will be selected as the victim based on the following formula:

$$\begin{aligned} Victim\ score &= W_{invalidity} \cdot invalid\ ratio \\ &+ W_{aging} \cdot aging\ ratio \\ &+ W_{read} \cdot read\ ratio \\ invalid\ ratio &= \frac{\#\ of\ invalid\ pages}{\#\ of\ valid\ pages + \#\ of\ invalid\ pages}, \\ aging\ ratio &= \frac{erase\ count}{endurance}, \\ read\ ratio &= \frac{\#\ of\ host\ read\ designated\ to\ the\ current\ block}{maximum\ host\ read\ among\ unretired\ blocks}. \end{aligned} \tag{1}$$

$W_{invalidity}$, W_{aging}, and W_{read} are weights to balance WAF, the aggressiveness of wear focusing, and the sensitivity of preventing type III scenarios, respectively. With that, read-intensive data stored in aged blocks are relocated by GC. Considering the read ratio could potentially affect the garbage collection efficiency. To avoid low GC efficiency, we set $W_{invalidity} = 0.4$, $W_{aging} = 0.3$, and $W_{read} = 0.3$, and their sensitivity analysis is shown in § 5.4.3. Increasing W_{read} is unfavorable not only because of adverse effects on WAF but also due to introducing unnecessary data movement. For example, a middle-aged block containing many valid pages but experiencing only a minimal number of reads is selected as the victim.

3.2.4 Degraded Mode

During normal conditions, CV-SSD intentionally unevens the wear state among blocks. As error-prone blocks retire, the physical capacity decreases gradually and performance is maintained. However, the physical capacity could decrease to a level where it will be insufficient to maintain current user data. Moreover, it can also cause high garbage collection overhead. In this case, $CV_{degraded}$ mode will be triggered and CV-SSD will slow down the further capacity loss. It is noteworthy that the triggering of degraded mode indicates a low remaining capacity to trade for performance, and storage administrators can gradually upgrade storage systems.

In particular, the $CV_{degraded}$ mode is triggered under two conditions: (1) when the effective over-provisioning

(EOP), calculated as $EOP = (physical\ capacity - utilization)/utilization$, falls below the factory-set over-provisioning (OP), or (2) when the remaining physical capacity is less than a user-defined watermark.

Once $CV_{degraded}$ mode is set, GC only considers WAF and aging to slow down further capacity loss. This mode allows young blocks to be cleaned with a relatively higher valid ratio than aged blocks. Specifically, young blocks with a high invalid ratio are optimal candidates. Moreover, middle-aged blocks are used to accommodate GC-ed data, and young blocks are allocated for host writes. As a result, blocks are used more evenly than in the initial stage and a particular amount of physical capacity is maintained for the user. When EOP becomes greater than OP if the host decides to move or delete some data, $CV_{degraded}$ will be reset by CV-manager.

3.3 Capacity-Variant Manager

To improve usability, CV-manager is responsible for automatically managing the capacity of the whole storage system. As illustrated in Figure 9, CV-manager monitors the aged state of the underlying storage device and provides a recommended logical partition size to the kernel.

Specifically, when CV-SSD maps out blocks and its physical capacity is reduced, CV-manager will get notified (Steps 1 and 2). The CV-manager figures out a recommended logical capacity by checking the current bad capacity within the device and issues capacity reduction requests to CV-FS through a system call (Step 3). Upon request, CV-FS performs a sanity test. If the file system checkpoint functionality is disabled or the file system is not ready to shrink (i.e., frozen or read-only), the reduction will not continue (Step 4). Otherwise, CV-FS starts shrinking capacity as described in § 3.1.3 and returns the execution result (Steps 5 and 6). Lastly, CV-manager notifies CV-SSD whether logical capacity is reduced properly or not. If the reduction fails, the $CV_{degraded}$ is activated to slow down further capacity loss (Steps 7 and 8).

For user-level capacity management, CV-manager provides necessary interfaces for users to explicitly initiate capacity reduction and set performance and reliability requirements for the device. The CV-SSD would retire blocks based on the host requirement. Similar to the read recovery level (RRL) command [47] in the NVMe specification that limits the number of read retry operations for a read request, this configurable attribute limits the maximum amount of recovery applied to a request and thus balances the performance.

4 Implementation

The capacity-variant file system (CV-FS) is implemented upon the Linux kernel v5.15. CV-FS uses F2FS [35] as the baseline file system due to its virtue of being a log-structure file system. We modify both CVSS and TrSS to employ a more aggressive discard policy than the baseline F2FS (i.e., 50ms interval if candidates exist and 10s max interval if no candidates) for better SSD garbage collection efficiency [32] (also shown in § 5.2.3).

To implement the `remap` command, we extend the block I/O layer. A new I/O request operation `REQ_OP_REMAP` is added to expose the remap command to the CV-FS. New attributes including `bio->bi_iter.bi_source_sector` and `bio->bi_iter.bi_source_size` are introduced in `bvec_iter`, which corresponds to the second and last parameter of the remap command. Functions related to `bio` splitting/merging procedure (e.g., `__blk_queue_split`) are modified to maintain added attributes (mainly in `/block/blk-merge.c`). Additionally, new `nvme_opcode` and related functions are added to support the remap command at the NVMe driver layer (mainly in `/block/blk-mq.c` and `/drivers/nvme/host/core.c`).

