CSGC: Collaborative File System Garbage Collection with Computational Storage

Jin Pu[1], Shengan Zheng[2], Penghao Sun[1], Guifeng Wang[1], Xin Xie[1], and Linpeng Huang[1]

[1] School of Computer Science and Engineering,
[2] MoE Key Lab of Artificial Intelligence, AI Institute,
Shanghai Jiao Tong University, Shanghai, China
{grey-hibari, shengan, sunpenghao, wangguifeng, xiex, lphuang}@sjtu.edu.cn

Abstract. Garbage collection (GC) in log-structured file systems (LFS) is known to cause performance degradation, particularly in write-intensive scenarios. Existing approaches, such as in-storage migration and hotness-based grouping, aim to enhance GC efficiency. However, these approaches lack effective host-device collaboration, leading to either excessive communication overhead from inefficient task offloading or severe write amplification due to the log-on-log issue. We present CSGC, a host-device collaborative GC approach that utilizes computational storage device (CSD) to optimize GC efficiency. CSGC uses a pipelined CSD-offloaded migration framework with metadata piggybacking to reduce host-device communication overhead, along with a separate flash translation layer (sFTL) to preserve data hotness and mitigate write amplification. Our evaluations using F2FS and Daisy+ OpenSSD show that CSGC significantly improves GC performance, contributing to up to 3.6× and 1.9× speedup in I/O throughput over vanilla F2FS and IPLFS respectively.

Keywords: Log-structured file system · Garbage collection · Computational storage device.

1 Introduction

Log-structured file systems (LFSs) are widely adopted in modern storage platforms for their ability to optimize write operations by converting random writes into sequential ones [16]. The log-structured write approach aligns well with the characteristics of storage devices, as it significantly reduces the penalties associated with random writes and improves overall performance [13]. However, the inherent design of LFS also necessitates garbage collection (GC) to reclaim space occupied by obsolete data. In file systems like F2FS, GC involves consolidating and erasing large data segments, during which valid data are copied elsewhere. This can result in high write amplification (WA) and severe interference with foreground I/Os, particularly under write-heavy workloads [20]. The efficiency of GC in LFS is therefore essential for maintaining predictable performance and prolonging the lifespan of the underlying storage device.

To improve GC efficiency, LFS groups incoming I/O data into different segments based on their hotness to reduce the amount of valid data in victim segments during GC [7]. Prior researches have primarily focused on either refining hotness separation approaches for higher precision and adaptability [14,15,19,23] or reducing data movement through in-storage migration mechanisms [6, 8, 17]. While these efforts aim to minimize the volume of migrated data, they encounter significant challenges due to the internal flash translation layer (FTL) in SSDs, which also employs log-structured out-of-place updates. This results in the "log-on-log" problem [10,22,25], where two layers of log-structured storage are stacked on the flash media, undermining the effectiveness of the applied optimizations in both hotness separation and data migration. Since the FTL lacks knowledge of the file system's layout and, in particular, hotness-based data grouping strategy, FS-level hotness grouping inside the SSD is disrupted [22]. Likewise, the file system is unaware of the FTL's data organization, causing inaccuracies in hotness separation and excessive, unnecessary data movement during FS-level GC. Effective collaboration between the host and the device is therefore crucial to addressing these challenges and enabling efficient GC with accurate hotness awareness and minimized write amplification.

Computational storage devices (CSDs) [5] offer the potential to overcome the inherent limitations of file system GC by effectively bridging the gap between the file system and FTL. By integrating host computation logic directly into storage, GC tasks can be executed closer to the storage media, allowing for tighter collaboration between the file system and storage device. Despite this promising potential, achieving efficient collaboration between the host and CSD for GC remains a non-trivial challenge. To fully harness the potential of CSDs, the division of labor during GC between the host and storage must be carefully orchestrated [12, 24]. On the one hand, the host is well suited for managing metadata and utilizing cached metadata in memory for rapid request processing. On the other hand, CSD excels at handling I/O-intensive tasks with its short data I/O path. To design an efficient collaborative GC scheme, it is also critical to minimize communication overhead between the two parties while granting in-storage GC the ability to track file hotness accurately. Additionally, the in-storage GC must remain compatible with FTL to ensure that essential storage functions are preserved.

