SlimIO: Lightweight I/O Path Design for Write Isolation in FDP-backed In-Memory Databases

Sangyun Lee
Sogang University
Seoul, Republic of Korea
tkddbs9801@sogang.ac.kr

Sungjin Byeon
Sogang University
Seoul, Republic of Korea
sjbyeon@sogang.ac.kr

Soon Hwang
Sogang University
Seoul, Republic of Korea
soonhw@sogang.ac.kr

Jaewan Park
Sogang University
Seoul, Republic of Korea
brian7567@sogang.ac.kr

Joo-Young Hwang
Samsung Electronics Co.
Hwaseong, Republic of Korea
jooyoung.hwang@samsung.com

Junyoung Han
Samsung Electronics Co.
Hwaseong, Republic of Korea
jy0.han@samsung.com

Javier González
Samsung Electronics Co.
Copenhagen, Denmark
javier.gonz@samsung.com

Awais Khan
Oak Ridge National Laboratory
Oak Ridge, TN, USA
khana@ornl.gov

Youngjae Kim*
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

Abstract

In-Memory Databases (IMDBs) are widely used with HPC applications to manage transient data, often using snapshot-based persistence for backups. Redis, a representative IMDB, employs both snapshot and Write-Ahead Log (WAL) mechanisms, storing data on persistent devices via the traditional kernel I/O path. This method incurs syscall overhead, I/O contention between processes, and SSD garbage collection (GC) delays. To address these issues, we propose SlimIO, which adopts I/O passthru to minimize syscall overhead and inter-process I/O interference. Additionally, it leverages Flexible Data Placement (FDP) SSDs as backup storage to avoid performance degradation from SSD GC. Experimental results show that SlimIO reduces snapshot time by up to 25%, increases query throughput by up to 30% during non-snapshot periods, and lowers 99.9%-ile latency by up to 50%. Furthermore, it achieves a write amplification factor (WAF) of 1.00, indicating no redundant internal writes, thus extending SSD lifespan.

CCS Concepts

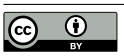
• **Information systems** → **Database recovery**; **Flash memory**.

Keywords

In-memory database, Snapshot, FDP SSDs

ACM Reference Format:
Sangyun Lee, Sungjin Byeon, Soon Hwang, Jaewan Park, Joo-Young Hwang, Junyoung Han, Javier González, Awais Khan, and Youngjae Kim. 2025. SlimIO: Lightweight I/O Path Design for Write Isolation in FDP-backed In-Memory Databases. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767511

*Corresponding author



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/25/11
https://doi.org/10.1145/3731599.3767511

1 Introduction

In high-performance computing (HPC) environments, data-intensive applications increasingly rely on fast and efficient access to transient data. Such data is often short-lived, intermediate data generated during processing but not retained long-term. To effectively manage these transient data without incurring expensive disk I/O, IMDBs provide a high-speed alternative for storage and retrieval during runtime [17, 22, 26]. In-memory databases such as Redis [5], a high-throughput, low-latency IMDB, have emerged as a critical component for such workloads [15]. Its ability to serve as a fast data store, cache, or message broker makes it particularly valuable in HPC workflows [10, 15, 22]. These workflows often prioritize minimizing I/O overhead and enabling rapid data sharing between distributed processes. For example, in computational fluid dynamics (CFD) simulations such as, those used in climate modeling or aerospace design where each simulation timestep can generate large volumes of intermediate data (e.g., pressure and velocity fields). This data must be rapidly exchanged across nodes to advance to the next computation phase. Therefore, storing such transient data in Redis allows for faster inter-process communication compared to traditional file-based I/O, significantly improving overall simulation performance.

As HPC systems evolve, leveraging Redis for real-time metadata access, workflow orchestration, and analytics continues to grow in importance [9, 16, 31]. Moreover, Redis serves as a supporting component across diverse HPC applications, enabling workflow orchestration, state management, real-time log streaming, and metadata indexing [10, 15]. Its role has evolved beyond traditional message brokering and queue management to supporting consistency and persistence in parallel workflows by coordinating task dependencies and maintaining runtime state [22]. In machine learning–driven HPC workflows, Redis reduces processing latency by facilitating real-time data exchange [17]. It also enables state sharing between

different workflow stages, such as scientific simulations, data preprocessing, and distributed training [17]. These examples show Redis's growing role as a middleware layer for workflow orchestration and state management. However, its in-memory nature means all data is lost on failure, and reloading large datasets after a restart can be slow [26].

To mitigate the limitation of losing all in-memory data after a failure, Redis uses two persistence mechanisms together: **Write-Ahead-Log (WAL)** and **Snapshot**. The WAL sequentially logs all write commands to a storage device, while the snapshot mechanism compresses and saves all in-memory objects to the storage device. To prevent unbounded growth of the WAL, Redis creates a new WAL after completing a snapshot. In particular, Redis creates separate processes for query handling (which performs WAL) and snapshotting, enabling true parallel execution and effectively preventing query processing from being blocked during a snapshot. When performing a snapshot, the query-handling process (parent) spawns a snapshot process (child).

However, Redis still suffers from degraded query throughput during snapshot generation due to frequent memory copies and locking issues [24, 29, 30]. Among these issues, the most critical cause is that both processes share the same kernel I/O path. This design causes substantial system call overhead, such as user-space to kernel-space data copying in write() and context switching, which account for approximately 20% of the snapshot time. It also creates I/O contention between the processes. As a result, the snapshot duration becomes longer. Furthermore, SSD garbage collection (GC) exacerbates the snapshot delay, a problem that the kernel I/O path cannot effectively address.

Several prior works have addressed snapshot optimization in IMDBs, including memory dump-based schemes [30], and page table copying optimizations [29]. Relying on the traditional kernel I/O stack, these approaches are inherently limited in their ability to reduce system call overhead and I/O contention between the processes. Furthermore, they do not effectively address SSD GC or enhance performance during non-snapshot periods.

