ScaleLFS: A Log-Structured File System with Scalable Garbage Collection for Commodity SSDs

Jin Yong Ha, *Seoul National University;* Sangjin Lee, *Chung-Ang University;* Hyeonsang Eom, *Seoul National University;* Yongseok Son, *Chung-Ang University*

https://www.usenix.org/conference/fast25/presentation/ha

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

ScaleLFS: A Log-Structured File System with Scalable Garbage Collection for Commodity SSDs

Jin Yong Ha[1], Sangjin Lee[2], Hyeonsang Eom[1], and Yongseok Son[2*]
[1]Seoul National University
[2]Chung-Ang University

Abstract

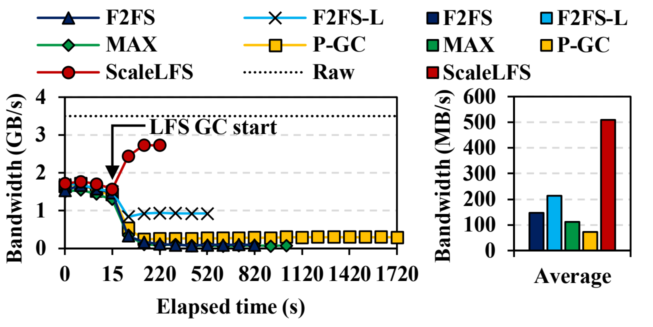
We present a log-structured file system (LFS) with scalable garbage collection (GC) called `ScaleLFS` for providing higher sustained performance on commodity SSDs. Specifically, we first introduce a per-core dedicated garbage collector to parallelize the GC operations and utilize dedicated resources. Second, we present a scalable victim manager that selects victim segments and updates the metadata of the segments concurrently. Finally, we propose a scalable victim protector to enable a page-level GC procedure instead of a file level to increase GC concurrency while resolving the conflict with victim pages. We implement `ScaleLFS` with three techniques based on F2FS in the Linux kernel. Our evaluations show that `ScaleLFS` provides higher sustained performance by up to 3.5×, 4.6×, and 7.0× compared with F2FS, a scalable LFS, and a parallel GC scheme, respectively.

1 Introduction

The log-structured file system (LFS) [16, 18, 19, 22, 24, 25, 29, 32, 34, 35, 41, 44–46] has gained popularity as a storage solution since it gathers small and random writes to large and sequential writes which brings significant performance advantage in solid-state drives (SSDs). However, LFSs rely on the availability of contiguous free segments, maintaining the segments requires expensive reclaim operations [41]. Specifically, as the log-structured file system ages, the file system runs out of free segments and needs to reclaim the invalid pages to make more free segments. The activity of reclaiming the invalid pages is called garbage collection (GC) which becomes a significant challenge in LFSs [10,16,35,37,37,43,45]. Unfortunately, when the file system performs the garbage collection in the foreground, it freezes the entire file system until it completes, resulting in high latency, low bandwidth, and long execution time of application.

Challenges: Figure 1 shows the application and device-level bandwidth in various LFSs [22, 25, 35] under a GC-intensive workload using a micro-benchmark (i.e., FIO [5]). As shown in Figure 1a, when GC occurs around 15 seconds, the device-level performance of existing LFSs notably drops, resulting

*Corresponding Author: Yongseok Son (`sysganda@cau.ac.kr`).



(a) Device BW. (b) Application BW.

Figure 1: Bandwidth (BW) and runtime under GC process.

in a decline in application-level performance as in Figure 1b. It is because 1) a *serialized* GC procedure with a one-to-all model (e.g., F2FS [22] and MAX [25]) and 2) *lock contention* on the GC procedure even if a parallel GC scheme (e.g., P-GC [35]) is adopted. Through this preliminary evaluation, we observe the potential to leverage the available device-level bandwidth, despite the active migration during GC.

Prior studies: Prior studies [16, 25, 35, 45] have investigated LFSs to enhance scalability and minimize GC overhead. MAX [25] targets improving the scalability of LFS with SSD-friendly design. P-GC [35] introduces new free segment search and parallel GC schemes for ZNS SSDs. However, the GC overhead in both studies [25, 35] still remains potential bottleneck points (e.g., high lock contention). ParaFS [45] mitigates GC overhead by coordinating GC in both the file system and FTL. IPLFS [16] removes GC from the log-structured file system by using infinite partition and interval mapping schemes on OpenSSD [2]. ParaFS and IPLFS mitigate or remove the GC overhead in LFS while leveraging specially customized SSDs. Our study is inspired by and in line with these studies [16, 25, 35, 45] in terms of investigating the scalability and GC overhead of LFS. In contrast, we focus on enabling dedicated resource utilization and concurrent GC in LFS to maximize GC scalability on commodity SSDs.

Contributions: In this paper, we propose a log-structured file system with scalable GC called `ScaleLFS` which enhances

GC scalability on multi-cores and commodity SSDs to provide the higher sustained[1] performance of LFS. Our key idea is to leverage the potential of increasing concurrency and parallelism of GC in LFS without sacrificing consistency. Specifically, we first introduce a per-core and dedicated garbage collector (`DGC`) to parallelize and accelerate the overall GC procedure. Each `DGC` has its own dedicated GC resource (e.g., victim segment, page buffer, and write stream). This enables a one(collector)-to-one(resource) model to exploit high parallelism inherent among multi-cores and SSDs, instead of a one(collector)-to-all(victims) model where a single collector handles all segments.

Second, we present a scalable victim manager (`SVM`) that selects victim segments concurrently with a combination of atomic bitmaps and instructions. In addition, it concurrently updates segment metadata with a loose-synchronization mechanism, albeit at the expense of optimal segment selection and minimal false-positive read which is unnecessarily submitted to read invalidated address. However, this approach does not compromise LFS consistency. Lastly, we propose a scalable victim protector (`SVP`) to enable a page-level GC procedure instead of a file-level one by leveraging a concurrent hash table. This allows pages within a file to be cleaned simultaneously while resolving the conflict with victim pages between I/O threads and `DGC`s.

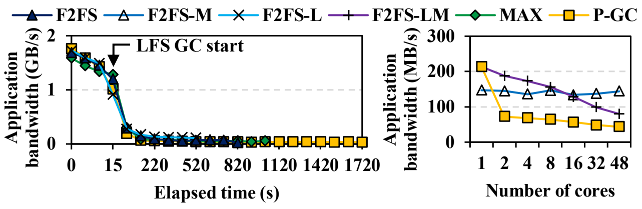
We implement `ScaleLFS` with three components based on F2FS [22] in Linux kernel 6.0.0. We evaluate `ScaleLFS` using the micro [5] and macro [38] benchmarks and real-world application [1]. The results show that `ScaleLFS` provides higher sustained performance by up to 3.5×, 4.6×, and 7.0× compared with F2FS, a scalable LFS (MAX [25]), and a parallel GC scheme (P-GC [35]), respectively. Finally, we open the source code at `https://github.com/syslab-CAU/ScaleLFS`.

2 Background and Motivation

Log-structured file system (LFS): The LFS is designed based on an append-only approach, consolidating random write requests into a larger sequential write request for hard disk drives and flash-based SSDs [18, 22, 34]. This append-only brings another complexity to the file system. As the LFS ages, it runs out of free segments and needs expensive garbage collection to free segments [41]. Accordingly, previous studies [16, 35, 45] show that this GC procedure in the LFSs significantly degrades the application performance.

Garbage collection: When free segments are not enough below a predefined GC threshold, the LFS triggers a GC procedure. Then, a single GC daemon thread wakes up and scans the dirty segment bitmap to find the victim segment that has the highest ratio of invalidated pages. Then, the GC thread scans the valid page bitmap of the victim segment to determine valid pages, reads the valid pages, writes them to

[1]In this paper, we define "sustained" as the state in which LFS level GC is actively triggered.



(a) Application bandwidth during GC. (b) Thread scalability.

Figure 2: Sustained performance of existing LFSs (M: modified with multiple threads, L: LFS mode).

the write stream dedicated to the new destination, and finally cleans the victim segment.

Opportunities in application and GC behaviors: When a GC thread wakes up, the application threads are blocked until the GC thread completes its work. Thus, the GC speed determines the blocking time of the application threads, which in turn affects the sustained performance. During the GC procedure, there are two potentials for improving the GC speed. First, while waiting for GC to complete, the blocked application threads do not occupy cores. Thus, the unoccupied cores can be utilized to speed up GC without interfering with the applications. Second, while the application threads are blocked, the bandwidth of high-end SSDs is underutilized since only a single GC thread performs GC I/O operations. By leveraging unoccupied cores and available device bandwidth, we aim to accelerate the GC speed to minimize the application blocking time.

3 Sustained Performance Analysis

3.1 Sustained Performance on Modern LFSs

To understand the sustained performance of modern LFSs (i.e., F2FS, MAX, and P-GC), we run FIO [5] with random write on our testbed and evaluation scenario as described in Section 5.1. Figure 2a shows application-level bandwidth changes in the LFSs during GC. When the GC procedure is triggered around 15 seconds due to insufficient free spaces, it sharply decreases the application write bandwidth by up to 68×. At the same time, as shown in Figure 1a, device-level bandwidth (sum of read and write) also decreases by up to 24.8× when the GC is triggered. This implies that even during GC, the SSD has idle bandwidth to handle additional I/O requests, indicating the bottleneck originates and the potential for improvement in the LFS GC.