We present CSGC, a host-device collaborative approach to file system GC that leverages CSDs to offload critical GC tasks, minimizing both data migration overhead and host-device communication. CSGC introduces a pipelined CSD-offloaded migration framework that divides the GC process into data migrations and metadata synchronization. I/O-intensive data migration tasks are offloaded to CSDs to shorten the I/O path, while metadata updates are handled by the host to take advantage of readily available runtime metadata in the host memory. This collaborative design allows CSGC to organize the entire GC process into an offload-migrate-update pipeline, enabling parallel processing of different GC stages on the host and CSD. To minimize communication overhead between the two sides, CSGC adopts distinct strategies for data migration and metadata

synchronization. CSGC employs a scatter-gather approach to efficiently collect data for migration, effectively improving bandwidth utilization, especially when dealing with fragmented data. For metadata synchronization, CSGC piggybacks a minimal set of necessary metadata with the migration requests from the host when triggering GC, and returns updated metadata along with the completion message when data migration completes, ensuring metadata consistency while keeping communication overhead minimal. To address the log-on-log issue, CSGC introduces a separate flash translation layer (sFTL), which divides the physical storage into out-of-place update (OPU) and in-place update (IPU) partitions. The OPU partition is managed directly by the file system for file data storage, bypassing the conventional block interface to preserve awareness of data hotness during garbage collection. In contrast, the IPU partition retains the block interface for metadata storage, which is particularly useful for handling in-place metadata updates, preventing unnecessary cascading writes caused by metadata logging.

This paper's main contributions are summarized below:

- We present a host-device collaborative GC approach that fully leverages the strengths of both CSDs and the host through a pipelined CSD-offloaded migration framework.
- We propose a scatter-gather approach for offloading data migration tasks coupled with a metadata piggybacking mechanism, enabling efficient data migration with minimal communication overhead.
- We introduce a separate flash translation layer (sFTL) that partitions physical storage into IPU and OPU regions, allowing the file system to directly manage the OPU partition and resolve the log-on-log issue.
- We implement CSGC in F2FS and a hardware-based SSD evaluation platform. Experimental results demonstrate that CSGC substantially mitigates GC overhead, improving throughput by up to 3.6× and 1.9× compared to the vanilla F2FS and IPLFS respectively.

2 Background and Related Work

2.1 Garbage Collection in F2FS

F2FS [9] is a log-structured file system tailored specifically for flash-based storage devices with various flash-optimized designs, including a flash-friendly on-disk layout, cost-effective index structures, and adaptive multi-head logging strategies. F2FS groups file system blocks into *segments*. Each segment is typed to be either *data* for user data or *node* for inodes or indices of data blocks[3]. F2FS separates data with different hotness by maintaining three log heads for data and node segments corresponding to three temperatures: *hot*, *warm*, and *cold*. The temperature of a block is statically determined based on its type and file attributes when it is allocated from the log head for writes.

[3] Unless explicitly clarified, we use "block" to refer to file system blocks instead of NAND flash blocks in this paper.

GC is triggered when the number of available segments is lower than a threshold and performed in the unit of a *section* (consisting of contiguous segments). A victim section is first selected using the greedy policy in foreground GC or cost-benefit policy in background GC. Valid blocks within the chosen section are then identified, read into host memory, and assigned new storage addresses for migration. The costly identify-read-allocate-write process is repeated for each block in the section. Once blocks have been relocated, F2FS writes a checkpoint to persist the change, and the section becomes free for future allocation.

2.2 Improving GC Efficiency

The migration of valid data during GC causes write amplification (WA) and is the key contributor to its performance overhead; therefore, improving GC efficiency hinges on optimizing data block migration.

Hotness-based grouping is a common strategy employed at the host level to reduce the migration cost. Grouping blocks with similar hotness into the same segment increases the chance that when one block is invalidated, others will soon follow, thus reducing the number of valid blocks during GC and lowering WA [21]. Previous works [14, 15, 23] adopt advanced hotness grouping strategy by using update frequencies, block lifespans, and dynamic group sizing based on statistical models to enhance data separation. However, the log-on-log issue undercuts the effectiveness of hotness separation: the FTL is agnostic of the FS hotness separation and may group data in flash storage against the intended will of the host, undermining the effectiveness of the applied optimizations.

In-storage migration, on the other hand, minimizes host-device data movement by directly migrating blocks within the storage device. Offloading migration tasks to the storage device can alleviate the I/O pressure and cache pollution due to the unnecessary detour through host memory in migration. Works [6, 17] based on ZNS interface [2] extend the interface with in-storage migration commands consisting of source and destination addresses. However, their approaches are restricted by the ZNS interface and incur high communication costs when data blocks are scattered due to lack of awareness of host file system. IPLFS [8] supports a vast LBA space (up to 2^{64} LBAs) in FS and FTL to eliminate FS-level GC by ensuring the FS never exhausts sequentially available addresses during the SSD's lifespan. Yet, it struggles to propagate FS-level hotness grouping into the storage media due to the presence of the FTL, leading to suboptimal GC performance.