The capacity-variant SSD is built on top of the FEMU [37]. SSD reliability enhancement techniques such as ECC and read retry ensure data integrity. To implement the error model, we use the additive power-law model proposed in prior works [30, 39, 44] that considers wear, retention loss, and disturbance to quantify RBER, as shown in the following equation:

$$\begin{aligned} &RBER(cycles, time, reads) \\ &= \varepsilon + \alpha \cdot cycles^{k} && (wear) \\ &\quad + \beta \cdot cycles^{m} \cdot time^{n} && (retention) \\ &\quad + \gamma \cdot cycles^{p} \cdot reads^{q} && (disturbance) \end{aligned} \tag{2}$$

The parameters used are particular to a real 2018 TLC flash chip [30], and the device internally keeps track of cycles, time, and reads for each block. During a read operation, read retry is applied if the error exceeds the ECC strength. We consider each read retry will lower the error rate by half [30, 53] and the maximum amount of recovery is limited for a single read retry so that blocks have more fine-grained error states.

We modify five major software components to support capacity variance.

- We make changes to the Linux kernel v5.15 to provide an `ioctl`-based user-space API supporting logical partition reduction. Users can specify the shrinking size and issue capacity reduction commands through this API.
- We modify the F2FS to handle address remapping triggered by capacity variance and revise its discard scheme.
- We extend the `f2fs-tool` (f2fs format utility) to support the CV-specific functionalities, such as initializing a variable logical partition and updating the attributes that control discard policies.
- We implement CV-SSD mode in FEMU, adding flash reliability enhancement techniques, error models, wear leveling, bad block management, and device lifetime features.
- We modify NVMe device driver and introduce new commands to `NVMe-Cli` [48], to support capacity variance. The

Table 1: System configurations. Wear leveling (PWL [9]) and youngest block first allocation are used for traditional SSDs.

PC platform			
Parameter	Value	Parameter	Value
CPU name	Intel Xeon 4208	Frequency	2.10GHz
Number of cores	32	Memory	1TiB
Kernel	Ubuntu v5.15	ISA	X86_64
FEMU			
Parameter	Value	Parameter	Value
Channels	8	Physical capacity	128 GiB
Luns per channel	8	Logical capacity	120 GiB
Planes per lun	1	Over-provisioning	7.37%
Blocks per plane	512	Garbage collection	Greedy
Pages per block	1024	Program latency	500 μs
Page size	4 KiB	Read latency	50 μs
Superblock size	256 MiB	Erase latency	5 ms
Endurance	300	Wear leveling	PWL [9]
ECC strength	50 bits	Block allocation	Youngest first

SMART [47] command is also extended to export more device statistics for capacity management.

5 Evaluation of Capacity Variance

We first describe our experimental setup and methodology, then present our evaluation results and demonstrate the effectiveness of capacity variance.

5.1 Experimental Setup and Methodology

Table 1 outlines the system configurations for our evaluation. For the traditional SSD, an adaptive WL, PWL [9], is used to even the wear among blocks. The error correction code (ECC) for both Tr-SSD and CV-SSD is configured to tolerate up to 50-bit errors per 4 KiB data, and errors beyond the correction strength are subsequently handled by read retry. We use 17 different workloads in our evaluation: (1) 4 FIO [23] workloads (Zipfian and random, each with two different utilization); (2) 3 Filebench [60] workloads; (3) 2 YCSB workloads [2] (YCSB-A and YCSB-F); and (4) 8 key-value traces from Twitter [64].

We compare CVSS with three different techniques: (1) TrSS, a traditional storage system with vanilla F2FS plus a fixed-capacity SSD; (2) AutoStream [63]; (3) ttFlash [62]. The evaluation comparisons are selected based on their broader applicability and implementation simplicity of the multi-stream interface (represented by AutoStream [63]) and the fast-fail mechanism (represented by ttFlash [62]). These approaches align with more general and widely used methods such as PCStream [33], LinnOS [22], and IODA [38]. Specifically, AutoStream [63] uses the multi-stream interface [28] and automatically assigns a stream ID to the data based on the I/O access pattern. The SSD then places data accordingly based on the assigned ID to reduce write amplification and thus, improve performance. On the other hand, ttFlash [62] reduces the tail latency of SSDs by utilizing a redundancy scheme (similar to RAID) to reconstruct data when blocked by GC. Since the original ttFlash is implemented on a simulator, we implement its logic in FEMU for a fair comparison.

To perform a more realistic evaluation, it is necessary to reach an aged FS and device state. Issuing workloads manually to the system is prohibitively expensive, as it takes years' worth of time. Moreover, this method lacks standardization and reproducibility, making the evaluation ineffective [1]. We extensively use aging frameworks in our evaluation. Prior to each experiment, we use impression [1] to generate a representative aged file system layout. After file system aging, the fast-forwardable SSD aging framework (FF-SSD) [26] is used to reach different aged states for SSD. The aging acceleration factor (AF) is strictly limited to 2 to maintain accuracy. Workloads will run until the underlying SSD fails.

We design the experiments with the following questions:

- Can CVSS maintain performance while the underlying storage device ages? (§ 5.2)
- Can CVSS extend the device lifetime under different performance requirements? (§ 5.3)
- What are the tradeoffs in CVSS design? (§ 5.4)

5.2 Performance Improvement

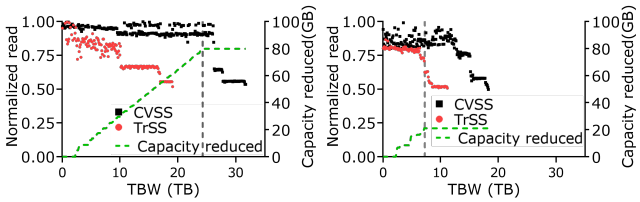
In this section, we evaluate the effectiveness of CVSS in maintaining performance and avoiding fail-slow symptoms under synthetic and real workloads.

5.2.1 FIO

We first examine the performance benefit of capacity variance under Zipfian workloads with two different workload sizes: 38GB (utilization of 30%) and 90GB (utilization of 70%). For this experiment, FIO continuously issues 16KB read and write requests to the device. We use the default setting of FIO and the read/write ratio is 0.5/0.5.