In this paper, we propose SlimIO, a design that mitigates fundamental I/O bottlenecks in the persistence process by reducing system call overhead, eliminating I/O contention between processes, addressing SSD garbage collection issues, and improving performance even during non-snapshot periods. SlimIO leverages I/O passthru [19], a state-of-the-art kernel I/O path based on the io_uring API introduced in Linux kernel 5.1, to reduce system call overhead and establish separate I/O paths for Snapshot and WAL operations. We also adopt Flexible Data Placement (FDP) SSDs, which are supported by NVMe 2.0. The use of FDP enables precise control over the physical placement of the data based on its lifetime, thereby eliminating GC-induced performance degradation. To validate the effectiveness of this design, we implemented SlimIO in Redis v7.4.2, a representative IMDB, while preserving its existing persistence mechanisms. Various evaluations demonstrate that SlimIO reduced snapshot duration by up to 25% compared to the existing design and increased query throughput by 30% during non-snapshot periods. Furthermore, SlimIO reduced the 99.9th percentile latency by half, providing more stable and predictable performance.

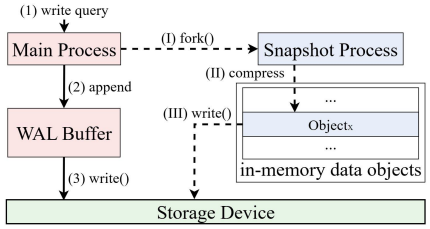


Figure 1: Concurrent Processing of Snapshot and WAL

2 Background

2.1 Persistence Mechanisms of Redis

Redis employs two persistence mechanisms in combination to protect against the volatility of DRAM [2]:

- **Write-Ahead-Log (WAL)**: Sequentially logs all write queries processed by Redis and stores them on storage device.
- **Snapshot**: Stores a compressed copy of the entire in-memory dataset to storage device.

The WAL is a sequential log that records all write queries processed by the Redis server to a storage device. To write data to the WAL, Redis employs a *Periodical-Log* policy: each WAL entry is first placed in a user-level write buffer and is flushed to storage when the system becomes idle. If the system does not become idle within a predefined time threshold, the flush is forcibly triggered.

Snapshots can be generated in two ways. First, to prevent unbounded WAL growth, Redis automatically creates a snapshot when the WAL reaches a certain size, which we refer to as a *WAL-Snapshot*. When a new WAL-Snapshot is generated, the previous WAL and WAL-Snapshot are deleted [2]. Second, Redis administrators can manually create a snapshot for purposes such as master-slave data transfer or point-in-time backups. We refer to this as an *On-Demand-Snapshot*. On-Demand-Snapshots can be scheduled to occur periodically—e.g., daily or weekly—or created manually to preserve the dataset at a specific moment, such as before a server release or testing. A WAL-Snapshot and an On-Demand-Snapshot cannot be created concurrently, and at most one of each type may exist at any given time [7].

2.2 Snapshot Process and Query Performance

When creating a snapshot, Redis spawns a child process using fork(), which is responsible for performing the snapshot generation. This approach delegates the high-overhead I/O operations involved in snapshot to the child process (snapshot process), allowing the main process to continuously handle query requests, thereby enabling parallel processing [25, 29].

Figure 1 illustrates a scenario where WAL operations and snapshot generation are executed in parallel. The Redis main process handles client requests and, for write query, temporarily stores the data in a user-level write buffer (WAL buffer). The buffered data is then flushed to disk using write() when the system becomes idle or a time threshold is reached. Meanwhile, as illustrated in the Figure 1, the snapshot process iterates over the Redis objects in memory. For each object, it performs compression and subsequently writes the compressed object to disk via write().

During snapshot generation, Redis experiences memory pressure and degraded query throughput. When a write query arrives during snapshot generation, a lock must be acquired on the memory region,

and the existing memory must be copied to support the snapshot process [25, 29]. During this time, both the main process and the snapshot process are stalled until the memory copy is completed and the lock is released, resulting in increased in-memory storage pressure and reduced query throughput [30].

Table 1: Performance Degradation and Increased Memory Usage During Snapshot Generation

		Requests per Second	Peak Memory Usage (GB)
EXT4	WAL Only	59512.38	26
	Snapshot&WAL	42885.10	51
F2FS	WAL Only	61327.40	26
	Snapshot&WAL	43111.97	52

Table 1 shows the query throughput and memory usage during periods without snapshots (WAL Only) and during periods when snapshots occur (Snapshot&WAL). The experiments were conducted by mounting EXT4 and F2FS file systems to evaluate the impact of writing WAL and snapshot data to storage device. Detailed descriptions of the experimental setup and workloads are provided in Section 5. During snapshot generation, memory usage increases regardless of the file system, and query throughput decreases by 28% and 31% under the EXT4 and F2FS environments, respectively. If the snapshot duration increases, both the period during which in-memory storage is under pressure due to memory copy and the period of query throughput degradation caused by memory copy and locks are prolonged. Therefore, shortening the snapshot duration is crucial for mitigating these issues.

2.3 Flexible Data Placement SSD

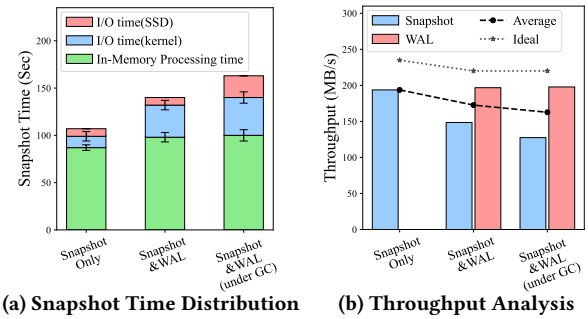
In conventional NVMe SSD, data with different lifetimes are mixed within a single physical block. When lifetimes are mixed, a process calls GC occurs [20, 35]. During GC, valid data within the physical block must be copied before the block can be erased, resulting in additional internal write operations within the SSD due to these copy processes. As a result, WAF, an indicator of additional internal write operations within the SSD, increases. An increase in WAF causes latency in all write operations from the host application, negatively impacting not only WAL operations but also snapshot generation. As a solution to this problem, FDP SSD [34] have been introduced. FDP SSDs allow the host to control the physical placement of data by specifying a placement hint called the PID (Placement Identifier) in NVMe write request [8, 12, 33]. When NVMe write command attached with 32bit PID comes to FDP SSD, firmware groups data with the same PID within the same physical region at the Reclaim Unit(RU) granularity. By assigning the same PID to data with the same lifetime, the host enables data within the same RU to be erased without additional copying during GC, thereby reducing the WAF.