In summary, the lack of a host-device collaborative GC approach in current studies results in either a host-centric migration that accounts for hotness but incurs unnecessary host-device data movement overhead or inefficient in-storage migration that neglects FS hotness separation. To address these challenges, we aim to propose a collaborative GC approach that integrates FS hotness awareness with in-storage data migration, minimizing redundant data movements and improving overall GC efficiency.

3 Design

We present CSGC, a host-device collaborative GC approach that offloads data migration tasks to CSD, minimizing data movement overhead and host-device communication. CSGC employs a pipelined CSD-offloaded migration framework, with an offloader in the host and an executor in the CSD, leveraging the hardware advantages on both sides (Section 3.1). The offloader prepares the GC request and offloads it to the executor, which then migrates the data directly within storage and returns only the dirty metadata to the offloader for synchronization. CSGC organizes the garbage collection process into a pipelined sequence of operations, enabling parallel execution of I/O-intensive tasks in the CSD and metadata management on the host (Section 3.2). To minimize communication overhead, CSGC adopts a scatter-gather approach for data migration and a piggybacking mechanism for metadata synchronization. To address the log-on-log issue, a separate FTL (sFTL) is proposed to manage the storage space separately and grant FS direct control over data to be GC'ed while preserving CSGC's capability to uphold hotness separation (Section 3.3).

3.1 Collaborative Migration

CSGC employs a CSD-offloaded migration framework that capitalizes on the complementary strengths of the host and CSD for garbage collection. The host, equipped with extensive metadata caches, efficiently handles allocation and synchronization, while the CSD directly executes migration operations near the storage media, significantly reducing data movement overhead and host processing load. This strategic division of labor ensures that GC operations are both streamlined and highly effective.

Figure 1 illustrates the framework of CSGC, comprising an offloader in the host for GC offloading and metadata synchronization, and an executor in the CSD for data migration. Upon receiving a GC request, the host invokes the offloader ① with a victim segment[4]. The offloader then initializes the GC request ②: it identifies the valid blocks in the victim segment and pre-allocates the destination addresses for these blocks, which can be quickly finished with the help of runtime metadata that are likely to be cached in the host memory[5]. The host proactively packs the allocated addresses, the identifier of the victim segment, and the validity bitmap into a CSGC request, and offloads it to CSD ③ to initiate the in-storage migration process. The GC request is dispatched by the NVMe controller through a circular queue to the executor, which then performs in-storage data migration ④. The executor migrates valid blocks using the provided addresses, significantly reducing data movement latency and alleviating host I/O pressure. Upon completion, CSD returns a completion message ⑤ with the

[4] Here we assume the unit of GC is a segment for ease of narration; the GC of a section is discussed in Section 3.2.

[5] If metadata is not cached, host will fetch it from storage. Our experiment showed that the cache hit ratio is high and the fetch cost is negligible.

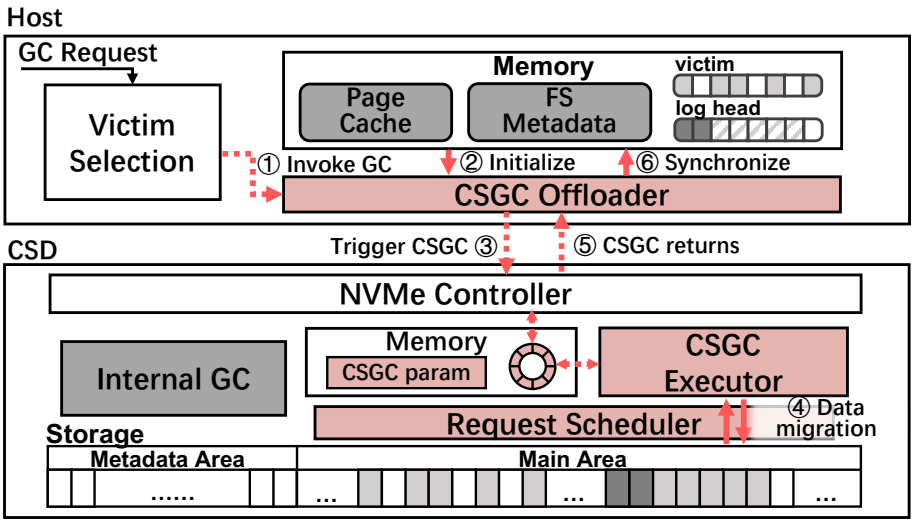

Fig. 1: Collaborative migration framework.

updated metadata to host, and the offloader synchronizes the updates ⑥ with the in-memory metadata to ensure metadata consistency.

Offloading data migration to CSD emphasizes the need for highly efficient in-storage execution. CSGC achieves this by leveraging asynchronous data transfer to overlap data movement with computational tasks, such as block identification and metadata processing. During data block migration, the executor submits data transfer requests asynchronously to the scheduler while simultaneously preparing the next block for processing. This overlap minimizes idle time, ensuring that processing latency is effectively concealed by concurrent data transfer, thus improving the utilization of CSD processors.