Insight#1*: LFS can accelerate the GC process by leveraging the available device-level bandwidth for higher sustained performance.*

3.2 Identifying Root Causes

3.2.1 Serialized GC procedure

The main reason for low sustained performance comes from the serialized GC process with a single garbage collector in LFSs (i.e., F2FS and MAX). Due to the substantial amount of job involved in performing GC, relying on a single garbage collector can induce a bottleneck. This bottleneck delays application I/O requests and significantly impacts overall application performance. Furthermore, the overhead of the serialized GC process is more noticeable on high performance systems equipped with multi-core CPUs and high-end commodity SSDs, as the system's full potential remains underutilized.

Insight#2*: One-to-all model is inefficient for leveraging the high device bandwidth of modern SSDs on a multi-core system.*

3.2.2 Obstacles for Scaling GC Procedure

Figure 2b shows the scalability of a modified version of F2FS in default (F2FS-M) and LFS modes (F2FS-LM) with multiple threads[2] and F2FS with P-GC (P-GC) [35]. The results show that they do not scale well (i.e., F2FS-M) or decrease the performance (i.e., F2FS-LM and P-GC) even if the number of threads increases. As a prior study, P-GC distributes valid pages in a victim segment among multiple threads if the number of its valid pages exceeds half of the number of pages in the segment. However, similar to F2FS-LM, continuously increasing the number of threads affects performance negatively due to increasingly severe lock contention in the GC procedure. Interestingly, the performance of F2FS-M is not affected by the number of GC threads. The rationale is that the default mode utilizes slack space recycling (SSR) which induces random writes to invalidated pages in a segment instead of cleaning the segment via GC. Although GC is not triggered frequently, there is still a substantial amount of available bandwidth as described in Figure 1a. By analyzing the design of existing LFSs, we observe the main obstacles that hinder the scaling of the GC procedure (see Table 3 in Section 5.4).

Single GC stream and lock contention: The LFSs maintain only one write stream for GC I/O operations (i.e., cold data stream). They also serialize the free space allocation from the stream via a mutex (`curseg_mutex`). This can hinder exploiting the high parallelism offered by modern SSDs and multi-cores.

Excessive contention on victim selection: To identify and select dirty segments, the LFSs scan a global bitmap (i.e., dirty segment bitmap) using a mutex lock (`seglist_lock`), ensuring the consistency of the bitmap during the scan process. However, this coarse-grained scan operation leads to the under-utilization of multi-core parallelism.

Over-strict synchronization for segment metadata: The segment metadata (e.g., valid page bitmap and valid page count) in the LFSs is strictly managed via a semaphore lock (`sentry_lock`), ensuring the update and access to the metadata are synchronized. However, this strict synchronization acts as an obstacle when segment metadata is accessed and updated frequently.

Coarse-grained file-level protection: The LFSs protect data consistency between GC and other I/O operations (e.g., direct I/O, trim, punch hole, etc.) via a semaphore (`i_gc_rwsem`) during a GC operation. However, since this GC semaphore is a file-level lock for GC inside the LFS, protecting the entire file data, it can limit SSD and multi-core parallelism.

Insight#3*: To further scale the GC procedure, additional concurrent and scalable techniques are essential.*

[2]We simply increase the number of GC threads in F2FS.

4 Design and Implementation

4.1 Design Goals of ScaleLFS

We design `ScaleLFS` to meet the following four goals:

- **Efficient parallel GC procedure:** To exploit the parallelism of multi-cores and multi-channel SSDs, `ScaleLFS` should perform the overall GC process in a dedicated and parallel manner.
- **Concurrent victim segment selection:** To allow victim segments to be selected without lock contention, `ScaleLFS` should select the victim segments concurrently.
- **Concurrent access/update on segment metadata:** To eliminate the lock contention on segment metadata, `ScaleLFS` should access and update segment metadata concurrently while not compromising file system consistency.
- **Finer and lightweight GC:** To enable a finer and lightweight GC process, `ScaleLFS` should concurrently perform GC at a page level while resolving the page conflict between I/O and GC threads.

4.2 Strategies of ScaleLFS

We present the four key strategies and explain how they meet the design goals of `ScaleLFS`.

Strategy #1: *To parallelize the GC process and minimize resource sharing, `ScaleLFS` adopts a per-core and resource dedication strategy. By doing so, `ScaleLFS` enables a one-to-one model instead of a one-to-all model.*

Strategy #2: *To enable scalable victim selection, `ScaleLFS` concurrently selects victim segments via concurrent data structures and atomic instructions without locking.*

Strategy #3: *To update segment metadata concurrently, `ScaleLFS` adopts a loose-consistency model by atomically updating each metadata individually at the cost of less optimal victim segment selection and additional false-positive read.*

Strategy #4: *To minimize contention between GC and I/O threads on a victim file, `ScaleLFS` adopts a scalable page-level GC while protecting victim pages. It allows multiple GC threads to clean a file simultaneously.*

4.3 Overall Architecture of ScaleLFS

Figure 3 depicts the overall architecture and procedure of `ScaleLFS`. `ScaleLFS` is comprised of three components: dedicated garbage collector (`DGC`), scalable victim manager (`SVM`), and scalable victim protector (`SVP`).

Dedicated garbage collector (DGC): `ScaleLFS` adopts per-core based `DGC` with a resource dedication (**Strategy #1**) where per-core garbage collector has its own dedicated GC resource (i.e., victim, page buffer, write stream) to realize

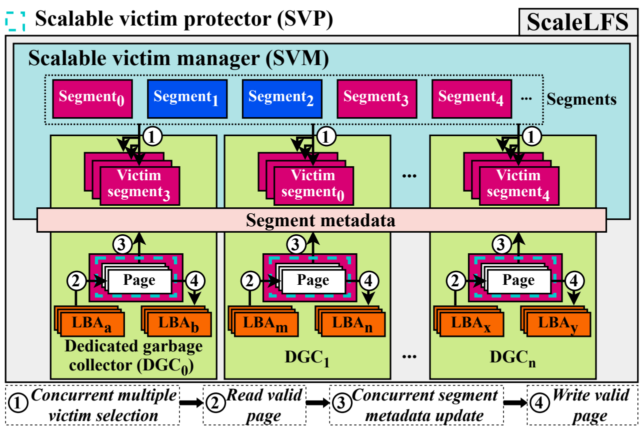

Scalable victim protector (SVP)
ScaleLFS
Scalable victim manager (SVM)
Segment_0 Segment_1 Segment_2 Segment_3 Segment_4 ··· Segments
Victim segment_3
Victim segment_0
Victim segment_4
Segment metadata
Page
LBA_a LBA_b LBA_m LBA_n LBA_x LBA_y
Dedicated garbage collector (DGC_0)
DGC_1
DGC_n
① Concurrent multiple victim selection
② Read valid page
③ Concurrent segment metadata update
④ Write valid page

Figure 3: ScaleLFS architecture and procedure.

one(collector)-to-one(resource) model. By doing so, this minimizes resource sharing between DGCs and leads to increasing GC parallelism.

Scalable victim manager (SVM): ScaleLFS adopts SVM to enable DGCs to select victim segments and update their metadata concurrently (**Strategy #2**). For concurrent victim selection, SVM leverages atomic bitmaps and a concurrent access/update mechanism with atomic instructions. SVM guarantees that DGCs concurrently select distinct victim segments, ensuring that no segment is shared between DGCs as shown in Figure 3. For the concurrent victim metadata update, SVM divides victim segment metadata into two pieces: valid page bitmap and valid page count. Then, SVM updates each metadata atomically with a loose-synchronization model (**Strategy #3**). However, the model can potentially lead to inconsistency in segment metadata temporarily and incur side effects (i.e., less-optimal victim selection and false-positive GC read). Thus, we analyze all the possible cases of side effects and resolve them. By doing so, eventually, we demonstrate SVM does not sacrifice file system consistency.

Scalable victim protector (SVP): ScaleLFS adopts SVP to enable DGC to perform GC operations at a page level while resolving page conflict between GC and I/O threads (**Strategy #4**). To this end, we utilize a scalable page-level protection approach based on a concurrent hash table (dashed boxes in the figure). By doing so, this approach eliminates the lock contention among threads (i.e., between I/O and GC threads, and between GC threads) to access the victim file while resolving the page conflicts.

Overall procedure: When the GC is triggered, per-core DGCs select their victim segments via SVM concurrently (①). Then, each DGC reads valid pages from each victim segment to the dedicated page buffers (②). After reads, DGC concurrently updates segment metadata (i.e., invalidate old LBAs and allocate new LBAs) via SVM (③). Finally, DGC writes valid pages to the allocated new LBAs (④). During reading or writing valid pages without file-level protection, pages are protected by SVP from the race between I/O threads and DGCs.

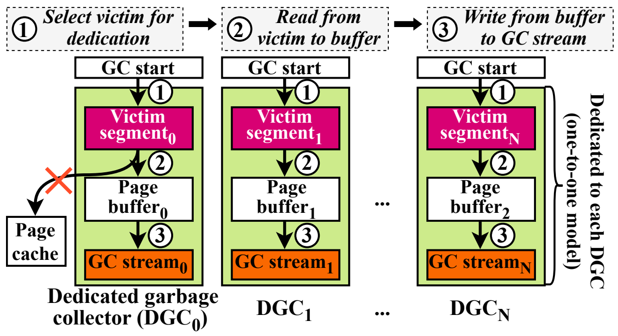

① Select victim for dedication
② Read from victim to buffer
③ Write from buffer to GC stream
GC start
Victim segment_0
Page buffer_0
GC stream_0
Page cache
Dedicated garbage collector (DGC_0)
Victim segment_1
Page buffer_1
GC stream_1
DGC_1
Victim segment_N
Page buffer_2
GC stream_N
DGC_N
Dedicated to each DGC (one-to-one model)

Figure 4: Procedure of dedicated garbage collector (DGC).