3 Motivation

3.1 I/O Path Bottleneck in Snapshots

We analyzed cases where Redis snapshot duration increases. Therefore, we measured and compared snapshot duration, snapshot throughput, and WAL throughput. These metrics were evaluated under three different scenarios. The experimental details are provided in the Section 5.

- **Snapshot Only**: On-Demand-Snapshot generation occurs without WAL operations.



(a) Snapshot Time Distribution **(b) Throughput Analysis**

Figure 2: Analysis of Snapshot Duration and Throughput

- **Snapshot & WAL**: Snapshot and WAL operations occur concurrently.
- **Snapshot & WAL (under GC)**: Snapshot and WAL operations occur concurrently while the SSD experiences GC pressure.

Figure 2 illustrates the results. An increase in snapshot duration was observed in both the Snapshot & WAL and Snapshot & WAL (under GC) scenarios. We identified that this increase is caused by four factors occurring within the kernel I/O path.

3.1.1 Traditional Kernel I/O path has high syscall overhead. When using the POSIX I/O interface, significant overhead arises from data copying between user-space and kernel-space as well as context switches during system call invocations [3, 36]. As a result, snapshot processing fails to achieve the ideal scenario in which in-memory processing tasks such as index search, compression, and memory copying are fully overlapped with kernel and SSD I/O times. As shown in Figure 2a (Snapshot Only), approximately 15% of the snapshot duration is spent within the kernel I/O path. Consequently, as depicted in Figure 2b (Snapshot Only), the throughput is about 15% lower than the ideal throughput.

3.1.2 Traditional Kernel I/O path has a scalability problem. Existing Linux file systems exhibit scalability problems, even when parallel processes write to different files, due to contention over the same lock [27]. For example, the widely used journaling file system EXT4 experiences contention when acquiring the journaling lock [18, 21]. Although F2FS offers relatively better scalability compared to other file systems [27], it still suffers from scalability problems. These scalability issues can be observed when snapshot and WAL operations occur concurrently.

Table 2: CPU Usage of File System Write Path in Snapshots

	CPU Usage of F2FS in the Snapshot Process
Snapshot Only	11.53%
Snapshot&WAL	13.61%

Table 2 shows the CPU cycle utilization by the snapshot process within the file system. Considering that this CPU consumption occurs in the control path rather than during actual data write operations, it represents a non-negligible overhead. Furthermore, as shown in Figure 2b (Snapshot Only), the average throughput is about 15% lower than the ideal throughput, whereas in Figure 2b (Snapshot & WAL), the average throughput is approximately 20% lower than ideal.

3.1.3 Traditional Kernel I/O path ignores per-process write patterns. As described in Section 2.2, the WAL operation writes large amounts

of buffered data to disk via a single write() call when the time threshold is reached. Meanwhile, the snapshot process iterates over Redis objects in memory, compresses each object, and frequently writes the compressed objects to disk using multiple write() calls. The kernel I/O path does not recognize these per-process write patterns, causing the snapshot process—with its more frequent write() calls—to be blocked more often. As shown in Figure 2b, because the snapshot process is blocked more frequently, the snapshot throughput is approximately 30% to 45% lower than the WAL throughput. Moreover, while the WAL throughput remains stable even under SSD GC, the snapshot throughput degrades. Consequently, as shown in Figure 2a, when snapshot and WAL operations occur simultaneously, frequent blocking of the snapshot process leads to increased time consumption in the kernel I/O path.

3.1.4 Traditional Kernel I/O path lacks sufficient mechanisms to eliminate SSD GC. Figure 2 (Snapshot & WAL (under GC)) illustrates that when SSD GC occurs during snapshot operations, the previously observed issues are exacerbated, leading to increased time consumption on the SSD. As discussed in Section 2.3, SSD GC occurs because data with different lifetimes reside within the same physical region of the SSD [11, 28]. On-Demand Snapshots generally have long lifetimes, as they are generated either automatically at long intervals (e.g., once per day) or manually to preserve data at specific points in time. In contrast, WAL-Snapshots and WAL data have much shorter lifetimes, especially in write-intensive workloads, because the creation of a new WAL-Snapshot invalidates all previous WAL-Snapshots and WAL data. While the latest NVMe protocols enable FDP SSDs to place data with different lifetimes into separate physical regions at the user-level, the traditional kernel I/O path lacks sufficient mechanisms to support these advanced NVMe features. Even though some of the latest file system updates have introduced support for such features, they still fail to resolve the aforementioned issues.

3.2 I/O Passthru for Efficient Write Isolation

io_uring is an asynchronous I/O API introduced in Linux kernel 5.1 [3]. It utilizes a pair of ring data structures called the Submission Queue (SQ) and Completion Queue (CQ). The SQ holds I/O requests submitted by the application, while the CQ contains the completion results of those requests. Applications submit commands to the SQ and poll the CQ to retrieve completion notifications. Both SQ and CQ are shared between user space and kernel space, reducing system call overhead by minimizing buffer copy operations between these spaces [3, 14]. Notably, when io_uring is created in SQPOLL mode, system calls are not required for I/O submission. In this mode, an additional kernel thread continuously polls the SQ to check for new requests. Thus, io_uring enables I/O requests to asynchronously reach the file system with minimal system call overhead.

I/O passthru [19] is a state-of-the-art kernel I/O path that has been upstreamed into the Linux kernel. It operates based on the io_uring API and internally bypasses kernel I/O layers such as the page cache and file system. Notably, I/O passthru supports advanced NVMe commands that are not available in the traditional kernel I/O path, enabling it to address issues related to SSD GC.