To enhance migration performance, a coalescing scheduler is deployed in the CSD to streamline internal data transfer request management during migration. Since data transfer overhead dominates migration costs while computational delays are now mitigated by asynchronous execution, the scheduler prioritizes effective request consolidation. It merges contiguous migration tasks whenever possible to enhance data transfer efficiency. It employs a lazy submission strategy, queuing incoming requests and maintaining a submission window anchored at the queue's head. When the flash data transfer engine is ready, the scheduler consolidates requests within the window, submits the merged operation, and advances the window.

Through the collaborative migration framework, we address the challenge of offloading GC tasks in a manner that capitalizes on the respective strengths of the host and CSD. The host offloader utilizes the readily available metadata in memory to efficiently handle the pre-GC allocation and post-GC metadata updates. The executor in CSD performs data migration near storage using the tailored asynchronous data transfer scheduling, improving data movement effi-

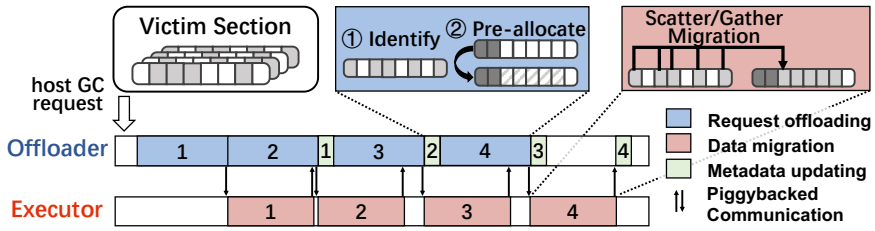

Fig. 2: GC execution pipeline.

ciency by eliminating the detour through host memory from the migration path. Under the current strategic division of GC subtasks, the resources in host and CSD are both effectively utilized, ensuring a more efficient execution of GC operations across the system.

3.2 Pipelined GC Execution

As depicted in Figure 2, CSGC organizes the entire GC operation into a pipeline, effectively overlapping host processing and CSD migration. The GC of a segment undergoes three main phases: request offloading, data migration, and metadata synchronization. We organize these stages into an approximate 2-stage pipeline (the metadata synchronization stage takes little time compared to the whole GC process). To parallelize GC process with the pipeline, CSGC batches the cleaning of the segments in one *section* into a request. By default, a section size of 8 segments is adopted (the default value in F2FS is 1). During execution, the offloader and executor operate in parallel: the offloader prepares requests and synchronizes metadata, while the executor migrates data. This ensures efficient pipeline utilization and resource sharing without blocking the application for too long due to prolonged GC latency with the larger section size.

During pipeline execution, the offloader and executor must coordinate to exchange data migration parameters and ensure metadata synchronization. To minimize host-device communication traffic, CSGC constructs data migration requests in a scatter-gather manner and piggybacks the associated up-to-date metadata with both the request and its completion message. There is thus only a single round trip between the host and the device when exchanging request and metadata, ensuring minimal communication overhead.

Scatter-Gather Data Migration. In the request offloading stage, CSGC employs a scatter-gather (SG) approach to collect data for migration. It consolidates the migration of all valid blocks within a victim segment into a single request to amortize the overhead of request offloading. To enable the SG migration, the offloader provides address descriptors that guide the executor to pinpoint both source and destination addresses for the migration. Specifically, the source address descriptor details the victim segment number along with its validity bitmap, while the destination address descriptor comprises pre-allocated addresses for the valid blocks. These hints are embedded in the migration re-

quest to CSD, allowing the executor to efficiently locate the scattered valid blocks within the segment and migrate them to the designated addresses.

To facilitate SG migration, destination blocks are pre-allocated in a contiguous address range, known as *chunk*. Unlike the per-block allocation method in the original F2FS, our approach avoids the fragmentation typically caused by simultaneous block allocation requests from multiple threads, which hinders the efficiency of SG migration. We develop a chunk allocator that secures blocks of contiguous addresses within the critical section of the allocation process, ensuring uninterrupted allocation of contiguous blocks. This chunk allocator significantly reduces address fragmentation, thereby enhancing the efficiency of SG migration.

Piggybacked Metadata Synchronization. In conjunction with SG data migration, the offloader employs a piggybacked metadata synchronization approach to ensure FS consistency across host and CSD with minimal communication overhead. During GC, the executor requires up-to-date reverse index to map data blocks to their index blocks for index updates to reflect the newly migrated data. To grant the executor access to the latest metadata, the offloader extracts the necessary reverse indexes from the in-memory metadata and piggybacks them with the migration request. This metadata is then transmitted to a designated area in CSD memory for easy retrieval by the executor when needed.