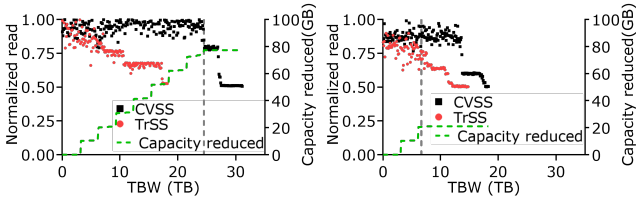
Zipfian. Figure 10 shows the read throughput under different aged states of TrSS and CVSS, in terms of terabytes written (TBW). We measure the performance until it drops below 50% of the initial value where no aging-related operations are performed. The green dotted line shows the amount of physical capacity that has been reduced within the CV-SSD and the straight vertical line represents the trigger of $CV_{degraded}$.

We observe that TrSS and CVSS behave similarly at first where both CV-SSD and Tr-SSD are relatively young. However, for TrSS, the read performance degrades gradually. As Tr-SSD gets aged, the amount of error corrected during each read operation increases and thus involves more expensive read retry processes. On the other hand, CV-SSD effectively trades the capacity for performance. The performance is maintained by excluding heavily aged blocks from use. Later, $CV_{degraded}$ is triggered to maintain a particular amount of capacity for the workloads. During this stage, blocks are used



(a) Zipfian (utilization 30%) (b) Zipfian (utilization 70%)

Figure 10: Read throughput under FIO Zipfian workloads. In CVSS, the performance is maintained by trading capacity. The straight vertical line represents the trigger of the $CV_{degraded}$ mode. After $CV_{degraded}$, the future capacity reduction is slowed down but the performance is compromised.



(a) Random (utilization 30%) (b) Random (utilization 70%)

Figure 11: Read throughput under FIO random workloads. CVSS delivers up to 0.6× (left) and 0.7× (right) higher performance compared to TrSS, under the same amount of host writes.

more evenly and the wear accumulates within the device. In this case, performance is traded for capacity in order to avoid data loss. However, even in this mode, CVSS delivers better performance compared to TrSS, thanks to the previous mapping out of most unreliable blocks. Overall, the read throughput of CVSS outperforms TrSS by up to 0.72× with the same amount of host writes.

Random. Figure 11 shows the measured read performance under random I/O. The configuration is similar to the previous case. As in-used blocks get aged, the read performance of TrSS degrades gradually and the fail-slow symptoms manifest. With the same amount of host writes, CVSS delivers a 0.6× and a 0.7× higher throughput at most than TrSS under the utilization of 30% and 70%, respectively.

Figure 13 compares the average write performance over the measurement. For Tr-SSD, when WL is initiated, data are relocated within the device, which decreases the throughput by 0.6×. Without WL, CV-SSD provides a more stable and better write performance than Tr-SSD. Overall, the write throughput of CVSS outperforms TrSS by 0.12× on average.

5.2.2 Filebench

We now use Filebench [60] to evaluate the capacity-variant system under file system metadata-heavy workloads. We use three pre-defined workloads in the benchmark, which exhibit differences in I/O patterns and fsync usage.

Figures 12a, 12b, and 12c show the CDF of operation latency under fileserver, netsfs, and varmail workloads throughout the devices' life. In particular, CVSS–normal represents the result before $CV_{degraded}$ is activated and CVSS represents the overall result. We use the default setting of Filebench, which measures the performance by running workloads for 60 seconds. Random writes are used to age CV-SSD and Tr-SSD. The measurement is performed after every 100GB of random data written until the device fails. The utilization for both TrSS and CVSS is 50%.

Compared to TrSS, CVSS reduces the average response time by 32% before the degraded mode is triggered and by 24% over the entire lifetime under netsfs workload, as shown in Figure 12b. The netsfs workload simulates the behavior of a network file server. It performs a more comprehensive set of operations such as application lunch, read-modify-write, file appending, and metadata retrieving, and thus reflects the state of the underlying devices more intuitively. Overall, CVSS reduces the average latency by 8% in the fileserver case (Figure 12a), and 10% in the varmail case (Figure 12c).

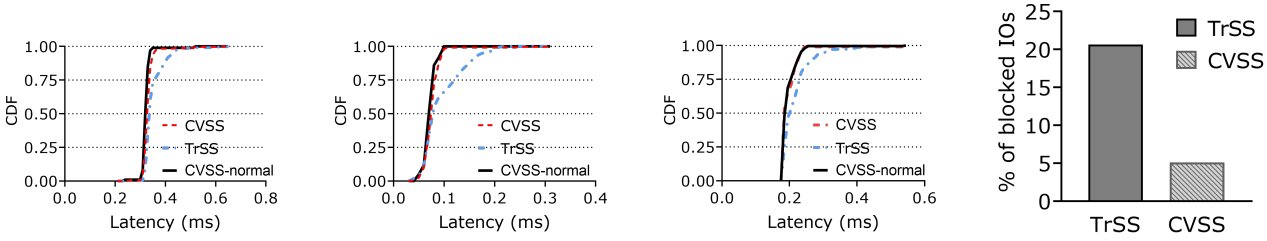
CV-SSD maps out blocks once their RBER exceeds 5% by default, while Tr-SSD only maps them out when their erase counts exceed the endurance limit, leading to more expensive error correction operations. The increased error correction operations not only affect the latency of the ongoing host request but also create backlogs in IO traffic. Figure 12d shows the percentage of host I/Os blocked by read retry operations measured inside FEMU under the varmail workload. In TrSS, more than 20% of I/O requests are delayed by SSD internal read retry, while it is no more than 5% in CVSS.