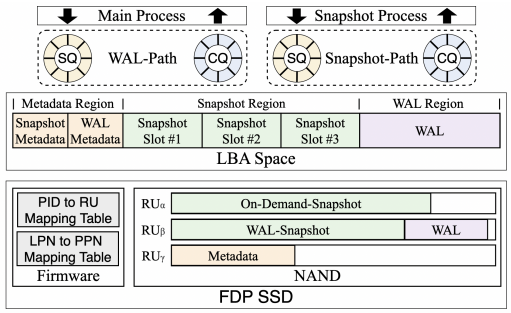


Figure 3: The overview of the SlimIO architecture

This study aims to address the issue of increased Redis snapshot duration caused by the limitations of the traditional kernel I/O path and GC overhead in conventional NVMe SSDs. To this end, we introduce a lightweight I/O path based on the I/O passthru mechanism utilizing the io_uring API in Redis, and simultaneously adopt state-of-the-art NVMe FDP SSDs.

4 Design of SlimIO

SlimIO adopts I/O passthru for the following three reasons.

- I/O passthru is built on io_uring, which significantly reduces system call overhead compared to traditional blocking APIs and libaio [14, 19]. This design leads to faster snapshot generation and higher query throughput during non-snapshot periods. Additionally, io_uring provides tunable configurations depending on the workload, allowing for further performance optimization.
- Second, although SPDK [36] could serve as an alternative, it requires the use of a user-space library that maps the entire PCI bar to a single application, which poses challenges in multi-tenant environments [19]. In contrast, I/O passthru does not require such mapping and is already upstreamed into the mainline Linux kernel.
- Third, many I/O schedulers operate based on priority, favoring synchronous writes such as WAL operations, which may cause snapshot writes to be deprioritized [1, 18]. I/O passthru bypasses the I/O scheduler [4, 19], avoiding such prioritization issues.

4.1 Snapshot–WAL Separation via I/O Passthru

As presented in Section 3.1, we observe that using the traditional kernel I/O path by two processes can cause scalability problems and unawareness of per-process write patterns. We attribute these issues fundamentally to the fact that the two processes share the same kernel I/O path, resulting in contention due to locking and scheduling. Therefore, we separate the I/O path for WAL and Snapshot via I/O passthru. Figure 3 presents an overview of the SlimIO architecture. As shown in Figure 3, the main process and the snapshot process use separate I/O paths to write to the Logical Block Address (LBA) space.

When the Redis engine starts, it initializes a SQ and CQ for WAL operations using the I/O passthru. We refer to this as the WAL-Path. A dedicated CQ handling thread is also spawned to process completions. SlimIO preserves the original Redis WAL logging policy without modification.

When a snapshot process is spawned, another pair of SQ and CQ is initialized within the process using I/O passthru. We refer to

Table 3: Performance evaluation using Redis-benchmark

		WAL Only		WAL&Snapshot		Average RPS	Snapshot time (sec)	SET p999 (ms)	SSD WAF
		RPS	Mem Usage (GB)	RPS	Mem Usage (GB)				
Periodical-Log	Baseline	57481.86	25.99	42300.51	52.27	47993.20	148	5.103	1.14
	SlimIO	75675.66	25.99	42516.72	51.99	55042.87	110	2.351	1.00
Always-Log	Baseline	21415.85	25.99	16418.87	51.98	19043.80	139	7.822	1.24
	SlimIO	33127.81	25.99	25541.80	51.99	31407.03	109	3.343	1.00

Table 4: Overall Evaluation with YCSB-A Workload

		WAL Only		WAL&Snapshot		Average RPS	Snapshot time (sec)	SET p999 (ms)	GET p999 (ms)
		RPS	Mem Usage (GB)	RPS	Mem Usage (GB)				
Periodical-Log	Baseline	65120.76	27.13	53774.30	54.26	61695.78	253	0.711	0.673
	SlimIO	74911.06	27.13	56239.39	54.26	68244.45	225	0.635	0.577
Always-Log	Baseline	6234.89	27.13	4987.45	54.26	6191.70	239	2.105	2.091
	SlimIO	12536.86	27.13	10285.05	54.26	12028.85	224	0.950	0.933

this path as the Snapshot-Path. A dedicated CQ handling thread is also created to process completions. Therefore, Snapshot-Path is configured in SQPOLL mode, where a kernel thread continuously polls the SQ, eliminating system call overhead. All snapshot data, including compression algorithms and compression ratios, remain fully compatible with Redis's original I/O module and are preserved without modification.

4.2 LBA Space Management and Recovery

A key challenge of I/O passthru is that, by bypassing the file system and writing directly to the LBA space, it necessitates an explicit mechanism for managing the LBA space, which poses a significant challenge for its adoption. Fortunately, the persistence mechanisms of most IMDBs, including Redis, rely on sequential rather than random writes, making LBA space management straightforward.

To prepare for potential failures during snapshot generation, Redis's existing persistence mechanism deletes old data only after a new snapshot has been successfully completed. To maintain this same mechanism in an I/O Passthru environment, we manage the LBA space by partitioning it into three regions: a WAL Region, a Snapshot Region, and a Metadata Region.

In the WAL Region, WAL entries are written sequentially. The previous WAL is only deallocated after a new WAL-Snapshot generation is successful. The Snapshot region is dynamically composed of two slots for WAL-Snapshots and On-Demand-Snapshots, along with a separate Reserve Slot to handle failures. As mentioned in Section 2.1, since snapshot generations cannot occur concurrently, a single Reserve Slot is sufficient. Using three slots, a new snapshot is always safely written to the Reserve Slot first. Once snapshot generation succeeds, the reserve slot is promoted to a valid snapshot slot, and the previous snapshot slot is deallocated and converted back into the Reserve Slot. All state information for these processes—such as the current WAL position and the roles of each snapshot slot—is recorded in the Metadata Region, ensuring the consistency and reliability of the entire LBA space.

Recovery Procedure. The recovery procedure is as follows: First, when the Redis engine starts, the entire metadata is read to determine the boundaries of the Snapshot Region and the WAL Region. Second, based on the metadata, either the WAL-Snapshot or On-Demand-Snapshot is loaded into memory to restore data, as requested. Third, if the recovery was based on a WAL-Snapshot, Redis then reads the WAL to restore any subsequent data changes to memory. This mechanism is identical to the existing Redis persistence recovery mechanism.