After GC, the offloader must synchronize metadata changed during migration back to the host. To minimize the synchronization overhead, the executor adopts a lazy metadata update strategy. During data migration, the executor adjusts reverse indexes and updates index blocks with new addresses while buffering these changes in the CSD memory. Upon completing the migration, these buffered updates are attached to the completion message and sent back to the host for batch synchronization. This approach effectively leverages runtime metadata in the host's memory to absorb subsequent changes, reducing overall metadata I/O traffic. Since the volume of metadata involved in a single GC request is small, attaching the prefetched and updated metadata to the GC request and completion message imposes negligible overhead. The metadata piggybacking scheme minimizes synchronization overhead while leveraging the host's metadata cache to maintain a consistent file system view between the host and the CSD with minimal resource consumption.

3.3 Separate Flash Translation Layer

The separate flash translation layer (sFTL) introduces a decoupled storage management mechanism to eliminate the redundant logging layer for file data updates within the file system, which accounts for the vast majority of on-disk data. As shown in Figure 3, F2FS divides the logical storage space into two areas: a metadata area for file system metadata blocks and a main area for file blocks, for which in-place updates (IPU) and out-of-place updates (OPU) are applied respectively. Correspondingly, sFTL splits physical storage into an IPU partition, which retains the log-structured management, and an OPU partition, which forgoes FTL-level logging to address the log-on-log issue for the main area.

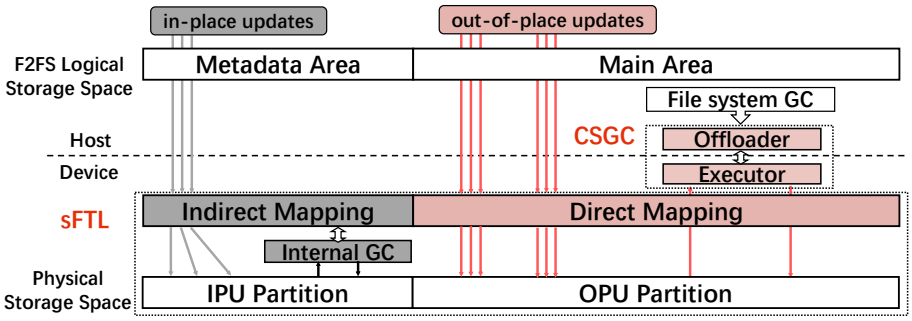

Fig. 3: Decoupled storage management through sFTL.

Specifically, the OPU partition directly exposes physical flash storage in the main area to the file system, bypassing conventional FTL abstractions. This is achieved via sFTL's direct LBA-to-PBA mapping, enabling the physical flash management groups (i.e., NAND blocks) to inherit the hotness characteristics of their logical counterparts in the file system. With this setup, the file system can directly employ its logging methods to the flash storage, managing the NAND pages and NAND blocks through their corresponding storage units in the file system[6]. To reclaim space from dirty NAND blocks, the offloaded FS-level GC replaces the original FTL-level GC. This is equivalent to performing the migration of both logical FS blocks and physical NAND pages simultaneously. The shift of flash management from FTL to FS eliminates redundant FTL logging and reduces write amplification, optimizing both performance and resource utilization. Although the OPU partition exposes an append-only interface similar to raw flash media, the FTL keeps an indirect mapping at superblock level for low-level maintenance such as wear leveling, error correction, etc. This indirection poses negligible runtime computation and memory overhead due to its coarse granularity. Since it is not within the topic of interest of this paper, we leave out the technical details and assume that the file system has direct access to raw flash for ease of narration. To address in-place metadata updates (e.g., adopted in F2FS to avoid cascading writes caused by metadata logging), sFTL includes an IPU partition. The IPU partition is maintained by L2P mapping and internal FTL-level GC, providing a traditional block interface.

sFTL's partitioned management of metadata and main area provides a hybrid interface to the host, eliminating redundant logging layers. This approach not only resolves inefficiencies stemming from the log-on-log issue but also bridges the gap between the file system GC and the underlying FTL, facilitating efficient collaboration between the host and CSD.

[6] We align the NAND page size with F2FS blocks and NAND block size with F2FS segments in our implementation, which is the intended usage of F2FS's flash-friendly layout.