4.4 Dedicated Garbage Collector (DGC)

Most existing LFSs [6, 22, 25, 29] perform a GC procedure in a serialized manner. This indicates that only one garbage collector processes all dirty segments (i.e., one-to-all model). To enable a parallel GC procedure with a one-to-one model, we propose a per-core and dedicated garbage collector (DGC) with a resource dedication. As shown in Figure 4, DGC includes two main dedicated resources: dedicated page buffer and dedicated write stream. When a GC procedure is triggered, per-core DGCs wake up and start GC as following three steps. First, each DGC gets dedicated victim segments from SVM (①). This dedicated victim allows DGC to avoid contention caused by sharing the same victim with other DGCs. Second, each DGC reads valid pages within its selected victim segments from the device to the dedicated page buffer (②). By utilizing a dedicated page buffer, DGC can avoid accessing the page cache which incurs the overhead of page cache management. Finally, DGC writes the valid pages to a dedicated write stream (i.e., GC stream) (③). The dedicated write stream eliminates the contention between DGCs to allocate LBA.

Note that, as described in Section 2, during a GC procedure, application threads are blocked until the procedure is completed, leading to underutilized cores. Under this situation, DGC utilizes the unused cores without interfering with the application threads. ScaleLFS can reduce the application blocking time by leveraging the CPU idle time.

4.4.1 DGC-dedicated Page Buffer

Page cache management can be expensive (e.g., lock contention) when multiple threads frequently access and update the page cache [33]. Furthermore, the fact that the valid pages have been selected as victims indicates that they have not been overwritten up to this point, and their hotness tends to be low. Thus, this can lead to the pollution of the page cache with less frequently accessed pages, decreasing the page cache hit ratio. To address the page cache overhead, DGC allocates its dedicated page buffer (file I/O page) for reading and writing valid pages independently of the page cache. As a result, the dedicated page buffer (DPB) enables DGC to avoid contention on the page cache, increasing GC scalability. Utilizing DPB instead of a page cache can incur a temporary data inconsistency issue. We explain the solution for this issue in Section 4.7.

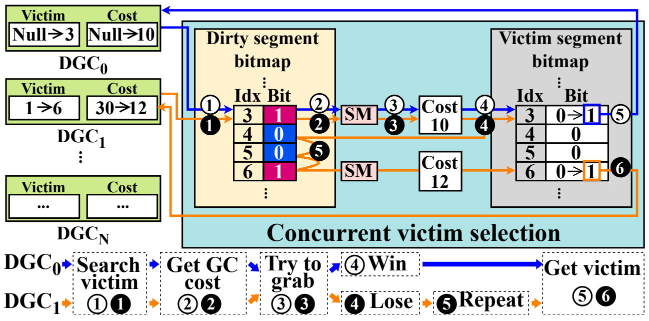


Figure 5: Concurrent victim selection (CVS) procedure (**SM**: segment metadata).

4.4.2 DGC-dedicated Write Stream

Allocating new LBAs for multiple DGCs from a single shared write stream can induce significant lock contention. To avoid this, ScaleLFS adopts a dedicated write stream (DWS) for each DGC. By doing this, DGC can allocate new LBAs from its dedicated write stream without any contention. Meanwhile, DWS incurs metadata space overhead to be stored in SSD. Specifically, to track the status of each write stream upon power cycle, the status should be checkpointed. Each DWS requires an additional 7 bytes, increasing the checkpoint size from 192 bytes in F2FS to 528 bytes in ScaleLFS at the 48 cores. However, the performance impact by the increased size is negligible since it is significantly small compared with the tens of KB of existing metadata written by a checkpoint (e.g., node address table, segment summary pages, etc, in F2FS).

4.5 Scalable Victim Manager (SVM)

Our dedicated technique (DGCs) increases the GC parallelism, however, it introduces another scalability issue of managing GC metadata in the LFS. In particular, the serialized selection of victim segments and management for metadata of the segments become potential bottlenecks. To address these bottlenecks, we propose a scalable victim manager (SVM) that incorporates two techniques, concurrent victim selection (CVS) and loose-synchronization update (LSU).

4.5.1 Concurrent Victim Selection (CVS)

To select a victim among many segments, LFSs should visit all dirty segments (e.g., scanning dirty segment bitmap in F2FS) and calculate the GC cost for each segment. This process is protected via a lock (i.e., seglist_lock) to atomically manage the segment bitmap. With a parallel GC technique with multiple GC threads, the contention on the lock increases, leading to a performance bottleneck.

Procedure: To avoid the bottleneck in victim selection, SVM adopts CVS with an atomic test and set operation per segment. We elaborate on CVS using Figure 5. In the figure, initially, DGC_0 has no victims yet, it tries to grab a victim while DGC_1 has a victim segment (i.e., $segment_1$ and its GC cost as 30) selected in a previous scan. Each DGC selects victims with lower GC costs up to a certain number (denoted as M) until iterating through all segments in the dirty segment bitmap. We explain the scenario where M equals one in the example for a simple description.

DGC_0 and DGC_1 concurrently scan the dirty segment bitmap to identify and fetch a dirty segment (①, ❶). Both DGCs recognize that the third segment ($segment_3$) denoted by the third bit in the dirty segment bitmap is dirty and get its GC cost (i.e., 10) from the segment metadata (SM) via loose-synchronization update (LSU) (②, ❷). Since DGC_0 has no victim yet, it tries to grab $segment_3$ as a victim (③). Meanwhile, DGC_1 already has one victim, however, the cost of $segment_3$ (i.e., 10) is lower than that of the selected one (i.e., 30). Thus, DGC_1 also tries to grab $segment_3$ (❸). Therefore, the two GC threads race by trying atomic_test_and_set_bit() to the victim segment bitmap, and DGC_0 wins in this example (④). Then, DGC_0 fetch the segment by adding $segment_3$ to its local victim array (⑤). Since DGC_1 loses in the race (❹), it resumes scanning the dirty segment bitmap from the next segment (❺). By scanning the bitmap, DGC_1 identifies that $segment_6$ is dirty and has lower cost than the current victim (cost of 30). DGC_1 garbs $segment_6$ and replaces the previous selected victim ($segment_1$) with $segment_6$ (❻). DGC_0 and DGC_1 restart this selection process until the iteration of all segments in the file system is finished. By doing this, DGCs can achieve high concurrency in selecting victims. As mentioned, each DGC in ScaleLFS selects M victims, a configurable value, and finding the optimal M can further reduce the overhead of victim selection. We utilize values of M as 0.1% of total capacity (16 segments and 4096 segments for 30 GB and 7.68 TB partitions, respectively) which achieves the highest performance from our empirical experiment.

4.5.2 Loose-synchronization Update (LSU)

Even though CVS provides the concurrent victim segment selection, segment metadata management can be the next bottleneck between DGCs. The metadata of victim segments such as valid page count (VPC) and valid page bitmap (VPB) is atomically managed by a single coarse-grained lock (e.g., sentry_lock in F2FS). However, the lock incurs high contention among DGCs which access and update the metadata of victims when selecting victims and after copying valid pages, respectively. To mitigate this, SVM adopts LSU on the segment metadata, allowing concurrent updates or accesses to each metadata without compromising file system consistency. LSU utilizes an atomic data structure (i.e., atomic bitmap) for VPB and a combination of atomic operations for VPC to update individual metadata independently. However, these individual atomic metadata updates via LSU can lead to two side effects: 1) less-optimal victim selection and 2) false-positive GC read. Figure 6 depicts examples of these side effects.

Side effect #1: Less-optimal victim segment selection. Less-optimal victim segment selection is caused by accessing inconsistent segment metadata (e.g., VPC is outdated). Each VPC can be updated by DGC (e.g., copying victim pages) or flush thread (e.g., writing data pages). During scanning VPCs, VPCs can be updated by the writers. In this case, DGC may

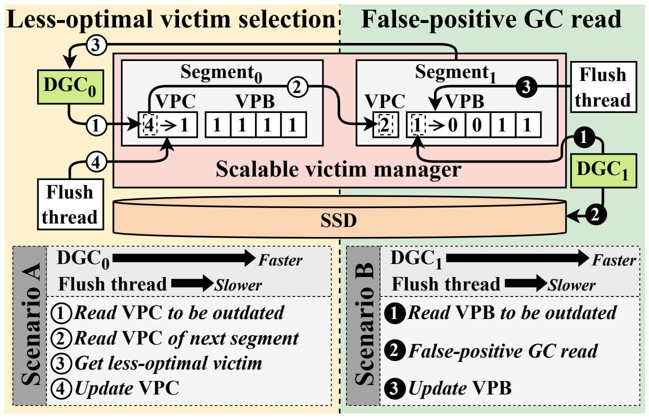


Figure 6: Side effects by loose-synchronization update (`LSU`).

fail to recognize the latest updated `VPC`, potentially leading to the selection of a victim segment which does not have the smallest `VPC` (i.e., less-optimal victim selection). Thus, less-optimal victim selection can increase the GC cost (i.e., write amplification factor [13, 31]) because it fails to minimize the number of valid page migrations. However, this less-optimal victim selection increases negligible write amplification as shown in our empirical results (Section 5.5).