4.3 Snapshot–WAL Separation on FDP SSD

SSD GC increases the duration of snapshots. To mitigate this GC overhead in Redis, we propose using FDP SSDs, which are capable of data placement, as backup storage for SlimIO. Using I/O passthru, the PID field can be passed along with the NVMe write commands [19]. We leverage this to assign different PIDs to data with different lifetimes (e.g., WAL = 1, On-Demand-Snapshot = 2). Figure 3 illustrates the internal state of the FDP SSD, where data with different lifetimes are separated into different RUs. Since the FDP SSD performs GC at the RU granularity, the valid data copy during GC is eliminated, significantly reducing GC overhead.

5 Evaluation

5.1 Experimental Setup

We implement SlimIO in Redis v7.4.2 and evaluate it on Linux v6.7.9 using the following FDP device and workloads.

FDP Emulator. For FDP SSD, we use the FDP emulator based on FEMU [23] v9.0. We emulated a 180GB FDP SSD with 12 cores and 55GB of DRAM. Our host environment is equipped with dual Intel Xeon Gold 5218R CPUs and 377GB of DRAM. The emulated FDP SSD features 8 channels with 8 dies per channel, a NAND page size of 4KB, and an RU size of 1GB; the NAND page read latency, write latency, and block erase latency are set to 40µs, 200µs, and 2ms respectively, all based on FEMU's default settings. The FDP SSD supports 8 PIDs.

Workloads. We use two workloads: one from the official Redis benchmark [6] and another from the YCSB-A workload of the YCSB benchmark [13].

For the Redis benchmark workload, we use the default settings with 50 concurrent clients. The key range is set to 5.3 million, with 8-byte keys and 4096-byte values. Each experiment consists of 28 million SET operations and is repeated five times. An On-Demand-Snapshot is generated once at the end of each repetition. The WAL size for triggering a WAL-Snapshot is configured to approximately 50–55 GB, allowing two WAL-Snapshots per repetition. In total, 15 Snapshots (each 20 GB) are generated across five repetitions, and the final results are reported based on all repetitions. The experiments in Sections 2.2 and 3.1 also use the Redis benchmark workload. In

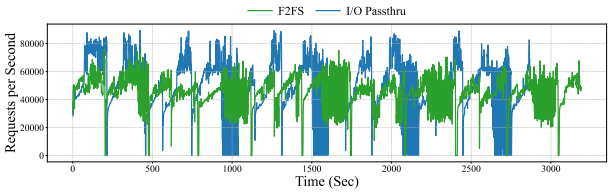


Figure 4: Baseline vs SlimIO without FDP

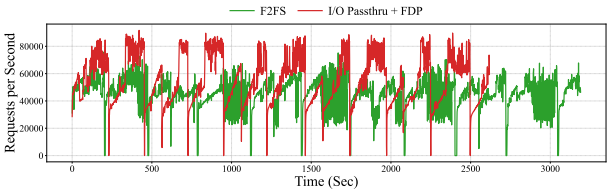


Figure 5: Baseline vs SlimIO

Section 2.2, however, the experiment runs once without generating an On-Demand-Snapshot.

For the YCSB benchmark, we use the YCSB-A workload with 8 threads, 8-byte keys, and 2048-byte values, comprising 9 million records and 115 million operations with a 0.5:0.5 GET-to-SET ratio. Each experiment runs once without generating an On-Demand-Snapshot. The WAL-Snapshot trigger size is set to 50–55 GB, The WAL size for triggering a WAL-Snapshot is configured to approximately 50–55 GB, allowing two WAL-Snapshots per repetition, producing two 20 GB WAL-Snapshots per run.

For comparison with SlimIO, we used a conventional NVMe SSD–based F2FS without FDP support as the baseline I/O stack, applying the same NAND-related configurations as those for the FDP SSD. The I/O scheduler of baseline was set to 'none' [32].

By default, Redis employs the *Periodical-Log* policy, but it also offers an *Always-Log* policy. Unlike the *Periodical-Log* policy, the *Always-Log* policy writes data to disk immediately upon write queries without buffering. Performance metrics were measured separately for each logging policy.

5.2 Overall Evaluation

We evaluate the overall performance of SlimIO, measuring Requests per Second (RPS), memory usage, snapshot time, and 99.9%-ile SET latency for both workloads. For the Redis benchmark, we additionally measure SSD WAF, as On-Demand-Snapshots can trigger SSD GC. For the YCSB-A workload, we also report 99.9%-ile GET latency. Tables 3 and 4 present the results for the Redis benchmark and YCSB-A workloads, respectively.

RPS Results. Under the *Periodical-Log* policy, SlimIO demonstrated improved RPS compared to the baseline across both workloads. For the Redis benchmark workload (Table 3), in the WAL Only phase, SlimIO achieved an RPS of 75,675, which is approximately 32% higher than the baseline's 57,481. The average RPS also improved by about 15%, with SlimIO achieving 55,042 compared to the baseline's 47,993. Even in the YCSB-A workload, which has a relatively low write ratio, smaller value sizes, and no SSD GC, SlimIO maintained performance advantages. As shown in Table 4, its RPS was 74,911 in the WAL Only phase, about 15% higher than the baseline's 65,120, and the average RPS of 68,244 was a roughly 10% improvement over the baseline's 61,695. However, during the WAL & Snapshot phase, there was a negligible performance difference between SlimIO and the baseline for both workloads. This is because the drop in RPS during a snapshot is primarily caused by memory copying and lock acquisition overhead resulting from the fork()'s Copy-on-Write policy [25, 29].

Under the *Always-Log* policy, the performance benefits of SlimIO become even more pronounced. For the Redis benchmark workload (Table 3), the RPS in the WAL Only phase for SlimIO was 33,127, an impressive 54% increase over the baseline's 21,415. The average RPS also increases by approximately 60%, from 19,043 (baseline) to 31,407 (SlimIO). A similar trend was observed with the YCSB-A workload (Table 3). Here, the baseline's average RPS was a mere 6,191, whereas SlimIO achieved 12,028—a remarkable improvement of about 95%.