5.2.3 Twitter Traces

The previous sections examine CVSS using block I/O workloads and file system metadata-heavy workloads. In this section, we evaluate CVSS and compare it with AutoStream [63] and ttFlash [62] at the overall application level. We use a set of key-value traces from Twitter production [64]. The Twitter workload contains 36.7 GB worth of key-value pairs in total. We first load the key-values pairs and then start and keep feeding the traces to RocksDB until the underlying SSD fails.

Figure 14 compares the average KIOPS over the entire device lifetime. Overall, capacity variance improves the throughput by 0.49× – 3.16× compared to TrSS; on the other hand, AutoStream and ttFlash present limited effectiveness in mitigating fail-slow symptoms. In particular, Trace38 highlights the benefits of capacity variance, achieving a 3.16× better throughput than the fixed-capacity storage. For RocksDB, point lookups may end up consulting all files in level 0 and at most one file from each of the other levels. Therefore, as the Tr-SSD ages, a single `Get()` request can cause multiple physical reads and each of them can trigger SSD read retry several times, degrading the read performance drastically.

Moreover, we find that traditional systems with the original discard policy show higher utilization inconsistency (i.e.,



(a) Fileserver (b) Netsfs (c) Varmail (d) Host I/Os blocked by error handling under varmail

Figure 12: Performance results under Filebench workloads. CVSS reduces the average latency by 8% under fileserver workload (Figure 12a), 24% under netsfs workload (Figure 12b), and 10% under varmail workload (Figure 12c) compared to TrSS throughout the devices' lifetime. Before $CV_{degraded}$ is triggered, CVSS–normal reduces the average latency by 32% under netsfs workload. Figure 12d shows the percentage of host I/Os blocked by read retry operations under varmail workload. Other workloads show a similar pattern.

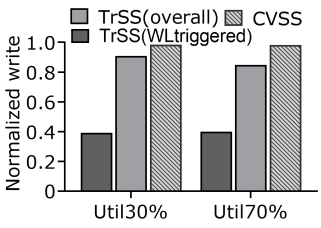


Figure 13: Average write throughput under FIO workloads. For TrSS, when wear leveling is triggered, the write throughput drops by 0.6×; on the other hand, by forgoing WL, CV-SSD provides a more stable and better write performance.

Figure 14: Performance results under Twitter traces. Capacity variance outperforms AutoStream and ttFlash and improves the throughput by 1.42× on average compared to TrSS.

$\frac{1}{n}\sum_{Observation=1}^{n} util_{SSD} - util_{FS}$) between FS and SSD, as shown in Figure 16. That is because of the high request rate during the experiments and F2FS only dispatches discard command when the device I/O is idle, which not only decreases SSD GC efficiency but also makes wear leveling more likely to misjudge data aliveness, limiting its effectiveness in maintaining capacity. During the experiments, the WAF of TrSS can be as high as 6.79, while only 1.12 for CVSS.

5.3 Lifetime Extension

In this section, we investigate how CVSS extends device lifetime given different performance requirements and thus leads to a longer replacement interval for SSD-based storage systems. We compare three different configurations: CVSS, TrSS, and AutoStream [63] in this evaluation since ttFlash introduces additional write (wear) overhead coming from RAIN (Redundant Array of Independent NAND) even for a small write [62]. The workloads used are similar to § 5.2.1.

Figure 15 shows the TBW before the device performance drops below 0.8, 0.6, 0.4, and 0 of the initial state. In particular, 0 represents the case where no performance requirement is applied so the workload runs until the underlying SSD is unusable. In cases of lower device utilization (as shown in Figures 15a and 15c), CVSS effectively extends the device lifetime, even when high performance is required. In Figure 15a, the device fails after accommodating 10 TB host writes for TrSS and 18 TB for AutoStream, considering the performance requirement of 0.8. On the other hand, CVSS accommodates 28 TB host writes with the same performance requirement, outlasting TrSS by 180% and AutoStream by 55%. Similarly, in Figure 15c, CVSS outlasts TrSS by 270% and AutoStream by 50%.

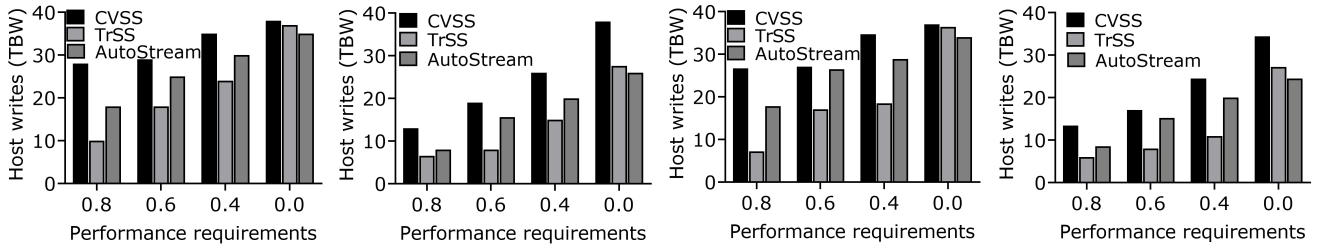
In the high device utilization cases (as shown in Figure 15b and 15d), CVSS outlasts TrSS by 123% and AutoStream by 55% on average with the highest performance requirement. In Figure 15b, before the device becomes unusable, CVSS accommodates 10.4 TB more in host writes compared to TrSS and 12 TB more compared to AutoStream. In our experiments, we found AutoStream achieves a longer lifetime than TrSS except for the no performance requirement case. In AutoStream, data are placed based on their characteristics, which in turn triggers more data relocation towards the end for wear leveling. Overall, with the highest performance requirement, CVSS ingests 168% host data more compared to TrSS and 57% more compared to AutoSteam on average, which in turn prolongs the replacement interval and reduces the cost.