Side effect #2: False-positive GC read. The inconsistent state of `VPB` can induce a false-positive GC read. A `VPB` of a segment is accessed by a `DGC` that occupies the corresponding segment for segment cleaning. At the same time, the flush thread can update the `VPB` when a page is overwritten. Specifically, during the valid page migration of the selected victim segment, a `DGC` identifies valid pages by scanning each bit in `VPB` of the victim segment in an atomic manner. However, during migration, since the flush thread can invalidate the valid page on `VPB` by clearing the corresponding bit, `DGC` may read a page that was previously identified as valid. In this case, a GC read for the invalidated page can occur (i.e., false-positive GC read). As depicted on the right side of Figure 6, if `DGC`$_1$ reads the first bit of `VPB` (❶) before the flush thread clears it (❸), `DGC`$_1$ identifies the first page of segment$_1$ as valid and reads it (❷) during migration. This false-positive GC read can induce additional read from SSD. To measure how often false-positive reads occur, we evaluate their frequency in Section 5.5. The result shows that such reads occur rarely, having a negligible impact on application performance.

Note that `LSU` has a temporary data inconsistency due to the false-positive read and may incur a crash consistency issue due to the independent updates of `VPC` and `VPB`. However, `ScaleLFS` naturally resolves these issues via node-level synchronization and checkpoint mechanisms of the existing LFS, respectively (see Section 4.7).

Analysis of all possible side effect cases: Table 1 lists all possible cases of two side effects. For selecting a victim segment, if `VPC` is outdated, a less-optimal victim segment can be selected (cases 1 and 3). Also, even if `VPC` is up-to-date, when `VPB` is outdated, the selected victim segment can be a less-optimal one (case 2). In this case, on the aspect of GC

Case	Valid page count	Valid page bitmap	Less-optimal victim	False-positive GC read
1	Outdated	Outdated	Yes	Yes
2	Up-to-date	Outdated	Yes	Yes
3	Outdated	Up-to-date	Yes	No
4	Up-to-date	Up-to-date	No	No

Table 1: Analysis of all possible side effect cases.

cost, the optimal victim segment can be selected, however, a false-positive GC read can occur when `VPB` of the victim segment is outdated. The optimal victim segment can be selected only when both `VPB` and `VPC` are up-to-date as in case 4. Meanwhile, as in cases 1 and 2 (even if `VPC` is up-to-date), when `VPB` is outdated, false-positive GC reads can occur. Otherwise, as in cases 3 and 4, false-positive GC reads do not happen.

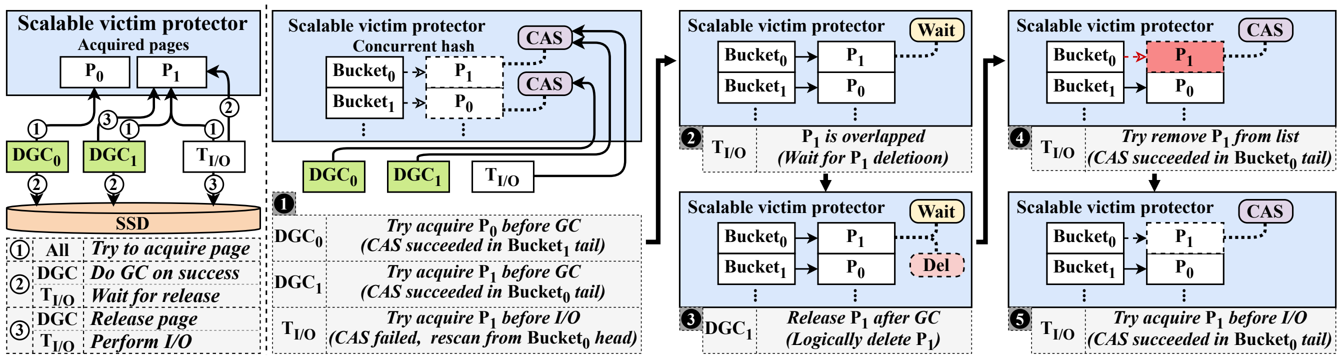
4.6 Scalable Victim Protector (SVP)

The existing LFSs perform the GC procedure at file granularity, and application threads (e.g., direct I/O, trim, and punch hole) can directly access or update the victim pages in the file undergoing GC, which may cause a conflict with the GC thread. To resolve this conflict, the existing LFSs employ a file-level lock for GC (e.g., `i_gc_rwsem`) to serialize the access to the victim file. However, this file-level lock cannot scale GC and limits its concurrency.

Enabling concurrent page-level GC: To eliminate the contention from the file-level GC, we propose a scalable victim protector (`SVP`) which enables a page-level GC procedure while protecting victim pages in a scalable manner. To do this, we devise a concurrent protection technique based on a concurrent hash table. It allows threads to simultaneously access or update the different pages in a file similar to a range lock [17]. Figure 7a shows overall page-level GC with `SVP`. As shown in the figure, during GC, `SVP` can enable threads (e.g., `DGC`s and I/O threads) to access different pages within the same file independently. Specifically, even if victim pages P_0 and P_1 are included in the same file (①), `SVP` allows `DGC`$_0$ and `DGC`$_1$ to access their respective pages, P_0 and P_1, independently (②). Meanwhile, `DGC`$_1$ and the I/O thread ($T_{I/O}$) race on P_1. Since `DGC`$_1$ becomes a winner in this race, `DGC`$_1$ performs GC while $T_{I/O}$ waits for the GC completion (②). After the GC process is completed, `DGC`$_1$ releases P_1, and $T_{I/O}$ can start to perform its I/O operation (③).

Procedure: Figure 7b depicts an example of `SVP` procedure with the concurrent hash table with a linked list. To access target pages, the threads locate their corresponding bucket and scan from head to tail in the linked list within the bucket to check if the target pages are already being used by other threads. If the pages are already in use, the threads wait for the release of the pages (e.g., completion of page migration). Otherwise, they start to insert the pages to reserve them.

For example, as depicted in the figure, three threads (`DGC`$_0$, `DGC`$_1$, and $T_{I/O}$) scan the lists associated with their target buckets (Bucket$_1$, Bucket$_0$, and Bucket$_0$) to reserve their target pages (P_0, P_1, and P_1), respectively. In this example, since the


Scalable victim protector
Acquired pages
P0
P1
DGC0
DGC1
TI/O
SSD
① All Try to acquire page
② DGC Do GC on success
TI/O Wait for release
③ DGC Release page
TI/O Perform I/O
Scalable victim protector
Concurrent hash
Bucket0
Bucket1
CAS
Wait
Del
❶ DGC0 Try acquire P0 before GC (CAS succeeded in Bucket1 tail)
DGC1 Try acquire P1 before GC (CAS succeeded in Bucket0 tail)
TI/O Try acquire P1 before I/O (CAS failed, rescan from Bucket0 head)
❷ TI/O P1 is overlapped (Wait for P1 deletioon)
❸ DGC1 Release P1 after GC (Logically delete P1)
❹ TI/O Try remove P1 from list (CAS succeeded in Bucket0 tail)
❺ TI/O Try acquire P1 before I/O (CAS succeeded in Bucket0 tail)

(a) Overall page-level GC. (b) Procedure of concurrent page-level GC.

Figure 7: Scalable victim protector (SVP) and its procedure ($\mathbf{T_{I/O}}$: I/O thread, $\mathbf{P_X}$: page$_X$).

target pages are available (i.e., the pages are not reserved by any threads), the threads (DGC$_0$, DGC$_1$, and $T_{I/O}$) concurrently perform the compare_and_swap (CAS) [12, 39] operation to insert their pages (P_0, P_1, and P_1) into the list tail of each bucket (❶). Because the tail of Bucket$_1$ for P_0 is not in race, DGC$_0$ succeeds in inserting P_0. Meanwhile, DGC$_1$ and $T_{I/O}$ race on P_1 to reserve it. In this case, since DGC$_1$ becomes the winner, DGC$_1$ inserts P_1 into Bucket$_0$. Meanwhile, $T_{I/O}$ re-starts scanning from the head of Bucket$_0$, discovers the target pages in the bucket, and waits for the release of P_1 (i.e., logical deletion of an entry) (❷). When DGC$_1$ deletes P_1 logically by setting a deletion flag of the page (❸), $T_{I/O}$ discovers the logical deletion and removes this entry from the list by CAS operation (❹). The removed entry is physically deleted when there is no reference to it by read-copy-update (RCU) [17, 27]. After removing the entry, $T_{I/O}$ re-scans from the head to the tail of the linked list of Bucket$_0$ to try to reserve P_1 again. When P_1 is available, $T_{I/O}$ inserts the page via CAS operation (❺). Consequently, SVP can enable a page-level GC procedure by eliminating the file-level contention between DGCs and I/O threads, leading to enhanced GC scalability.

4.7 Data and Crash Consistency

Data consistency: The dedicated page buffer (DPB) or loose-synchronization update (LSU) can incur data consistency issues as described in Section 4.4.1 and 4.5.2, respectively. First, using DPB can lead to a data inconsistency issue between DPB and the page cache. For example, as shown in Figure 8a, after an application thread (T_{APP}) updates a page (P_1) in the page cache (PC), the flush thread (T_F) writes P_1 while DGC simultaneously reads/writes the same page (P_1) from/to the storage device. Second, the separated atomic update of VPB and VPC by LSU can incur a data inconsistency issue between T_F and DGC. For example, as shown in Figure 8b, T_F updates old P_1's bit in VPB to 0 to invalidate old P_1 and try to write new P_1. Simultaneously, DGC reads and tries to write old P_1 due to scanning outdated VPB by T_F via LSU (i.e., false positive). As a result, both DGC and T_F simultaneously try to write P_1.