4 Implementation

We integrate CSGC into F2FS (Linux 6.1.54) and introduce a new request flag in the block layer to mark CSGC requests. The NVMe driver is also modified to build customized CSGC commands. We implement the executor along with sFTL in the firmware of Daisy+ OpenSSD [1], a hardware-based SSD evaluation platform featuring a quad-core Cortex-A53 controller and 2GB LPDDR4 DRAM. The OpenSSD uses two 32GB DDR4 DIMMs as the storage backend, thus offering 64GB of storage space. With 7% over-provisioning, the effective storage capacity is 59.5GB. We port the page-level FTL from FEMU [11] to serve as the baseline FTL and simulate flash operations on top of the DRAM backend. The FTL is configured with 8 channels, each channel comprises 2 dies, and a NAND block comprises 512 pages, each 4KB in size. It allocates flash pages in a round-robin fashion across different dies. FTL-level GC is performed in the unit of a superblock with 16 NAND blocks when available superblocks drops below 5%. The sFTL is implemented based on the baseline FTL. It follows the same flash configuration but only applies the L2P mapping and internal GC for the IPU partition. We use one controller core to handle NVMe requests, another for flash emulation and internal data transfer scheduling, and the remaining two for CSGC executor to concurrently process GC requests. The host server is equipped with two Intel Xeon Gold 6240 CPUs and 384GB DRAM. Our implementation includes 4K and 8K lines of code for host-side offloader and device-side executor and sFTL.

5 Evaluation

5.1 Evaluation Setup

We evaluate CSGC against vanilla F2FS and IPLFS using macrobenchmarks (Filebench [18], YCSB [3] on MySQL) and microbenchmarks (fio) to assess its performance across varied workloads. We set F2FS in log-structured mode to ensure consistent GC activity, with background GC deactivated to focus on foreground GC. The logical storage partition size is set to the entire storage capacity. For workloads employing buffered I/O, memory usage is restricted to 8GB via Cgroup, emulating a realistic memory-to-storage ratio. Unless specified otherwise, F2FS uses the baseline FTL, IPLFS employs baseline FTL with interval mapping, and CSGC uses sFTL. For F2FS and CSGC, F2FS section size (granularity of GC) is set to 8 segments. For IPLFS, since the kernel crashes when the section size exceeds 1 segment [4], we set its section size to 1 segment, which should have minimal impact on performance given that IPLFS eliminates FS-level GC. Other configurations are left with their default values.

5.2 Macrobenchmarks

Performance Overview. We evaluate CSGC using fileserver, varmail, YCSB (A/F), and fio benchmarks, with storage pre-filled to 70%-85% capacity to

quickly trigger GC. In fileserver, 4 threads randomly create, append (1MB), write (32KB), read (1MB), and delete files from a set of 54,000 files (1MB each) for 300 seconds. In varmail, 4 threads randomly delete, create, append (32KB), and read files from 860,000 files (64KB each) for 300 seconds. In YCSB-A, 32 threads perform 2M operations (50% reads/50% updates) on 1M records (1KB each). YCSB-F changes the mix to 20% reads/80% read-modify-writes. In fio, 4 threads perform buffered writes (64KB each) totaling 20GB per thread on a 51GB file (85% storage). Two distributions are tested: uniform (fio-uniform) and Zipfian (fio-skewed, $\theta = 1.1$).

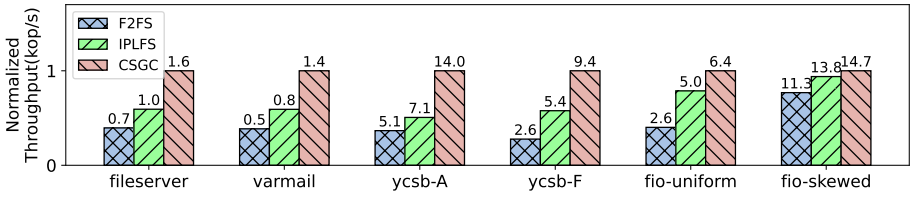

Fig. 4: Performance overview.

Figure 4 shows the overall performance of CSGC compared to F2FS and IPLFS. On average, CSGC outperforms IPLFS by 1.65× and F2FS by 2.76×, with maximum improvements of 1.9× (YCSB-A) and 3.61× (YCSB-F), respectively. CSGC achieves superior performance in YCSB workloads due to frequent small random I/Os, which typically cause severe write amplification and GC overhead. Although IPLFS removes FS-level GC to reduce write amplification, it incurs great overhead in management of interval mapping (e.g. map node compaction) to keep memory usage small, especially with fragmented small I/Os. Moreover, IPLFS relies heavily on frequently sending discard commands to the storage (since it never rewrites the same LBA), which may interfere with normal I/Os and cause performance degradation.