5.4 Sensitivity Analysis

We next investigate the tradeoffs in CVSS regarding the block retirement threshold, the strength of ECC engine, and the impact of different GC formula weights.

5.4.1 Block Retirement Threshold

The mapping-out behavior for aged blocks in CV-SSD is controlled by a user-defined threshold. By default, blocks



(a) Zipfian (30% utilization) (b) Zipfian (70% utilization) (c) Random (30% utilization) (d) Random (70% utilization)

Figure 15: Terabytes written (TBW) with different performance requirements. Compared to TrSS and AutoStream, CVSS significantly extends the lifetime while meeting performance requirements.

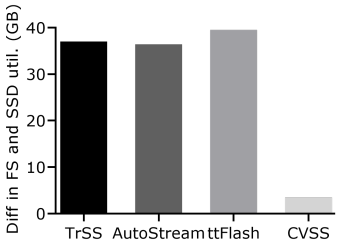


Figure 16: The average difference in FS and SSD utilization under Twitter traces. The original discard policy shows higher utilization inconsistency between FS and SSD, making data aliveness misjudged.

are mapped out and turn to a retired state once their RBER exceeds 5%. In this section, we investigate how this threshold affects the performance and device lifetime.

We utilize YCSB-A and YCSB-F with their data set configured to have thirty million key-value pairs (10 fields, 100 bytes each, plus key). We compare three different configurations: (1) TrSS, vanilla F2FS plus a fixed-capacity SSD; (2) CVSS(4%), CVSS with a higher reliability requirement. Superblocks will be mapped out if RBER is greater than 4%; (3) CVSS(6%), CVSS with a lower reliability requirement. Superblocks will be mapped out if RBER is greater than 6%.

Figure 17 shows the latencies at major percentile values (p75 to p99) and the device lifetimes for each workload. As shown in Figures 17a and 17b, CVSS(4%) reduces p99 latency by 51% for the YCSB-A workload and by 53% for the YCSB-F workload compared to TrSS, which are 44% and 40% for CVSS(6%). With a higher reliability requirement, blocks are retired earlier in CVSS(4%), which in turn causes a relatively shorter device lifetime than CVSS(6%). As depicted in Figures 17c and 17d, CVSS(6%) and CVSS(4%) ingests 3.27× and 2.68× host I/O than TrSS on average, respectively.

5.4.2 ECC Strength

We now study the impact of ECC strength on SSD error handling and demonstrate the usefulness of capacity variance in simplifying SSD FTL design. As discussed earlier, CVSS excludes aged blocks from use and thus incurs fewer error correction operations. This further allows the CV-SSD to be equipped with a less robust error-handling mechanism without compromising reliability.

Figure 18 compares the average number of read retries triggered per GiB read over the device's lifetime for CVSS with ECC strength set as up to 50 bits corrected per 4KiB and TrSS with ECC strength set to 50 – 90 bits. The results are measured under the FIO Zipfian read/write workload with device utilization of 30%. We make two observations. First, with the same ECC capability, TrSS(50) performs 1.93× more read retry operations than CVSS(50). Second, TrSS requires a stronger ECC engine to improve the efficiency of the error correction process, which complicates the FTL design in SSDs. On the other hand, with a weaker ECC engine, CVSS(50) achieves similar performance to TrSS(90).

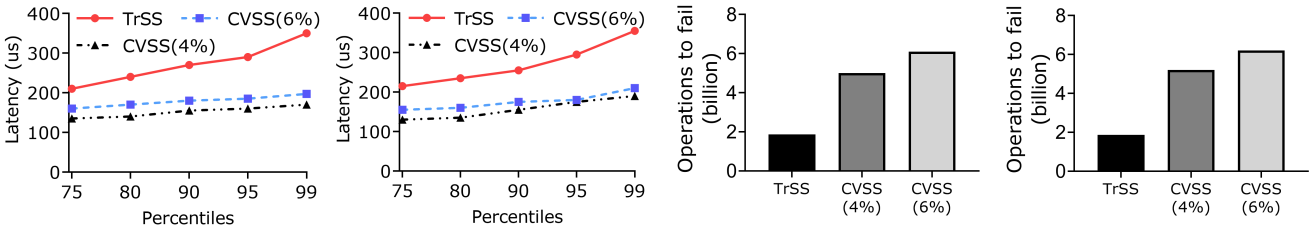
5.4.3 GC Formula

As described in § 3.2.3, the GC formula consists of three parameters: $W_{invalidity}$, W_{aging}, and W_{read}. We analyze how different weights used in GC formula affect the performance of CVSS in this section. We compare the configured weights with three different configurations: (1) GC prioritizes blocks with more invalid pages, with $W_{invalidity}$ = 0.6, W_{read} = 0.2, and W_{aging} = 0.2; (2) GC prioritizes blocks with more reads, with $W_{invalidity}$ = 0.2, W_{read} = 0.6, and W_{aging} = 0.2; (3) GC prioritizes blocks with more erases, with $W_{invalidity}$ = 0.2, W_{read} = 0.2, and W_{aging} = 0.6. FIO is used to generate Zipfian read-/write workloads to the device. Figure 19 illustrates the measured WAF and read retry. Overall, the configured weights result in a lower WAF and fewer read retry operations.

In particular, compared to the configured case, the high $W_{invalidity}$ case achieves a lower WAF but involves 0.78× more read retry operations. This is because read-intensive data are stored in aged blocks. For the high W_{read} case, it triggers fewer read retry operations but decreases the cleaning efficiency of GC since the invalidity is not adequately considered during the victim selection. In the high W_{aging} case, GC always selects the most aged blocks, leading to a significant increase in WAF and faster device aging. In contrast, the configured weights balance WAF and read retry within the device.