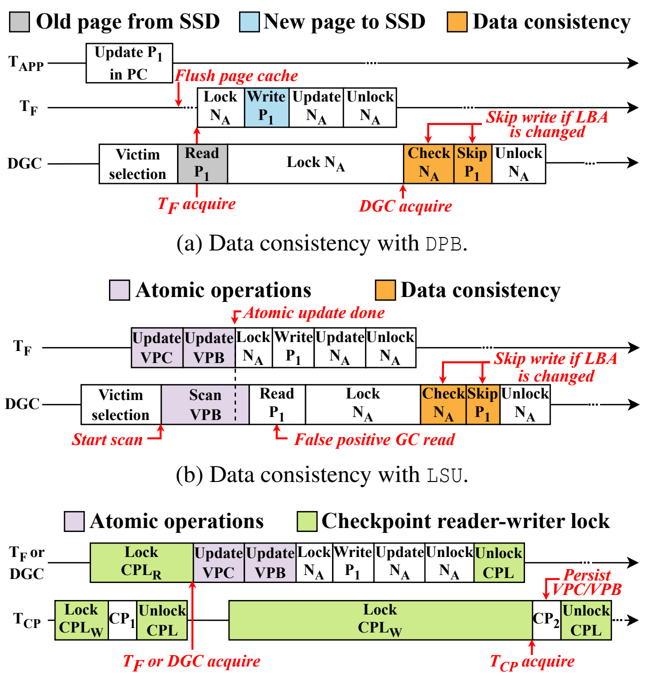
In the first and second cases, T_F and DGC race to write their versions of P_1 simultaneously to the device. To resolve this issue, ScaleLFS uses a simple rule based on the node lock[3] [21] to keep data consistency between T_F and DGC. As shown in Figures 8a and 8b, both T_F and DGC try to acquire the node (N_A) lock to write P_1 to the device and update its LBA in N_A. If T_F acquires the lock earlier than DGC, it writes P_1 with the new data to the device and updates N_A with the new LBA. When DGC acquires the node lock after the lock is released by T_F, it checks whether LBA in N_A has been updated. If the LBA has been updated, DGC can recognize that T_F has written P_1 and skip writing its own P_1. If DGC acquires the node lock earlier than T_F, DGC can recognize that the LBA in N_A is the same as the victim LBA (i.e., source address), meaning that T_F has not yet written P_1. Therefore, DGC writes P_1 to the device and updates its LBA to N_A. Later, T_F writes new P_1 in the page cache to SSD after DGC releases the node lock. By this rule, ScaleLFS guarantees the data consistency with DPB and LSU.

Crash consistency: By leveraging the LFS checkpoint [20, 42] and its reader-writer lock [11, 25] (i.e., cp_rwsem), ScaleLFS guarantees crash consistency even though LSU incurs a temporary inconsistency between VPB and VPC. Under the checkpoint reader lock, the metadata including VPC and VPB is updated simultaneously. On the other hand, when a checkpoint is triggered, the updated metadata is flushed to the device under the checkpoint writer lock, ensuring that VPC and VPB are reflected together, with no further updates to VPC and VPB during this checkpoint. Specifically, as shown in Figure 8c, depending on the order in which the CPL_W is acquired, the first checkpoint can be stored without updates from the I/O thread, while the second checkpoint can include these updates. As a result, since the LFS checkpoint guarantees persisting metadata atomically, LSU does not incur any crash consistency issues. Note that we observe that the checkpoint occurs infrequently due to the roll-forward recovery [23, 24, 29, 43], resulting in minimal checkpoint lock contention.

5 Evaluation

Experimental setup: We use a machine including Intel Xeon E5-2650 CPU (2.2 GHz, 24 physical cores and up to 48 logical cores), 160 GB of DRAM, and a 7.68 TB Samsung 9A3

[3]The node lock protects the node page when reading/writing its associated data page and updating the data page's LBA.



(a) Data consistency with DPB.

(b) Data consistency with LSU.

(c) Crash consistency with LSU.

Figure 8: Data and crash consistency ($\mathbf{T_{APP}}$: application thread, $\mathbf{T_F}$: flush thread, $\mathbf{T_{CP}}$: checkpoint thread, $\mathbf{P_1}$: page$_1$, $\mathbf{N_A}$: node$_A$, **PC**: page cache, **VPC**: valid page count, **VPB**: valid page bitmap, $\mathbf{CP_X}$: checkpoint version$_X$, $\mathbf{CPL_{R/W}}$: checkpoint reader/writer lock).

SSD. For GC experimental setup, we reference the experimental setup in a state-of-the-art study (IPLFS [16]). For example, we reduce the total memory size to 8 GB of DRAM and make a partition of 30 GB for micro-benchmark (FIO), macro benchmark (filebench), and real-work application (MySQL), unless stated otherwise. This prevents the storage device from performing device-level GC, while causing filesystem-level GC to trigger frequently. By doing so, we can isolate and understand the impact of filesystem-level GC on the overall performance. Furthermore, we measure the performance of LFSs at the full SSD capacity (7.68 TB) with 160 GB of DRAM to show how much `ScaleLFS` is effective under frequent device-level GC (Figure 9f).

Comparison: We conduct the experiments using F2FS [22], MAX [25], and P-GC [35] on Ubuntu 22.04 LTS with kernel version 6.0.0. We evaluate MAX in default mode and P-GC in LFS mode as conducted in the studies [25, 35]. We use F2FS in both modes, referred to as F2FS (default) and F2FS-L (LFS mode). With default mode, LFS adaptively performs random writes such as in-place updates or slack space recycling (SSR). `ScaleLFS` performs as LFS mode does without random writes, and utilizes 32 `DGC`s. To evaluate scalability, we modify F2FS and F2FS-L to operate with multiple GC threads, referred to as F2FS-M and F2FS-LM, respectively. Other LFSs such as IPLFS [16] and ParaFS [45] are not evaluated because they require customized SSDs, which is beyond the scope of `ScaleLFS` that targets commodity SSDs.

5.1 Micro-benchmark

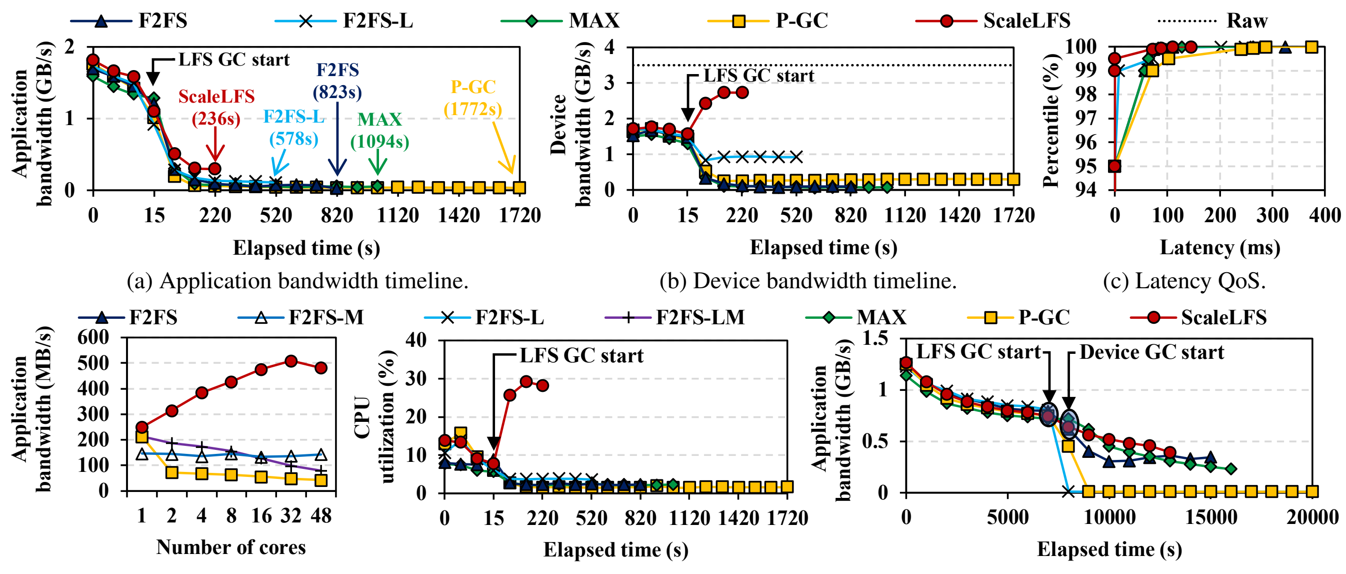
To show the sustained performance of `ScaleLFS`, we use the random write workload from FIO. Specifically, we use each thread per core (48 cores) which submits random writes with a 583 MB file (i.e., a total file size of 28 GB). The total write amount written by the application is 120 GB.

Application-level bandwidth: Figure 9a shows the application-level bandwidth changed as time elapsed. Before 15 seconds, all the LFSs including MAX show similar performance even if MAX focuses on scaling the performance in a clean state. It is because a NAT list lock (`nat_list_lock`) which is adopted at the 6.0.0 kernel incurs a bottleneck in MAX under the clean state. After about 15 seconds, when the space utilization becomes higher and GC is frequently triggered, the write throughput of existing LFSs drops sharply. Accordingly, the existing LFSs (F2FS, F2FS-L, MAX, and P-GC) show low sustained performance and long application execution times (823, 578, 1094, and 1772 seconds, respectively). Though MAX utilizes multiple streams for GC, the bottleneck still comes from the serialized GC procedure. P-GC shows the lowest GC performance even if it utilizes multiple GC threads since they incur high lock contention. As a result, `ScaleLFS` improves/reduces the average bandwidth/execution time by 3.5×/71.1%, 2.4×/58.1%, 4.6×/78.1%, and 7.0×/85.8%, compared with F2FS, F2FS-L, MAX, and P-GC, respectively. The results demonstrate that `ScaleLFS` achieves higher sustained performance than existing LFSs by scaling the GC procedure.