Snapshot Time Results. For the Redis benchmark workload, SlimIO completed snapshot generation in 110 seconds, which is approximately 25% faster than the baseline. While the Redis benchmark involves larger value sizes, the YCSB-A workload includes a larger number of smaller values, leading to increased compression time and overall longer snapshot durations. For the YCSB-A workload, SlimIO completed snapshot generation in 225 seconds, achieving a 10% improvement over the baseline.

Latency Results. Under the *Periodical-Log* policy, SlimIO reduced tail latency in both workloads compared to baseline. For the Redis benchmark workload, SlimIO achieved a SET tail latency of 2.351ms, which is approximately 50% lower. This workload involves 50 concurrent clients, triggers SSD GC, and uses large value sizes. In contrast, the YCSB-A workload involves only 8 threads, does not induce SSD GC, and uses smaller values, resulting in generally lower latency. For YCSB-A, SlimIO reduced the SET tail latency from 0.711ms to 0.635ms, and the improved write performance and reduced snapshot time also led to a decrease in the GET tail latency from 0.673ms to 0.577ms.

Under the *Always-Log* policy, the reduction in tail latency with SlimIO is even more pronounced. As observed, SlimIO reduces the tail latency by more than 50% compared to the baseline in both workloads.

Memory Usage Results. We use the memory footprint as the metric for memory usage, which reflects the amount of physical memory occupied in the user space. As shown in Table 3 and Table 4, in all workloads, SlimIO exhibits memory usage comparable to baseline. Although SlimIO introduces additional threads, the memory overhead is negligible and does not significantly impact the overall memory footprint.

SSD WAF Results. As shown in Table 3, the baseline shows WAF values of 1.14 and 1.24 under the *Periodical-Log* and *Always-Log* policies, respectively. In contrast, SlimIO achieves an ideal WAF of 1.00, indicating that its write operations do not incur additional internal writes within the SSD due to GC. This result suggests that SlimIO not only improves performance, but also contributes to extending SSD lifespan.

5.3 Recovery Test

We evaluate the performance of data recovery from a snapshot of approximately 20GB. We measure two key metrics: Recovery time and recovery throughput.

Table 5 shows that the baseline required a recovery time of 55.38 seconds with a corresponding recovery throughput of 374.77 MB/s.

Table 5: Recovery Evaluation on Snapshot

	Recovery Time (sec)	Recovery I/O Throughput (MB/s)
Baseline	55.38	374.77
SlimIO	44.12	471.13

In contrast, SlimIO demonstrated a significantly improved recovery time of 44.12 seconds and a recovery throughput of 471.13 MB/s. The baseline utilizes the page cache to enable fast read caching, but still suffers from high system call overhead. Since the Redis recovery process involves only sequential reads, SlimIO can easily implement an efficient read caching mechanism. The read-ahead buffer mechanism optimized for sequential I/O will eliminate system call overhead, resulting in faster recovery.

5.4 Microscopic Analysis

We evaluate whether SlimIO without FDP can provide stable runtime performance. Figure 4 and 5 show the runtime RPS of Redis benchmark workloads under the *Periodical-Log* policy. Figure 4 compares the runtime RPS of the baseline and SlimIO without FDP. We observe several extended periods of RPS instability in both setups, such as around 1000s, 1500s, and beyond, which correspond to the timing of SSD garbage collection (GC) events. During these periods, the baseline maintains relatively stable RPS, whereas SlimIO, which writes directly to the SSD, suffers from sharp drops in RPS—occasionally nosedive to zero—due to the direct impact of GC. In contrast, Figure 5 shows that when SlimIO is backed by an FDP SSD, runtime RPS remains stable between 70000 and 80,000, except during snapshot duration. This shows that while direct SSD writes in SlimIO make it vulnerable to GC-induced performance degradation, adopting FDP SSD as the underlying storage effectively mitigates this issue and enables stable performance.

6 Related Work

Several studies have been conducted on snapshots in IMDBs. Li et al. [24] evaluated and compared existing snapshot algorithms. They showed that for write-intensive workloads, even a simple fork-based method can outperform more advanced snapshot algorithms. Park et al. [30] proposed a memory dump-based snapshot scheme that reduces the overhead caused by memory copying, thus lowering peak memory usage during snapshot generation. Pang et al. [29] optimized copying of page tables from the parent process to the child process, which helped reduce tail latency during snapshot generation. However, these snapshot optimization studies operate on the traditional kernel I/O stack and thus do not address the issues of the kernel I/O path or storage device limitations.

Recently, Samsung and Meta integrated I/O passthru and FDP SSD into the hybrid caching library, Cachelib, demonstrating that such integration can effectively reduce WAF and extend SSD lifetime [8]. While this study shares a similar approach by leveraging I/O passthru and FDP SSD, our focus and target application differ. It targets hybrid caching systems that use SSDs as cache, where the complexity of the LBA structure leads them to prioritize reducing WAF and extending SSD lifetime over query throughput. On the contrary, SlimIO focuses mainly on persistence in IMDB, centered not only on reducing WAF but also on resolving interference between snapshot and WAL operations to improve query throughput.

7 Conclusion

This paper investigates the causes of prolonged snapshot times in Redis, a representative IMDB, leading to performance degradation: high syscall overhead, I/O interference between snapshot and WAL processes, and SSD garbage collection. The traditional kernel I/O path cannot address these issues. To overcome, we propose SlimIO, which employs an io_uring-based I/O passthru to mitigate syscall overhead and inter-process I/O interference, and leverages an FDP SSD to eliminate GC-induced performance loss. Experiments show that SlimIO substantially shortens snapshot duration and improves overall performance during non-snapshot periods.