6 Discussion and Future Work

In this section, we discuss different use cases of capacity variance and its intersection with ZNS and RAID systems.



(a) Read latency (YCSB-A) (b) Read latency (YCSB-F) (c) Lifetime (YCSB-A) (d) Lifetime (YCSB-F)

Figure 17: Sensitivity analysis on the mapping-out threshold in CVSS. CVSS with a higher reliability requirement, CVSS(4%), achieves better performance but with a relatively shorter lifetime compared to CVSS(6%) because blocks are retired earlier.

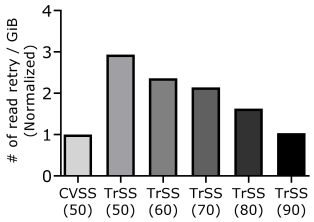


Figure 18: The average number of read retries triggered per GiB read over the device's lifetime. The *x*-axis represents different ECC strengths in bits.

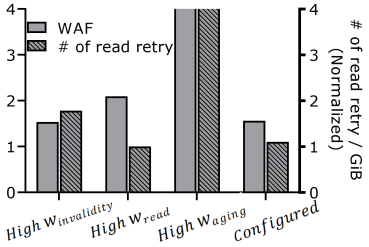


Figure 19: The WAF and read retries triggered under different weights used for GC formula.

Use cases of capacity variance. CVSS aims to significantly outperform fixed-capacity systems in the best case, and perform at a similar level in the worst case. The degraded mode serves the role of addressing the worst case by reserving a particular amount of capacity for the host. CVSS would be most useful for cases where IO performance is bottlenecked but has spare capacity. For instance, Haystack is the storage system specialized for new blobs (Binary Large Objects) and bottlenecks on IOPS but has spare capacity [51].

Moreover, for SSD vendors, capacity variance can simplify SSD design, as it allows for the tradeoff of performance and reliability with capacity. For data centers, introducing capacity variance can automatically exclude unreliable blocks and enable easy monitoring of device capacity, resulting in longer device replacement interval and mitigating SSD failure-related issues in data center environments. Lastly, for desktop users, capacity variance extends the lifetime of SSDs significantly and thus reduces the overall cost of storage.

ZNS-SSD. Capacity variance can be harmonious with ZNS. Specifically, due to a wear-out, a device may (1) choose to take a zone offline, or (2) report a new, smaller size for a zone after a reset. Both of these result in a shrinking capacity SSD. However, there is no software that can handle capacity reduction for ZNS-SSDs currently. The offline command simply makes a zone inaccessible and data relocation has to be done by users. Moreover, file systems are typically unaware of this change except for ZoneFS [34]. The capacity-variant SSD interface is a more streamlined solution where the software and the hardware cooperate to automate the process.

Capacity variance with RAID. The current CVSS does not support RAID systems. Existing RAID architectures require symmetrical capacity across devices and its overall capacity depends on the underlying minimal-capacity device. For parity RAID, the invalid data can not always be trimmed because it may be required to ensure the parity correctness. We will investigate the capacity-variant RAID system as our next direction, in which we consider modifying the disk layout and data placement scheme to support dynamically changing asymmetrical capacity with multiple heterogeneous CV-SSDs.

7 Conclusion

The basic principle behind a capacity-variant storage system is simple: relax the fixed-capacity abstraction of the underlying storage device. We implement this idea and describe the key designs and implementation details of a capacity-variant storage system. Our evaluation result demonstrates how capacity variance leads to performance advantages and shows its effectiveness and usefulness in avoiding SSD fail-slow symptoms and extending device lifetime. We expect new optimizations and features will be continuously added to the capacity-variant storage system.

Acknowledgment

We would like to thank our shepherd, Keith A. Smith, and the anonymous reviewers for their constructive feedback. This research was supported in part by the National Science Foundation (NSF CNS-2008453), Samsung Memory Solutions Lab through the Alternative Sustainable and Intelligent Computing Industry-University Cooperative Research Center (NSF CNS-1822165), and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (MSIT), Korea (No. 2021-0-01475, SW Starlab).

References

[1] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 125–138. USENIX, 2009.

[2] Brian Cooper. Yahoo Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/, 2010.

[3] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, pages 1666–1704, 2017.

[4] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in MLC NAND flash memory: characterization, analysis, and modeling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1285–1290. ACM, 2013.

[5] Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE Computer Society, 2015.

[6] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrián Cristal, Osman S. Ünsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *International Conference on Computer Design (ICCD)*, pages 94–101. IEEE Computer Society, 2012.

[7] Chandranil Chakraborttii and Heiner Litz. Improving the accuracy, adaptability, and interpretability of SSD failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SOCC)*, pages 120–133. ACM, 2020.

[8] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *ACM Symposium on Applied Computing (SAC)*, pages 1126–1130, 2007.

[9] Fu-Hsin Chen, Ming-Chang Yang, Yuan-Hao Chang, and Tei-Wei Kuo. PWL: a progressive wear leveling to minimize data migration overheads for NAND flash devices. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1209–1212, 2015.

[10] Zhe Chen and Yuelong Zhao. DA-GC: A dynamic adjustment garbage collection method considering wear-leveling for SSD. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 475–480, 2020.

[11] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, pages 267–290, 1999.

[12] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58. USENIX, 2017.

[13] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 15–21. USENIX, 2019.

[14] Yajuan Du, Siyi Huang, Yao Zhou, and Qiao Li. Towards LDPC read performance of 3D flash memories with layer-induced error characteristics. *ACM Transactions on Design Automation of Electronic Systems*, pages 44:1–44:25, 2023.