Device-level bandwidth: Figure 9b depicts the timeline of device-level bandwidth measured in the block layer which denotes the bandwidth of I/O submitted to SSD from file systems. `ScaleLFS` utilizes more device-level bandwidth by 15.2×, 2.7×, 19.6× and 8.1× compared with F2FS, F2FS-L, MAX, and P-GC, respectively. Note that `ScaleLFS` achieves 2.9 GB/s, which almost reaches the peak bandwidth (3.4 GB/s) at the device level. As a result, `ScaleLFS` successfully scales the GC procedure by leveraging the potential bandwidth.

Latency QoS: Figure 9c describes the cumulative distribution of latency in the LFSs. In the case of latencies at or below the 95 percentile, all the LFSs show a few microseconds (8–18 us). Meanwhile, for latencies above the 95 percentile, `ScaleLFS` exhibits much shorter I/O latency compared with other LFSs. Specifically, `ScaleLFS` reduces the latency by 99.95%, 99.64%, 99.95%, and 99.96% compared with F2FS, F2FS-L, MAX, and P-GC at the 99th percentile, respectively. This result shows that `ScaleLFS` outperforms existing LFSs significantly in terms of tail latency as well as throughput.

Core scalability: To evaluate the core scalability, we compare `ScaleLFS` with F2FS-M, F2FS-LM, and P-GC. We measure write throughput with various numbers of cores, where the number of GC threads equals the number of cores. The throughput is the average throughput of writing 120GB, in-



(a) Application bandwidth timeline. (b) Device bandwidth timeline. (c) Latency QoS.

(d) Core scalability. (e) CPU utilization timeline. (f) Application bandwidth timeline during device GC.

Figure 9: Micro-benchmark results.

cluding the period before LFS GC is triggered. In Figure 9d, the performance of existing LFSs does not scale or decrease. Especially, the throughput of F2FS-M remains almost the same because SSR is dominantly triggered rather than GC as described in Section 3.2.2. Meanwhile, `ScaleLFS` scales the performance by up to 3.4×, 3.7×, 3.9×, 4.4×, 5.2×, and 5.8×, in the 2, 4, 8, 16, 32, and 48 cores, respectively, compared with the LFSs. The performance in `ScaleLFS` is saturated and slightly decreased at 48 cores. Consequently, `ScaleLFS` can improve multi-core scalability in the GC procedure by using our dedicated and concurrency techniques.

CPU utilization: We measure the CPU utilization with 48 cores under random writes in Figure 9e. In the case of P-GC, we employ two threads as the study does [35]. In `ScaleLFS`, there are 48 application threads and 48 `DGC`s, making a total of 96 threads, which is twice the number of cores. However, application threads are mostly blocked during the GC procedure. As a result, most of the CPU utilization is consumed by 48 `DGC`s. As expected, `ScaleLFS` incurs higher CPU utilization compared with existing LFSs. Specifically, when GC is triggered, CPU utilization of F2FS/F2FS-L/MAX/P-GC is rather and sharply decreased from 7.6%/13.9%/7.4%/15.7% to 2.7%/3.8%/2.7%/2.7%, respectively. It is because F2FS, F2FS-L, and MAX use one core by serialized GC, and the GC threads in P-GC are blocked frequently due to the file-level locking. Meanwhile, CPU utilization of `ScaleLFS` increases by up to 29.1% when GC is triggered. However, `ScaleLFS` reduces the execution time significantly compared with other LFSs. As a result, `ScaleLFS` achieves higher multi-core scalability at the cost of higher CPU utilization.

Impact of device-level GC on filesystem-level GC: For a more practical scenario, we consider the device-level GC where the overhead becomes overwhelmingly expensive for all the LFSs. Before evaluating LFSs, we conduct a preliminary experiment to identify the point at which device-level GC is triggered. In this experiment, we perform random writes on the raw device exceeding the total SSD capacity of 7.68 TB until the throughput sharply drops as described in previous studies [6, 14]. When the written amount exceeds 8 TB, the throughput drops sharply, which we consider the trigger point of the device-level GC. For the evaluation with the LFSs, we set the file size per application thread as 44.7 GB (i.e., a total file size of 2 TB). The total amount written by all the threads is 10.3 TB. By doing so, as shown in Figure 9f, we can expect the trigger point of device-level GC around 8,800–9,740 seconds at which point the written amount exceeds 8 TB.

In the figure, all LFSs show similar throughput changes as time elapses before LFS GC is triggered around 7,500 seconds. After that, in the case of `ScaleLFS`, F2FS, and MAX, the application bandwidth sharply drops due to LFS GC and device-level GC. Meanwhile, `ScaleLFS` exhibits a significantly shorter execution time compared with other LFSs. Specifically, the execution time of `ScaleLFS` is 13,110 seconds while the execution time of F2FS and MAX is 15,340 and 16,490 seconds, respectively. In other words, `ScaleLFS` completes the I/O operations about 37 minutes faster than F2FS and 56 minutes faster than MAX. Unexpectedly, in the case of LFS mode (F2FS-L and P-GC), the device-level GC is not triggered within 20,000 seconds. F2FS-L and P-GC cannot terminate the I/O operations even when their execution time exceeds 20,000 seconds, showing a bandwidth of 11 MB/s. This is because, in the LFS mode, more than 3.5 million segments for the entire capacity should be scanned in a serialized manner. As a result, this slow write speed cannot trigger the device-level GC within 20,000 seconds. Meanwhile, thanks to the concurrent and fast GC victim selection of `CVS`, `ScaleLFS` overcomes this bottleneck even though it is a log-structured approach without in-place update and SSR.

Workload	# of files	File size	# of threads
Fileserver	500	64MB on average	48
Varmail	1000	28MB on average	48
OLTP	28	Data: 1000MB, Log: 200MB	Writer: 3, Reader: 16

Table 2: Workload configurations for macro-benchmark.

5.2 Macro-benchmark

To evaluate ScaleLFS under more realistic workloads, we use filebench [38] with three write-intensive workloads, fileserver, varmail, and OLTP. The configurations for these workloads are listed in Table 2. All workloads begin with more than 95% space utilization. As shown in Figure 10, ScaleLFS improves throughput by up to 30.3%, 40.6%, and 83.1% compared with existing LFSs with fileserver, varmail, and OLTP workloads, respectively. However, this improvement is less than that observed in the micro-benchmark. This is because fileserver and varmail frequently delete the files which generate many free segments and result in infrequent GC. In OLTP, reader and writer threads block each other, resulting in less frequent write and GC I/Os. Despite these factors, ScaleLFS still outperforms other LFSs in more realistic workloads.

5.3 Real-world Application

To evaluate ScaleLFS with the real-world application as shown in Figure 11, we use MySQL with YCSB workload A, B, and update-only [3, 4], which have update ratios of 50%, 5%, and 100%, respectively. We initially insert 6.5 million records (70% of total space occupied) and start workloads with operation counts 2, 5, and 1 million in workloads A, B, and update-only, respectively, with 48 application threads.

YCSB workload A: In Figure 11a, ScaleLFS improves the throughput by up to 3.38× and reduces the execution time by up to 70.4% compared with existing LFSs. As depicted in Figure 11b, ScaleLFS reduces the average/tail latency of reads and updates by up to 72.3%/87.0% and 70.1%/60.9% compared with existing LFSs, respectively. To measure the scalability of GC in ScaleLFS and existing LFSs, we vary the number of cores from 1 to 48 as shown in Figure 11c. The throughput of ScaleLFS reaches 21.9 KIOPS at the 32 cores, and slightly decreases at the 48 cores. Meanwhile, the throughput of existing LFSs remains unchanged or decreases as the number of cores increases. This result demonstrates that ScaleLFS can be also effective in the real-world application.

YCSB workload B: As shown in Figure 11d, interestingly, even in the read-intensive workload, ScaleLFS significantly improves and reduces the average throughput and total execution time by up to 1.37× and 27.3% compared with existing LFSs, respectively. It indicates that the GC procedure considerately impacts the read performance by blocking the read operation due to the file-level locking. In Figure 11e, ScaleLFS decreases the average and tail latency of read/update by up to 23.5%/66.5% and 54.3%/59.8%, respectively. ScaleLFS shows higher GC scalability than F2FS and P-GC in the read-intensive workload as depicted in Figure 11f. The throughput of ScaleLFS is saturated at the 16 cores earlier than that of

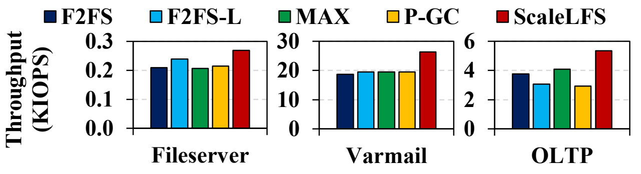


Figure 10: Throughput on macro-benchmark.