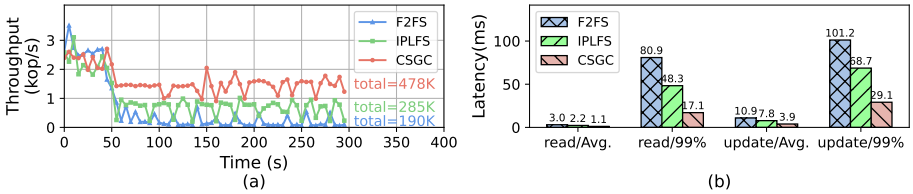

Fig. 5: (a) Throughput during fileserver benchmark. (b) Average and tail latencies for read and update operations in YCSB-A benchmark.

Throughput and Latency. We evaluate the throughput and latency of CSGC in fileserver and YCSB-A benchmarks. Figure 5a shows fileserver throughput over 300 seconds. At about 40 seconds, GC starts and causes significant performance drops in all systems. However, CSGC recovers quickly and sustains higher throughput than F2FS and IPLFS. Figure 5b illustrates average and tail latencies in YCSB-A. By host-CSD collaboration and reduced GC overhead, CSGC improves average latency by up to 2.7× and tail latency by up to 4.7×

over F2FS. It also outperforms IPLFS by up to 1.4× (average) and 1.7× (tail), benefiting from simpler FTL design and improved FS data-hotness awareness.

5.3 Microbenchmarks

We use fio to analyze the impact of storage utilization, GC granularity, and write skewness on CSGC.

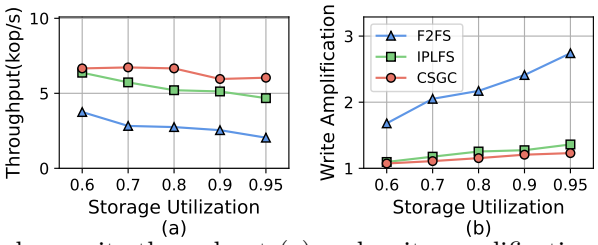

Fig. 6: Fio random write throughput (a) and write amplification (b) under different storage utilization.

Influence of Storage Utilization. With other configuration following default, We vary storage utilization from 60% to 95% and measure throughput and write amplification (WA) for random writes. Figure 6a shows throughput decreases with higher utilization for IPLFS and CSGC, though CSGC consistently achieves higher performance. Figure 6b explains this by showing CSGC's consistently lower WA compared to F2FS and IPLFS. Both IPLFS and CSGC remove one redundant layer of GC, reducing WA significantly versus F2FS. However, IPLFS has to inform the storage of the invalidated data by discard commands, which introduces a delay between the invalidation of data at the FS level and the corresponding invalidation at the FTL level, thus resulting in higher WA compared to CSGC.

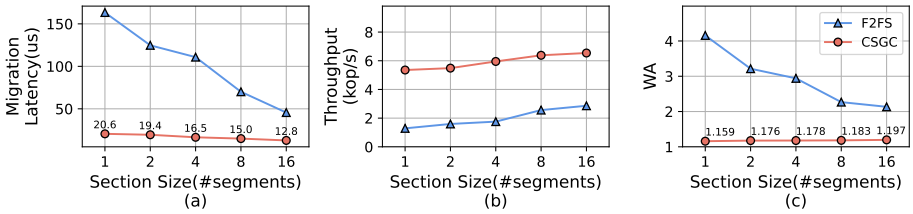

Fig. 7: Fio performance under different section size. (a) Average block migration latency during FS GC. (b) Throughput. (c) Write amplification.

Influence of GC Granularity. We evaluate the impact of section size (ranging from 1 to 16 segments) on throughput and WA for F2FS and CSGC. As depicted in Figure 7b, CSGC consistently outperforms F2FS in throughput across all section sizes. For CSGC, larger sections improve pipeline utilization and resource sharing during pipeline execution, increasing throughput and decreasing average block migration latency (Figure 7a). For vanilla F2FS, throughput also significantly improves with larger sections, primarily due to reduced WA

(Figure 7c), which results from better alignment between FS-level and FTL-level GC granularities [22].

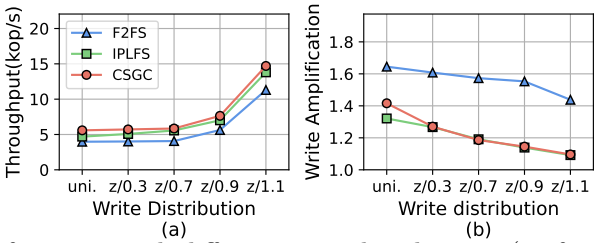

Fig. 8: Fio performance with different write distributions (uniform, zipfian with *theta* = 0.3 - 1.1). (a) Throughput. (b) Write amplification.

