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

Sangyun Lee[1], Sungjin Byeon[1], Soon Hwang[1], Jaewan Park[1], Joo-Young Hwang[2], Junyoung Han[2], Javier González[2], Awais Khan[3], Youngjae Kim[1]

[1] *Sogang University*

[2] *Samsung Electronics Co.*

[3] *Oak Ridge National Laboratory*

2025 10th International Workshop on Parallel Data Storage, held in conjunction with SC25, Nov 16–21, 2025, St. Louis, MO

Contents

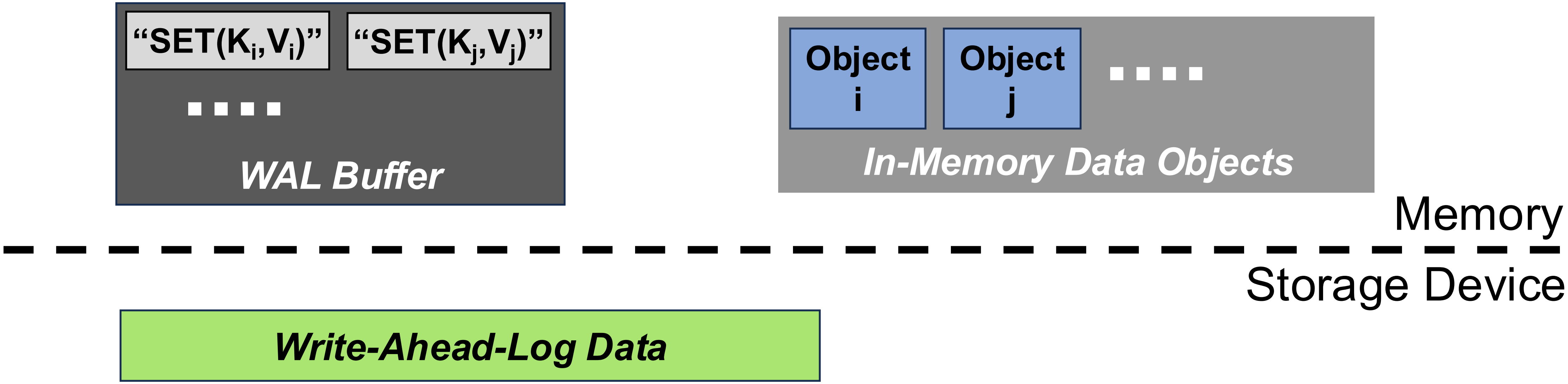
Redis Persistence

- **Redis**, a representative In-Memory Database (IMDB), stores all data in DRAM.
- **Data Loss**
  - IMDBs enables high performance but risks total data loss on unexpected failures.
  - Recovery in real-time systems can take several days.
- To prevent data loss, IMDBs employ persistence mechanisms.

Redis Persistence

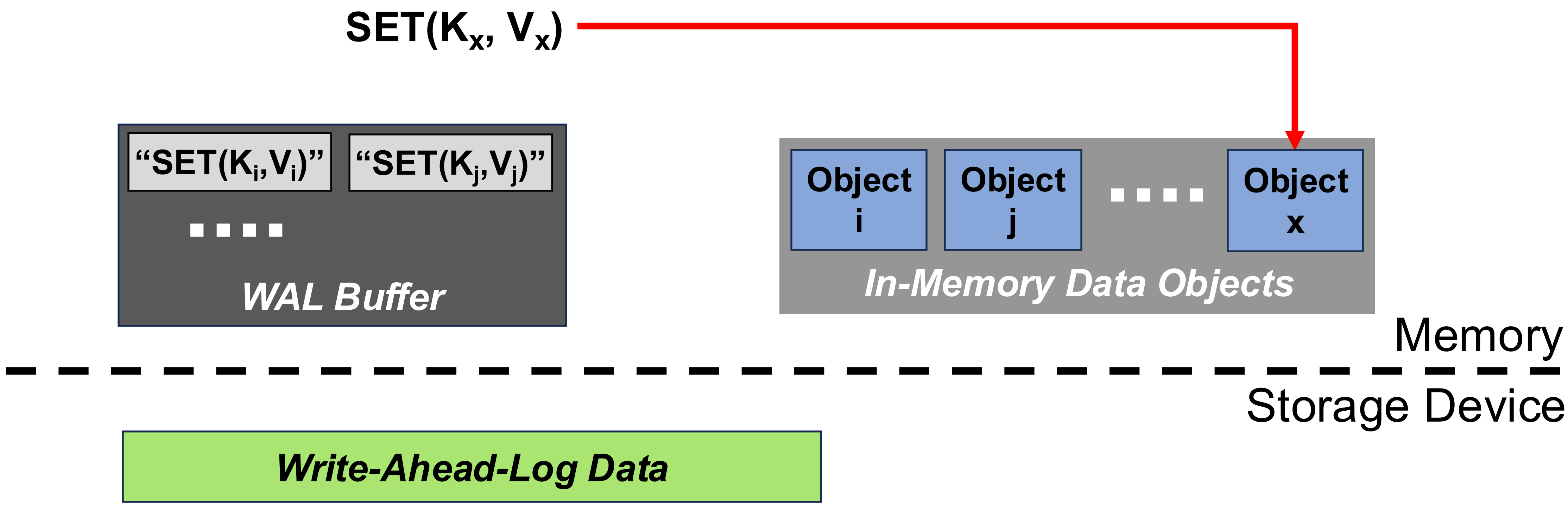
- **Redis**, a representative In-Memory Database (IMDB), stores all data in DRAM.
- **Data Loss**
  - IMDBs enables high performance but risks total data loss on unexpected failures.
  - Recovery in real-time systems can take several days.
- To prevent data loss, IMDBs employ persistence mechanisms.
- Redis uses two mechanisms:
  Write-Ahead Log (WAL) and **Snapshot**