Acknowledgments

This work was partially supported by Samsung Electronics Co., Ltd. (IO221014-02908-01) and the National Research Foundation of Korea (NRF), funded by the Korean government (MSIT), under Grant No. RS-2025-00564249. This research also used resources of the Oak Ridge Leadership Computing Facility, located at the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

References

[1] 2015. Budget Fair Queueing. https://docs.kernel.org/block/bfq-iosched.html.
[2] 2022. Redis-Multipart. https://github.com/redis/redis/pull/9788.
[3] 2025. io_uring Linux manual page. https://man7.org/linux/man-pages/man7/io_uring.7.html.
[4] 2025. Linux blk-mq.c. https://github.com/torvalds/linux/blob/master/block/blk-mq.c#L2610.
[5] 2025. Redis. https://github.com/redis/redis.
[6] 2025. Redis Benchmark. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/.
[7] 2025. Redis-Persistence. https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/.
[8] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan Nair, and Vivek Shah. 2025. Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. *arXiv preprint arXiv:2503.11665* (2025).
[9] Wes Brewer, Ana Gainaru, Frédéric Suter, Feiyi Wang, Murali Emani, and Shantenu Jha. 2024. AI-coupled HPC workflow applications, middleware and performance. *arXiv preprint arXiv:2406.14315* (2024).
[10] Michael J Brim, Anjus George, Amir Shehata, Corwin Lester, David Rogers, Patrick Widener, Ross Miller, Gustav Jansen, Rafael Ferreira Da Silva, and Sarp Oral. 2024. *A high-level design for bidirectional data streaming to high-performance computing systems from external science facilities.* Technical Report. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States).
[11] Sungjin Byeon, Joseph Ro, Jun Young Han, Jeong-Uk Kang, and Youngjae Kim. 2024. Ensuring Compaction and Zone Cleaning Efficiency through Same-Zone Compaction in ZNS Key-Value Store. In *Proceedings of the 38th International Conference on Massive Storage Systems and Technology (MSST).*
[12] Ping-Xiang Chen, Dongjoo Seo, and Nikil Dutt. 2024. FDPFS: Leveraging File System Abstraction for FDP SSD Data Placement. *IEEE Embedded Systems Letters* 16, 4 (2024), 349–352.
[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing.*
[14] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22).* Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945
[15] Mathieu Doucet, Tanner C. Hobson, and Ricardo Miguel Ferraz Leal. 2017. Django Remote Submission. *Journal of Open Source Software* 2, 16 (08 2017). https://doi.org/10.21105/joss.00366
[16] Silvia Gioiosa, Beatrice Chiavarini, Mattia D'Antonio, Giuseppe Trotta, Balasubramanian Chandramouli, Juan Mata Naranjo, Giuseppa Muscianisi, Mirko Cestari, and Elisa Rossi. 2025. A GDPR-compliant solution for analysis of large-scale genomics datasets on HPC cloud infrastructure. *Journal of Big Data* 12, 1 (2025), 31.

[17] Muhammed Tawfiqul Islam, Renata Borovica-Gajic, and Shanika Karunasekera. 2022. A multi-level caching architecture for stateful stream computation. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. 67–78.

[18] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. 2015. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 191–202. https://www.usenix.org/conference/fast15/technical-sessions/presentation/jeong

[19] Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. 2024. {I/O} Passthru: Upstreaming a flexible and efficient {I/O} Path in Linux. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 107–121.

[20] Yunji Kang and Dongkun Shin. 2021. mStream: stream management for mobile file system using android file contexts. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1203–1208.

[21] Kyoungho Koo, Yongjun Park, and Youjip Won. 2020. LOCKED-Free Journaling: Improving the Coalescing Degree in EXT4 Journaling. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA51238.2020.9188082

[22] Feng Li, Dali Wang, Feng Yan, and Fengguang Song. 2020. ElasticBroker: Combining HPC with Cloud to Provide Realtime Insights into Simulations. *arXiv preprint arXiv:2010.04828* (2020).

[23] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. 2018. The Case of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*.

[24] Liang Li, Guoren Wang, Gang Wu, and Ye Yuan. 2018. Consistent Snapshot Algorithms for In-Memory Database Systems: Experiments and Analysis. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1284–1287. https://doi.org/10.1109/ICDE.2018.00131

[25] Liang Li, Guoren Wang, Gang Wu, and Ye Yuan. 2018. Consistent Snapshot Algorithms for In-Memory Database Systems: Experiments and Analysis. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1284–1287. https://doi.org/10.1109/ICDE.2018.00131

[26] Liang Liang, Heting Zhang, Guang Yang, Thomas Heinis, and Rosa Filgueira. 2023. Optimization towards Efficiency and Stateful of dispel4py. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 2021–2032.

[27] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (ATC)*. 877–891.

[28] Gaoji Liu, Chongzhuo Yang, Qiaolin Yu, Chang Guo, Wen Xia, and Zhichao Cao. 2024. Prophet: Optimizing LSM-Based Key-Value Store on ZNS SSDs with File Lifetime Prediction and Compaction Compensation. In *Proceedings of the 38th International Conference on Massive Storage Systems and Technology (MSST)*.

[29] Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, Zheng Liu, Zhuo Song, Yong Yang, Tao Ma, and Minyi Guo. 2023. Async-Fork: Mitigating Query Latency Spikes Incurred by the Fork-based Snapshot Mechanism from the OS Level. *Proc. VLDB Endow.* 16, 5 (Jan. 2023), 1033–1045. https://doi.org/10.14778/3579075.3579079

[30] Jiwoong Park, Yunjae Lee, Heon Young Yeom, and Yongseok Son. 2020. Memory efficient fork-based checkpointing mechanism for in-memory database systems *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 420–427. https://doi.org/10.1145/3341105.3375782

[31] J Luc Peterson, K Athey, PT Bremer, V Castillo, F Di Natale, JE Field, D Fox, J Gaffney, D Hysom, SA Jacobs, et al. 2019. *Merlin: enabling machine learning-ready HPC ensembles*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[32] Zebin Ren and Animesh Trivedi. 2023. Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. 35–45.

[33] Samsung. 2023. Host Workloads Achieving WAF==1 in an FDP SSD. https://www.sniadeveloper.org/sites/default/files/SDC/2023/presentations/SNIA-SDC23-Helmick-Host-Workloads-Achieving-WAF_0.pdf.

[34] Samsung. 2024. Getting Started with Flexible Data Placement (FDP). https://download.semiconductor.samsung.com/resources/white-paper/getting-started-with-fdp-v4.pdf.