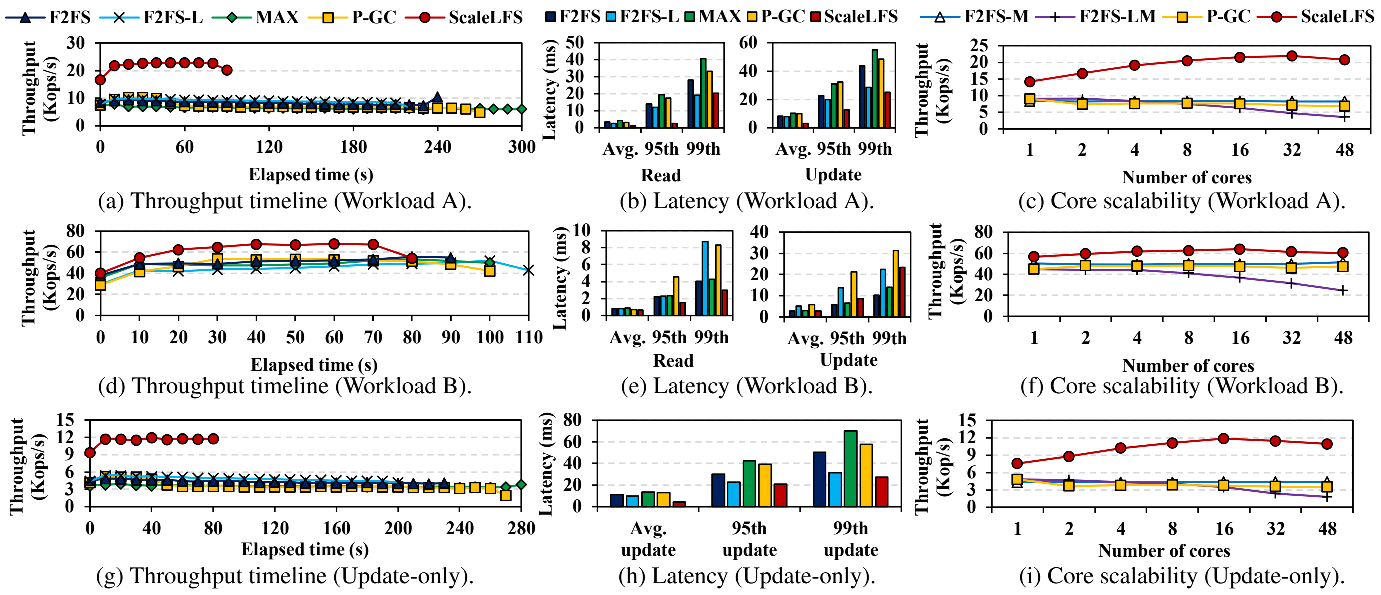
workload A. Since workload B has a lower ratio of updates which reduces the amount of GC works, ScaleLFS with an even smaller number of cores can reach the highest performance earlier. The results reveal that ScaleLFS is also effective on the read-intensive workload.

YCSB update-only workload: Figure 11g shows ScaleLFS increases the throughput by up to 3.22× and reduces the execution time by up to 68.9% compared with the existing LFSs in the update-only workload. In Figure 11h, ScaleLFS reduces the average update latency of 11.0, 9.8, 13.4, and 12.9ms in F2FS, F2FS-L, MAX, and P-GC, respectively, to 4.1ms. Furthermore, ScaleLFS reduces the update latency at the 99th percentile by up to 61% compared with existing LFSs. Finally, ScaleLFS scales GC performance well and shows the largest performance gap among the YCSB workloads in Figure 11i. It improves the performance by up to 4.82× compared with other LFSs at 32 GC threads. This result demonstrates that ScaleLFS exhibits higher performance as the update ratio increases.

5.4 Impact of Individual Techniques

To illustrate the impact of individual techniques in Table 3, we measure the total execution time, total GC time, and time on target locks by each techniques using FIO with the same configuration on 32 cores. We first show the performance of a simple per-core garbage collector (SPGC) without dedicated resources. From SPGC, we incrementally apply scalable victim protector (+SVP), dedicated write stream (+DWS), loose-synchronization update (+LSU), concurrent victim selection (+CVS), and dedicated page buffer (+DPB) in the order of resolving overhead.

In the table, SPGC rather increases the total execution and GC times due to the high lock contention on the coarse-grained lock (i_gc_rwsem). This lock accounts for 21% of total execution time, taking 256.5 seconds out of the total 1221.5 seconds. SVP resolves the file-level lock contention and significantly reduces the total GC time by up to 67.0% compared with SPGC. However, the overhead is moved to the lock contention on curseg_mutex which accounts for 45.7%. DWS eliminates the lock contention via dedicated write streams, and subsequently, the overhead is moved to sentry_lock which incurs 32.8% overhead. LSU resolves the lock contention on sentry_lock, however, seglist_lock becomes another bottleneck by accounting for 31.8%. CVS eliminates the overhead of seglist_lock by concurrently selecting victims. Finally, DPB reduces the GC time by up to 7.8% via dedicated page resources. The result demonstrates that each technique successfully solves the target scalability issues dur-



(a) Throughput timeline (Workload A). (b) Latency (Workload A). (c) Core scalability (Workload A).
(d) Throughput timeline (Workload B). (e) Latency (Workload B). (f) Core scalability (Workload B).
(g) Throughput timeline (Update-only). (h) Latency (Update-only). (i) Core scalability (Update-only).

Figure 11: Performance on real application with YCSB workload A, B, and update-only.

ing the GC procedure.

5.5 Side Effect of ScaleLFS

`ScaleLFS` causes two side effects including less-optimal victim selection and false-positive GC read by `LSU` as described in Section 4.5.2. We evaluate the impact of these side effects using fileserver and MySQL with the YCSB update-only workload. They are the write-intensive, which can have the greatest adverse impact on the two side effects.

Less-optimal victim selection: We measure the write amplification factor (WAF) of F2FS and `ScaleLFS` to show the impact of selecting less-optimal GC victims. WAF slightly increases by 0.3% in fileserver from 1.0036 in F2FS to 1.0040 in `ScaleLFS`. In YCSB, WAF increases by 3.6% from 9.05 in F2FS to 9.37 in `ScaleLFS`. The result shows that `ScaleLFS` can update segment metadata with loose synchronization to scale the GC procedure at the cost of a slight increase of WAF.

False-positive GC read: We measure the number of false-positive reads occurring during GC. In the case of fileserver, there is no false-positive GC read, meanwhile, false-positive GC reads occur 5890 times in the case of YCSB under one million I/O requests. In other words, a false-positive GC read occurs at a rate of 0.006 per I/O request even in a high write-intensive workload. As described in Section 4.5.2, although the false-positive GC reads occur, their frequency is low, and their performance impact is negligible.

5.6 Overhead of ScaleLFS

`ScaleLFS` can disrupt the execution of non-I/O intensive applications due to its high core usage and incur memory overhead due to its additional metadata requirements.

CPU overhead with non-I/O intensive applications: To clarify the impact of `ScaleLFS` in the environment where it shares system resources with non-I/O intensive applications, we measure the execution time of a 48-thread in-memory database (Redis) workload (YCSB-C) running alongside a 48-thread FIO random write workload. `ScaleLFS` increases Redis execution time by 9.8% but reduces FIO execution time by 3.5× compared with F2FS. The results indicate that `ScaleLFS` slightly affects non-I/O-intensive workloads while significantly accelerating I/O-intensive workloads.

Memory consumption: `ScaleLFS` requires additional memory consumption for `DPB` and `SVP`. For `DPB`, its size is 2MB per thread (the same as the segment size), requiring a total of 64MB in 32 `DGC`s. For `SVP`, each hash bucket and hash element require 8 and 36 bytes, respectively. We use 32 buckets and 33 hash elements per file and at most 1444 bytes per file. In the case of 32 files, the total memory footprint is about 64MB. During the evaluation with FIO using 32 `DGC`s, `ScaleLFS` utilizes up to 40MB more memory than F2FS. The amount is slightly less than the theoretical maximum (i.e., 64MB) since `DGC`s may not utilize all `DPB`s, buckets, or hash elements simultaneously.

6 Discussion

`ScaleLFS` utilizes the unoccupied CPU resources created by blocked threads in I/O-intensive applications. However, leveraging these resources at the file system level may not be suitable in environments where non-I/O-intensive applications, such as computation- or memory-intensive applications, run alongside `ScaleLFS`. Since these applications are rarely blocked to wait for LFS-level GC, `ScaleLFS` may not have sufficient unoccupied CPU resources to operate effectively. Consequently, it is important to identify environments where `ScaleLFS` is well-suited. For `ScaleLFS` to be effective, two key conditions must be met: 1) sufficient unoccupied CPU resources, and 2) acceptability of CPU resource usage. For the first requirement, the majority of applications in the en-

File system	Bandwidth	Average latency	Execution time	GC time	Others	`i_gc_rwsem`	`curseg_mutex`	`sentry_lock`	`seglist_lock`
Baseline	213.5 MB/s	615.5 us	577.8s (100%)	530.4s (91.8%)	40.4s (8.2%)	0 (0%)	1.7s (0.3%)	2.3s (0.4%)	0 (0%)
SPGC	98.4 MB/s	1157.7 us	1221.5s (100%)	1156.8s (94.7%)	64.7s (5.3%)	256.5s (21%)	1.2s (0.1%)	4.9s (0.4%)	0 (0%)
+SVP	277.8 MB/s	468.8 us	444.9s (100%)	381.7s (85.8%)	63.2s (14.2%)	0 (0%)	203.3s (45.7%)	4.4s (1.0%)	0 (0%)
+DWS	297.1 MB/s	436.7 us	412.7s (100%)	349.1s (84.6%)	63.6s (15.4%)	0 (0%)	0 (0%)	135.4s (32.8%)	0 (0%)
+LSU	371.6 MB/s	348.5 us	331.9s (100%)	269.2s (81.1%)	62.7s (18.9%)	0 (0%)	0 (0%)	0 (0%)	105.5s (31.8%)
+CVS	476.9 MB/s	270.7 us	246.4s (100%)	188.7s (76.6%)	57.7s (23.4%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
+DPB	509.3 MB/s	254.1 us	236.1s (100%)	174.0s (73.7%)	62.1s (26.3%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Table 3: Performance breakdown and lock overhead (**Baseline**: F2FS-L, **SPGC**: simple per-core garbage collector, **SVP**: scalable victim protector, **DWS**: dedicated write stream, **LSU**: loose-synchronization update, **CVS**: concurrent victim selection, **DPB**: dedicated page buffer).

vironment should be I/O-intensive and frequently blocked while waiting for GC, providing ample idle CPU cycles for ScaleLFS. For the second requirement, it must be acceptable to allocate these CPU resources to ScaleLFS, prioritizing file I/O operations over other tasks.