Redis Persistence - WAL



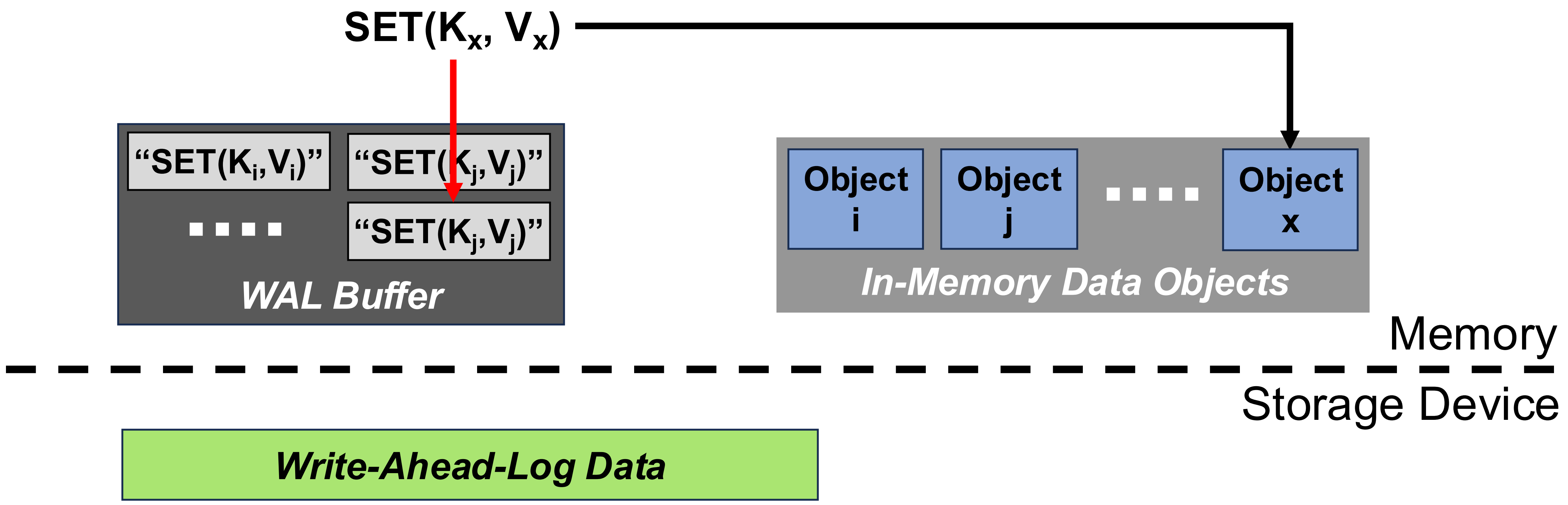
- **Write-Ahead-Log (WAL):**
  Sequentially logs all write queries to storage device.