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

[36] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

Appendix: Artifact Description

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 We conducted an in-depth analysis of the snapshot and non-snapshot periods of Redis, a representative in-memory database (IMDB). Our findings show that using the traditional kernel I/O path causes significant performance degradation due to issues such as syscall overhead, I/O contention between processes, and delays from SSD garbage collection (GC).

C_2 We designed SlimIO to (i) eliminate syscall overhead and inter-process I/O interference through I/O passthru, and (ii) utilize Flexible Data Placement (FDP) SSDs as backup storage to avoid performance degradation caused by SSD GC.

C_3 We implemented SlimIO on Redis v7.4.2 and demonstrated its effectiveness through various experiments. Experimental results show that SlimIO reduces snapshot time by up to 25%, increases query throughput by up to 30% during non-snapshot periods, and lowers 99.9%-ile latency by up to 50%.

A.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.16789521

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1,C_2,C_3	Table 3-5, Figure 3-5

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

The SlimIO artifact provides the source code necessary to reproduce the experiments for our proposed system, supporting contributions C2 and C3. These experiments aim to evaluate the impact of integrating I/O passthru and FDP SSDs on the persistence performance of Redis, a representative IMDB. By analyzing the results, we can gain valuable insights into SlimIO's effectiveness in reducing snapshot time, increasing query throughput, lowering tail latency, and improving the write amplification factor (WAF), thereby demonstrating its ability to minimize I/O overhead and extend SSD lifespan.

Expected Results

This artifact aims to reproduce the following key experimental results, demonstrating improvements over the baseline that relies on the traditional kernel I/O path.

- **Shortened Snapshot Duration:** SlimIO reduces snapshot duration—which constitutes the primary period of Redis performance degradation—by up to 25%.
- **Improved Query Throughput:** SlimIO achieves up to 30% higher Requests per Second (RPS) than the baseline in the non-snapshot periods.
- **Stable Reduction in Tail Latency:** SlimIO reduces the 99.9%-ile latency by up to 50%, providing much more stable and predictable performance.
- **SSD WAF Optimization:** By leveraging FDP SSDs, SlimIO achieves an ideal WAF of 1.00, indicating no redundant internal writes within the SSD.
- **Fast Data Recovery:** SlimIO's recovery mechanism delivers approximately 20% faster recovery time and improved I/O throughput compared to the baseline.

Expected Reproduction Time (in Minutes)

The expected time to reproduce the artifact is as follows:

- Artifact Setup: Approximately 30 minutes, including installation of Redis, the FEMU-based FDP emulator, required dependencies, and configuration of the experimental environment.
- Artifact Execution: 50–150 minutes, depending on the Redis logging policy and workload configuration.
- Artifact Analysis: Approximately 30 minutes, focusing on extracting performance metrics (RPS, latency, snapshot time, and WAF) from generated results and comparing them with the tables and figures in the paper.

Artifact Setup (incl. Inputs)

Hardware. The FDP SSD is emulated using FEMU v9.0, configured with 12 cores, 55 GB DRAM, and a total capacity of 180 GB. The emulated FDP SSD features 8 channels with 8 dies per channel, a NAND page size of 4KB, and an RU size of 1GB; the NAND page read latency, write latency, and block erase latency are set to 40µs, 200µs, and 2ms respectively, all based on FEMU's default settings. Our host environment is equipped with dual Intel Xeon Gold 5218R CPUs and 377GB of DRAM.

Software. The implementation of SlimIO is done in C on top of Redis and includes the following dependencies:

- liburing (version 2.5)
- GCC (version 13.3.0)
- Linux Kernel (version 6.7.9)
- FEMU (version 9.0.0)
- Redis (version 7.4.2)

Datasets / Inputs. Two standard benchmark workloads are used. No separate download is required; the benchmark tools generate the workloads.

- **Redis-benchmark:** We configure 50 concurrent clients, a key range of 5.3 million, 8-byte keys, and 4096-byte values. Each experiment issues 28 million SET operations and is repeated five times. Final results are aggregated over all repetitions.
- **YCSB benchmark:** We use the YCSB-A workload with 8 threads, 8-byte keys, and 2048-byte values, comprising 9 million records and 115 million operations with a 0.5:0.5 GET-to-SET ratio.

Installation and Deployment. First, set up the emulated environment using the FEMU FDP SSD from the repository at https://github.com/lass-lab/ConfFDP/tree/solesie. Detailed download instructions are provided in the README file. To run the emulator, execute the script `./femu-scripts/run-fdp.sh`.

Second, within the emulated environment, install the liburing dependency and then install SlimIO. The installation procedure for SlimIO follows the standard Redis installation instructions as described in the README.

Third, for baseline experiments, run the script `redis/origin_redis/redis/fs_*.sh`. These scripts use the original Redis setup, and the results are saved in the directory `redis/origin_redis/redis/bench-results`. For SlimIO experiments, run the scripts `redis/slimio_*.sh`. The results for these experiments are stored in `redis/bench-results`.

Artifact Execution

The experimental workflow consists of T_1 (environment setup), T_2 (benchmark execution), and T_3 (result analysis).

- Task T_1: Install the mentioned dependencies, and for the baseline configuration, set the I/O scheduler to 'none' and mount the baseline file system.
- Task T_2: Redis is configured with two logging policies: the Periodical-Log policy, which accumulates write queries in a user-level Write-Ahead-Log buffer for several seconds before flushing them to the storage device and serves as the default logging policy; and the Always-Log policy, which immediately reflects every write query to the storage device. For each workload, results are measured separately for these two policies. The Redis-benchmark workload parameters are configured to emulate a large-data, write-intensive scenario, while the YCSB-A workload parameters are set to emulate a small-data, less write-intensive scenario.
- Task T_3: For both logging policies and each workload, we measure RPS, 99.9%-ile latency, snapshot duration, SSD WAF, and data recovery performance. To evaluate the efficiency of FDP SSDs, runtime RPS is continuously tracked during execution.

Artifact Analysis (incl. Outputs)

For the overall performance comparison, refer to Tables 3–5 in the paper, which present results against the baseline. Figures 4–5 show the runtime RPS measured through a microscopic analysis, which evaluates the efficiency of FDP SSDs.