Based on these criteria, disaggregated storage servers in data centers and file servers are suitable environments for ScaleLFS. First, both types of servers are dedicated to processing file I/O requests. This specialization makes the additional CPU usage by ScaleLFS acceptable in such environments. Second, since these servers predominantly handle I/O-intensive workloads that require frequent GC procedures, the applications running on them are often blocked, leaving sufficient unoccupied CPU resources for ScaleLFS to utilize. Moreover, even in scenarios where non-I/O-intensive applications coexist with ScaleLFS, the impact on these applications is minimal, as detailed in Section 5.6. This result indicates that ScaleLFS can be effectively deployed in a broader range of environments.

7 Related Work

Concurrent data structures: Clever [8] and SEPH [40] propose concurrent hashing optimized for persistent memory (PM) utilizing lock-free or semi-lock-free concurrency control with awareness of PM access granularity. Kogan *et al.* [17] present a design of scalable range locks leveraging a concurrent linked list to scale the virtual memory management. Our study is inspired by these concurrent data structures [8, 17, 40]. In contrast, we focus on designing a scalable GC in LFS leveraging the concurrent mechanisms.

Scalable file systems: MAX [25] scales in-memory data structure access while delivering concurrency-friendly on-disk format in LFS. It targets LFS scalability, meanwhile, it performs a serialized GC procedure. Son *et al.* [36], CJFS [30], and Z-Journal [15] address the scalability issues in journaling file systems with concurrent and per-core schemes. RFUSE [9] enhances the scalability of message communication in user-level file systems with a per-core ring buffer. KucoFS [7] proposes a scalable file system for persistent memory with two-level locking and versioned read. uFS [26] enhances the scalability of user-level file systems by enabling concurrent access without locking and thread load balancing. Our study is in line with these works [7, 9, 15, 25, 26, 30, 36] in terms of scaling file systems. Unlike these studies, we focus on scaling the GC procedure to improve scalability and sustained performance in LFSs.

Improving GC performance in LFSs: ParaFS [45] coordinates GCs in LFS and FTL levels to mitigate the GC overhead. IPLFS [16] adopts infinite address space to remove GC and develops interval mapping to minimize the memory requirement for the LBA-to-PBA translation in FTL. These studies are in line with ScaleLFS in terms of improving the sustained performance in LFSs. However, these studies require hardware modifications with specialized devices, meanwhile, ScaleLFS focuses on a software-based GC approach which can be widely adopted in commodity SSDs. To reduce GC overhead, SFS [28] classifies file blocks based on their update likelihood and writes those with similar hotness into the same log segment. F2FS [22] uses multi-head logging and writes metadata and data to separate logs. P-GC [35] enables a parallel GC procedure but incurs still high lock contention. Our study is in line with these studies [22, 28, 35] in improving the GC performance of log-structure systems. Meanwhile, we focus on scaling the GC procedure via dedicated and concurrent mechanisms as well as the parallel GC process.

8 Conclusion

This paper introduces ScaleLFS, a log-structured file system (LFS) with scalable garbage collection (GC) to achieve higher sustained performance on commodity SSDs. ScaleLFS parallelizes the GC procedure with a per-core and dedicated-resource-based mechanism. Second, ScaleLFS concurrently selects victim segments and updates the metadata of the segments via a concurrent data structure and loose-synchronization updates. Finally, ScaleLFS enables concurrent page-level GC and protection with a scalable data structure. In the evaluation, ScaleLFS demonstrates sustained performance by up to 3.5×, 4.6×, and 7.0× compared with F2FS, a scalable LFS, and a parallel GC scheme, respectively.

Acknowledgments

We sincerely thank our shepherd, Samer Al-Kiswany, and the anonymous reviewers for their invaluable feedback. This work was supported by the National Research Foundation of Korea (NRF) (No. NRF-2022R1A4A5034130) and Korea Institute for Advancement of Technology (KIAT) (No. KIAT-P0012724) grant funded by the Korea Government (Corresponding Author: Yongseok Son).

References

[1] Mysql, 2024. URL: https://www.mysql.com/.

[2] The openssd project, 2024. URL: http://www.openssd-project.org/.

[3] Yahoo cloud servicing benchmark, 2024. URL: https://github.com/brianfrankcooper/YCSB.

[4] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. Analyzing the impact of system architecture on the scalability of oltp engines for high-contention workloads. *Proc. VLDB Endow.*, 11(2):121–134, oct 2017. doi:10.14778/3149193.3149194.

[5] Jens Axboe. Flexible I/O Tester, 2024. URL: https://github.com/axboe/fio.

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/bjorling.

[7] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95. USENIX Association, February 2021. URL: https://www.usenix.org/conference/fast21/presentation/chen-youmin.

[8] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/chen.

[9] Kyu-Jin Cho, Jaewon Choi, Hyungjoon Kwon, and Jin-Soo Kim. RFUSE: Modernizing userspace filesystem framework through scalable Kernel-Userspace communication. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 141–157, Santa Clara, CA, February 2024. USENIX Association. URL: https://www.usenix.org/conference/fast24/presentation/cho.

[10] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new LSM-style garbage collection scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/choi.

[11] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, oct 1971. doi:10.1145/362759.362813.

[12] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *Distributed Computing*, pages 265–279, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[13] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR ’09, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1534530.1534544.

[14] Ping Huang, Guanying Wu, Xubin He, and Weijun Xiao. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2592798.2592818.

[15] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-Journal: Scalable Per-Core journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 893–906. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/kim-jongseok.

[16] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: Log-Structured file system without garbage collection. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 739–754, Carlsbad, CA, July 2022. USENIX Association. URL: https://www.usenix.org/conference/atc22/presentation/kim-juwon.

[17] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387533.

[18] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS*

Oper. Syst. Rev., 40(3):102–107, jul 2006. doi:10.1145/1151374.1151375.

[19] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132770.

[20] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel i/o on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 21–32, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319647.3325828.

[21] Chang-Gyu Lee, Sunghyun Noh, Hyeongu Kang, Soon Hwang, and Youngjae Kim. Concurrent file metadata structure using readers-writer lock. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1172–1181, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3412841.3441992.

[22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[23] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. An efficient order-preserving recovery for f2fs with zns ssd. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 116–122, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3599691.3603416.

[24] Yongmyung Lee, Jong-Hyeok Park, Jonggyu Park, Hyunho Gwak, Dongkun Shin, Young Ik Eom, and Sang-Won Lee. When f2fs meets address remapping. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 31–36, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3538643.3539755.

[25] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 877–891. USENIX Association, July 2021. URL: https://www.usenix.org/conference/atc21/presentation/liao.

[26] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483581.

[27] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, pages 509–518, 1998.

[28] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 12, USA, 2012. USENIX Association. URL: https://dl.acm.org/doi/abs/10.5555/2208461.2208473.

[29] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction support in Log-Structured filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 345–362, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/oh.

[30] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. CJFS: Concurrent journaling for better scalability. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 167–182, Santa Clara, CA, February 2023. USENIX Association. URL: https://www.usenix.org/conference/fast23/presentation/oh.

[31] Jonghyeok Park, Soyee Choi, Gihwan Oh, Soojun Im, Moon-Wook Oh, and Sang-Won Lee. Flashalloc: Dedicating flash blocks by objects. *Proc. VLDB Endow.*, 16(11):3266–3278, jul 2023. doi:10.14778/3611479.3611524.

[32] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. The linear tape file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–8, 2010. doi:10.1109/MSST.2010.5496989.

[33] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A

scalable page cache for multiple solid-state drives. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys'24, page 641–656, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3627703.3629588.

[34] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992. doi:10.1145/146941.146943.

[35] Dongjoo Seo, Ping-Xiang Chen, Huaicheng Li, Matias Bjørling, and Nikil Dutt. Is garbage collection overhead gone? case study of f2fs on zns ssds. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, page 102–108, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3599691.3603409.

[36] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. High-Performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, February 2018. USENIX Association. URL: https://www.usenix.org/conference/fast18/presentation/son.

[37] Diansen Sun, Yunlong Song, Yunpeng Chai, Baoling Peng, Fangzhou Lu, and Xiang Deng. Light-gc: a lightweight and efficient garbage collection scheme for embedded file systems. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, Middleware'22, page 216–227, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3528535.3565246.

[38] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016. URL: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[39] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.

[40] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. SEPH: Scalable, efficient, and predictable hashing on persistent memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 479–495, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/osdi23/presentation/wang-chao.

[41] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[42] Lihua Yang, Zhipeng Tan, Fang Wang, Shiyun Tu, and Jicheng Shao. M2h: Optimizing f2fs via multi-log delayed writing and modified segment cleaning based on dynamically identified hotness. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 808–811, 2021. doi:10.23919/DATE51398.2021.9474006.

[43] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. Ars: Reducing f2fs fragmentation for smartphones using decision trees. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1061–1066, 2020. doi:10.23919/DATE48585.2020.9116318.

[44] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. Orcfs: Orchestrated file system for flash storage. *ACM Trans. Storage*, 14(2), apr 2018. doi:10.1145/3162614.

[45] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. Parafs: a log-structured file system to exploit the internal parallelism of flash devices. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 87–100, USA, 2016. USENIX Association. URL: https://dl.acm.org/doi/abs/10.5555/3026959.3026968.

[46] Zhihui Zhang and Kanad Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. *SIGOPS Oper. Syst. Rev.*, 41(3):175–187, mar 2007. doi:10.1145/1272998.1273016.