[15] Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. UBI - Unsorted Block Images. http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf, 2006.

[16] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14. USENIX, 2018.

[17] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 91–103. USENIX, 2011.

[18] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An integrated approach for managing read disturbs in high-density NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1079–1091, 2016.

[19] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of

mobile storage systems using a decoupled defragmenter. In *USENIX Annual Technical Conference (ATC)*, pages 759–771. USENIX, 2017.

[20] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting millisecond tail tolerance with fast rejecting slo-aware OS interface. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 168–183. ACM, 2017.

[21] Mingzhe Hao, Gokul Soundararajan, Deepak R. Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 263–276. USENIX, 2016.

[22] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–190. USENIX, 2020.

[23] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio/, 2005.

[24] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing (TMC)*, pages 2062–2076, 2019.

[25] Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. Wear leveling in SSDs considered harmful. In *ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 72–78. ACM, 2022.

[26] Ziyang Jiao and Bryan S. Kim. Generating realistic wear distributions for SSDs. In *ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 65–71. ACM, 2022.

[27] Myoungsoo Jung and Mahmut T. Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 203–216. ACM, 2013.

[28] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 13–17. USENIX, 2014.

[29] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production Windows servers. In *International Symposium on Workload Characterization (IISWC)*, pages 119–128, 2008.

[30] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 281–294. USENIX, 2019.

[31] Bryan S. Kim, Eunji Lee, Sungjin Lee, and Sang Lyul Min. CPR for SSDs. In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*, pages 201–208. ACM, 2019.

[32] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: log-structured file system without garbage collection. In *USENIX Annual Technical Conference (ATC)*, pages 739–754. USENIX, 2022.

[33] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Joo Young Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 295–308. USENIX, 2019.

[34] Damien Le Moal and Ting Yao. Zonefs: Mapping POSIX file system interface to raw zoned block device accesses. In *Linux Storage and Filesystems Conference (VAULT)*, page 19. USENIX, 2020.

[35] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: a new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286. USENIX, 2015.

[36] Jaeyong Lee, Myungsuk Kim, Wonil Choi, Sanggu Lee, and Jihong Kim. Tailcut: improving performance and lifetime of SSDs using pattern-aware state encoding. In *ACM/IEEE Design Automation Conference (DAC)*, pages 409–414. ACM, 2022.

[37] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *USENIX Conference on File and Storage Technologies (FAST)*, page 83–90. USENIX, 2018.

[38] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A host/device co-design for strong predictability contract on modern flash storage. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 263–279. ACM, 2021.

[39] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conference on File and Storage Technologies (FAST)*, page 11. USENIX, 2012.

[40] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A fail-slow detection framework for cloud storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 49–64. USENIX, 2023.

[41] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD failures in the field: the fail-stop and the fail-slow. In *USENIX Annual Technical Conference (ATC)*, pages 1005–1020. USENIX, 2022.

[42] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 504–517. IEEE Computer Society, 2018.

[43] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of SSDs in enterprise storage systems: A large-scale field study. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 165–180, 2022.

[44] Neal R. Mielke, Robert E. Frickey, Ivan Kalastirsky, Minyan Quan, Dmitry Ustinov, and Venkatesh J. Vasudevan. Reliability of solid-state drives based on NAND flash memory. *Proceedings of the IEEE (JPROC)*, pages 1725–1750, 2017.

[45] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey. https://github.com/utsaslab/crashmonkey/tree/master/code/tests/seq1, 2018.

[46] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 33–50. USENIX, 2018.

[47] NVM Express. NVM Express base specification 2.0. https://nvmexpress.org/developers/nvme-specification/, 2021.

[48] NVM Express. NVMe Command Line Interface. https://github.com/linux-nvme/nvme-cli, 2021.

[49] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 343–354. ACM, 2016.

[50] Open NAND Flash Interface. ONFI 5.0 spec. http://www.onfi.org/specifications/, 2021.

[51] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic filesystem: Efficiency from exascale. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 217–231. USENIX, 2021.

[52] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: a fail-slow detection and mitigation framework for distributed storage services. In *USENIX Annual Technical Conference (ATC)*, pages 47–62. USENIX, 2019.

[53] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. Reducing solid-state drive read latency by optimizing read-retry. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 702–716. ACM, 2021.

[54] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15. ACM, 1991.

[55] Xin Shi, Fei Wu, Shunzhuo Wang, Changsheng Xie, and Zhonghai Lu. Program error rate-based wear leveling for NAND flash memory. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1241–1246. IEEE, 2018.

[56] Youngseop Shim, Myungsuk Kim, Myoungjun Chun, Jisung Park, Yoona Kim, and Jihong Kim. Exploiting process similarity of 3D flash memory for high performance SSDs. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 211–223. ACM, 2019.

[57] Seungwoo Son and Jaeho Kim. Differentiated protection and hot/cold-aware data placement policies through k-means clustering analysis for 3D-NAND SSDs. *Electronics*, pages 398–412, 2022.

[58] The SSD Guy. Comparing wear figures on SSDs. https://thessdguy.com/comparing-wear-figures-on-ssds/, 2017.

[59] Theodore Ts'o. Ext2/3/4 file system utilities. https://e2fsprogs.sourceforge.net/, 2009.

[60] Vasily Tar Asov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf, 2016.

[61] Fan Xu, Shujie Han, Patrick P. C. Lee, Yi Liu, Cheng He, and Jiongzhou Liu. General feature selection for failure prediction in large-scale SSD deployment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 263–270. IEEE, 2021.

[62] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 15–28. USENIX, 2017.

[63] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*, pages 1–11. ACM, 2017.

[64] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 191–208. USENIX, 2020.