Influence of Write Skewness. We evaluate how write skewness affects throughput and WA by varying the write distribution from uniform to Zipfian (θ = 0.3-1.1). Unlike previous fio tests, where four threads perform a fixed total of 80GB writes, here each test runs random writes for 300 seconds. Figure 8 shows that all systems exhibit higher throughput and lower WA as skewness increases, since data becomes more likely to be overwritten, reducing the overhead of migration. Although CSGC achieves higher throughput than F2FS and IPLFS, it shows slightly higher WA compared to IPLFS under the uniform distribution. We attribute this observation to CSGC performing a greater total amount of writes (108GB compared to IPLFS's 85GB) during the 300-second run; initially, WA is low but rises significantly later in the run, leading to higher cumulative WA. This explanation is supported by Figure 6b, where CSGC shows lower WA than IPLFS when the total write amount is constrained to 80GB.

6 Conclusion

In this paper, we present CSGC, a host-device collaborative GC approach that utilizes the strengths of host and computational storage devices to optimize GC efficiency. CSGC integrates a pipelined CSD-offloaded migration framework, facilitating efficient task division between the host and CSD. To minimize communication overhead, CSGC employs a scatter-gather data migration and piggybacked metadata synchronization scheme. Moreover, the adoption of separate flash translation layer (sFTL) grants the file system precise control over storage operations, resolving the log-on-log issue and preserving hotness separation. Our evaluation based on F2FS and Daisy+ OpenSSD shows that CSGC significantly mitigates GC overhead, demonstrating its effectiveness in enhancing system performance and responsiveness in write-heavy scenarios.

References

1. Daisyplus openssd. https://www.crz-tech.com/crz/article/DaisyPlus
2. Bjørling, M., et al.: {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In: USENIX ATC. pp. 689–703 (2021)

3. Cooper, B.F., et al.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
4. ESOS-Lab: Iplfs source code. https://github.com/ESOS-Lab/IPLFS/blob/56bf20fdeee3405330e4d894e3991b0d67f211f2/IPLFS_srcode/fs/f2fs/segment.c#L2639
5. Fakhry, D., et al.: A review on computational storage devices and near memory computing for high performance applications. Memories-Materials, Devices, Circuits and Systems **4**, 100051 (2023)
6. Han, K., et al.: Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In: {OSDI}. pp. 147–162 (2021)
7. He, J., et al.: The unwritten contract of solid state drives. In: Proceedings of the twelfth European conference on computer systems. pp. 127–144 (2017)
8. Kim, J., et al.: Iplfs:log-structured file system without garbage collection. In: USENIX ATC. pp. 739–754 (2022)
9. Lee, C., et al.: F2fs: A new file system for flash storage. In: FAST. pp. 273–286 (2015)
10. Lee, S., et al.: Application-managed flash. In: FAST. pp. 339–353 (2016)
11. Li, H., et al.: The case of femu: Cheap, accurate, scalable and extensible flash emulator. In: FAST. pp. 83–90 (2018)
12. Liu, Y.C., et al.: Rethinking programming frameworks for in-storage processing. In: DAC. pp. 1–6. IEEE (2023)
13. Min, C., et al.: Sfs: random write considered harmful in solid state drives. In: FAST. vol. 12, pp. 1–16 (2012)
14. Oh, S., et al.: Midas: Minimizing write amplification in log-structured systems through adaptive group number and size configuration. In: FAST. pp. 259–275 (2024)
15. Park, H., et al.: Lightweight data lifetime classification using migration counts to improve performance and lifetime of flash-based ssds. In: Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems. pp. 25–33 (2021)
16. Rosenblum, M., et al.: The design and implementation of a log-structured file system. TOCS **10**(1), 26–52 (1992)
17. Tan, Z., et al.: Optimizing data migration for garbage collection in zns ssds. In: DATE. pp. 1–2. IEEE (2023)
18. Tarasov, V., et al.: Filebench: A flexible framework for file system benchmarking. USENIX login **41**(1), 6–12 (2016)
19. Wang, Q., et al.: Separating data via block invalidation time inference for write amplification reduction in log-structured storage. In: FAST. pp. 429–444 (2022)
20. Yan, S., et al.: Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. ACM TOS **13**(3), 1–26 (2017)
21. Yang, J., et al.: Warcip: Write amplification reduction by clustering i/o pages. In: Proceedings of the 12th ACM International Conference on Systems and Storage. pp. 155–166 (2019)
22. Yang, J., et al.: Don't stack your log on my log. In: INFLOW (2014)
23. Yang, L., et al.: M2h: Optimizing f2fs via multi-log delayed writing and modified segment cleaning based on dynamically identified hotness. In: DATE. pp. 808–811. IEEE (2021)
24. Yang, Z., et al.: λ-io: A unified io stack for computational storage. In: FAST. pp. 347–362 (2023)
25. Zhang, J., et al.: Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In: USENIX ATC. pp. 87–100 (2016)