Redis Persistence - WAL



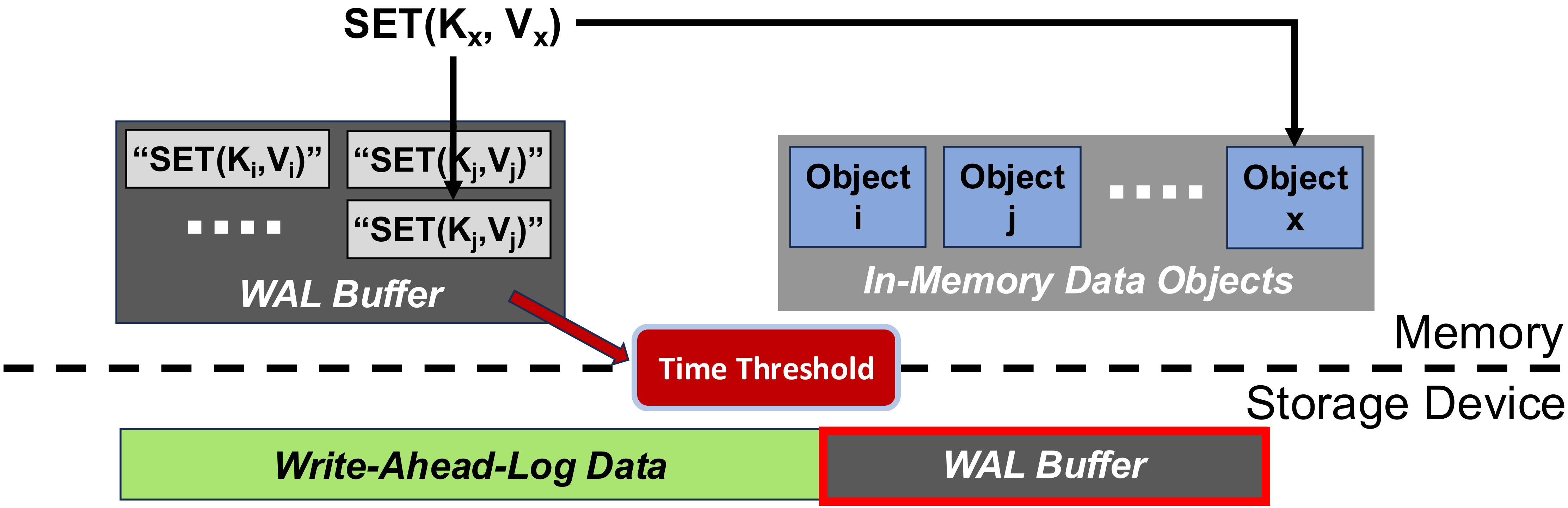
- **Store Data in Memory:**
  (KEY_x ,$Value_x$) data is first stored as in-memory objects.

Redis Persistence - WAL



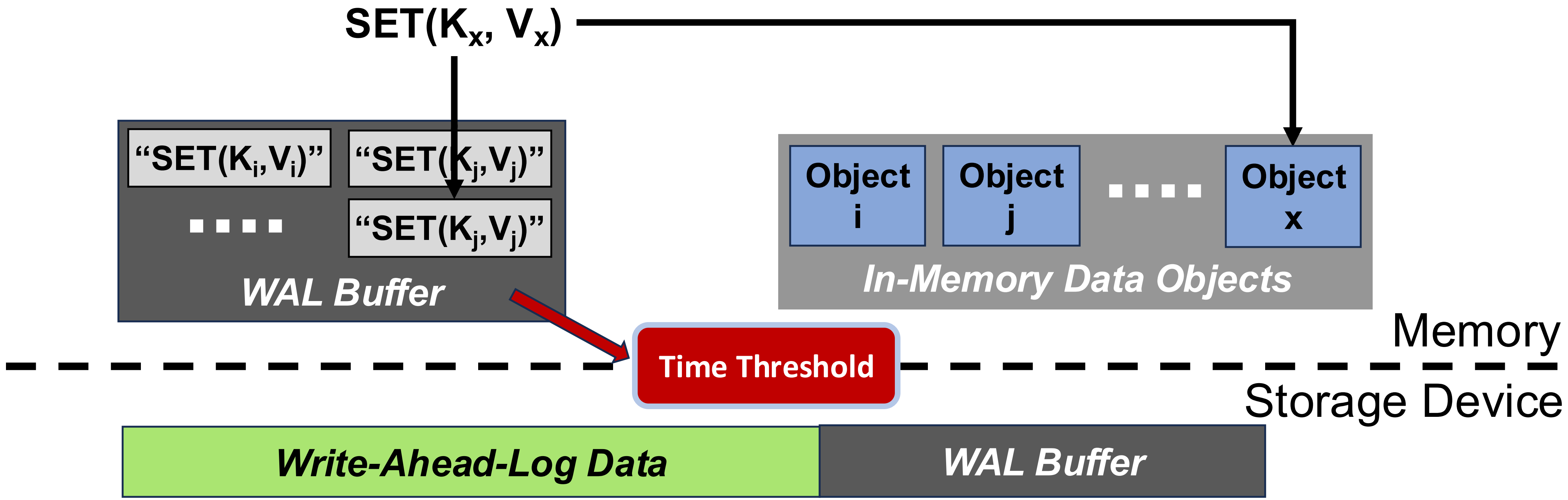
- **Log Query in WAL Buffer:**
  Then, the query statement "SET Key x, Value x" itself is temporarily stored in the WAL buffer.