[65] Kong-Kiat Yong and Li-Pin Chang. Error diluting: Exploiting 3-D NAND flash process variation for efficient read on ldpc-based SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 3467–3478, 2020.

[66] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. Multi-view feature-based SSD failure prediction: What, when, and why. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 409–424. USENIX, 2023.

[67] Hao Zhou, Zhiheng Niu, Gang Wang, Xiaoguang Liu, Dongshi Liu, Bingnan Kang, Hu Zheng, and Yong Zhang. A proactive failure tolerant mechanism for SSDs storage systems based on unsupervised learning. In *IEEE International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2021.

[68] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 187–202. USENIX, 2021.

A Artifact Appendix

Abstract

As introduced in the paper, the current storage system abstraction of fixed capacity exacerbates aging-related performance degradation for modern SSDs, and enabling capacity variance allows for more effective tradeoffs between capacity, performance, and reliability. This artifact includes the code and describes the steps for measuring and comparing the performance of the proposed capacity-variant storage system against the traditional storage system to support our major claims. The experiments are performed on a machine with 32 CPUs and 1 TiB of memory running Ubuntu 20.04 LTS.

Scope

The provided code and scripts facilitate the testing of the following experiments:

- The performance degradation caused by aging observed on a real SSD (Figure 1).
- The functionality of CVSS, including CV-FS, CV-SSD, and CV-manager.
- The FIO experiments (Figure 10, Figure 11, and Figure 13).
- The Filebench experiments (Figure 12).
- The Twitter traces experiments (Figure 14).
- The lifetime experiments (Figure 15).

Contents

A.0.1 Fail-slow Experiments (Section 1)

Scripts are provided to age the SSD and measure its read-only I/O performance. To initiate the experiment:

```
$ ./fio_aging.sh
```

Note that the content of the tested SSD will be wiped out by the above script. The estimated time for this experiment depends on the endurance of the tested device and it may take several months to fully age the device.

A.0.2 Installation of CVSS

This section describes the steps to set up CVSS and perform basic tests. The `REMAP` interface is implemented on Linux kernel v5.15. To compile the kernel:

```
$ make -j$(nproc) bindeb-pkg
```

The CV-FS is configured as a kernel module. To compile and install the CV-FS:

```
$ ./run.sh
```

The CV-SSD is based on FEMU. To compile the code and start the virtual machine, please run the following commands after cloning the repository:

```
$ cd FAST24_CVSS_FEMU
$ mkdir build-femu
$ cd build-femu
$ cp ../femu-scripts/femu-copy-scripts.sh ./
$ ./femu-copy-scripts.sh ./
$ ./run-blackbox.sh
```

This will start the virtual machine with the emulated CV-SSD. You can set the path to your VM image via `IMGDIR=/path/to/image` in the `run-blackbox.sh` file.

A.0.3 Basic Test

To test the functionality of CVSS:

```
$ inscvfs
$ diskcvfs
$ df -h /dev/nvme0n1
```

The logical capacity of CVSS can be adjusted online by issuing the following command:

```
$ sudo cvfs.f2fs /dev/nvme0n1 -t 118
```

The parameter for the `-t` flag (e.g., 118) is the newly configured logical capacity in GiB that we have set for the system.

A.0.4 Evaluation Workflow

FIO experiments (Section 5.2.1). To evaluate the performance of CVSS under FIO-related workloads, please run:

```
$ ./test_fio_zipfian_util30.sh
$ ./test_fio_zipfian_util70.sh
$ ./test_fio_random_util30.sh
$ ./test_fio_random_util70.sh
```

Each experiment may take 4 days to finish. The virtual machine will be turned off when the experiment finishes, and the performance results will be stored in `.log` files in the working directory.

Filebench experiments (Section 5.2.2). To perform the filebench-related experiments, please run:

```
$ ./fs_test.sh
```

This script will age the system and issue *Fileserver*, *Netsfs*, and *Varmail* workloads under different aged states of the underlying device. The latency results are logged in `.log` files in the working directory.

Twitter traces experiments (Section 5.2.3). To set up RocksDB and issue Twitter traces to the system, please run:

```
$ cd ./rocksdb/examples
$ gcc twitter_load.c -o twitter_load
$ gcc twitter_run.c -o twitter_run
$ ./twitter.sh
```

Each test may take one week to complete. The IOPS and trace profiles are stored in the `.log` files.

Lifetime experiments (Section 5.3). To test the amount of host writes under different performance requirements and workloads, please run:

```
$ ./test_lifetime_zipfian_util30.sh
$ ./test_lifetime_zipfian_util70.sh
$ ./test_lifetime_random_util30.sh
$ ./test_lifetime_random_util70.sh
```

Each experiment is expected to take approximately 5 days to complete. The experiments will continue running until the underlying SSD fails. Performance results are documented in the `.log` files within the working directory. Additionally, device statistics, such as the write amplification factor, can be found in the `wa.log` file located in the host directory of the virtual machine.

Hosting

The artifact is available on github repositories:

- Kernel: https://github.com/ZiyangJiao/FAST24_CVSS_Kernel.
- CV-FS: https://github.com/ZiyangJiao/FAST24_CVSS_CVFS.
- CV-SSD: https://github.com/ZiyangJiao/FAST24_CVSS_FEMU.

Requirements

Please make sure you have at least 160 GiB of memory and 150 GiB of free space on your disk if testing on your machine. Our evaluation is based on the following system specifications:

Component	Specification
Processor	Intel(R) Xeon(R) Silver 4208 CPU, 32-Core
Architecture	x86_64
Memory	DDR4 2666 MHz, 1 TiB (64 GiB x16)
SSD	Intel DC P4510 1.6TiB
OS	Ubuntu 20.04 LTS (Focal Fossa)