Redis Persistence - WAL



- **Flush WAL Buffer:**
  When the time threshold is reached, WAL buffer contents are flushed to the storage device.

Redis Persistence - WAL



- **Recovery from Failure:**
  If in-memory data objects are lost due to an unexpected failure, Redis replays the Write-Ahead Log to restore data.

Redis Persistence - Snapshot

- **Snapshot:**
  Compresses and saves full in-memory dataset to storage device.
- Snapshots in Redis are created two ways:

Redis Persistence - Snapshot

- **Snapshot:**
  Compresses and saves full in-memory dataset to storage device.
- Snapshots in Redis are created two ways:
  - **WAL-Snapshot:**
    Automatically triggered when WAL grows too large to limit its size.

Redis Persistence - Snapshot

- **Snapshot:**
  Compresses and saves full in-memory dataset to storage device.
- Snapshots in Redis are created two ways:
  - **WAL-Snapshot:**
    Automatically triggered when WAL grows too large to limit its size.
  - **On-Demand-Snapshot:**
    Manually or periodically created by admins for backups, data transfer, or specific points in time (e.g., before a server release or testing).

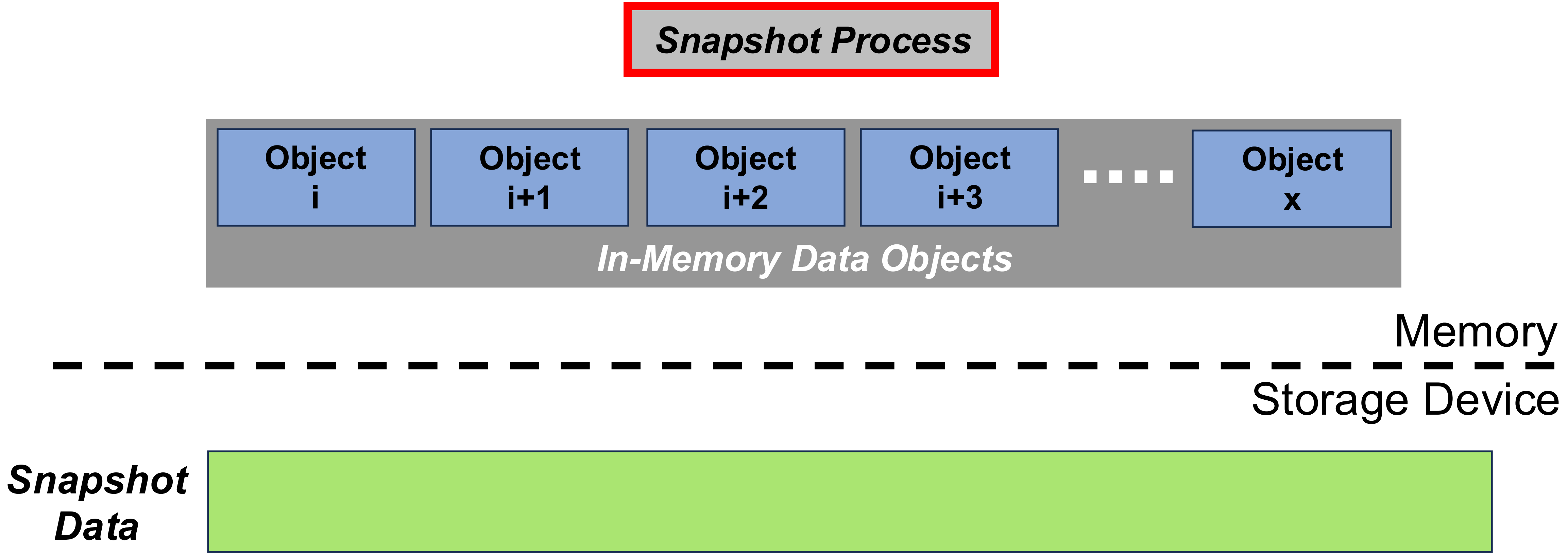
Redis Persistence - Snapshot

- **Snapshot:**
  Compresses and saves full in-memory dataset to storage device.
- Snapshots in Redis are created two ways:
  - **WAL-Snapshot:**
    Automatically triggered when WAL grows too large to limit its size.
  - **On-Demand-Snapshot:**
    Manually or periodically created by admins for backups, data transfer, or specific points in time (e.g., before a server release or testing).
  - Only one WAL-Snapshot and one On-Demand-Snapshot can exist at once; they cannot be created simultaneously.

Redis Persistence - Snapshot

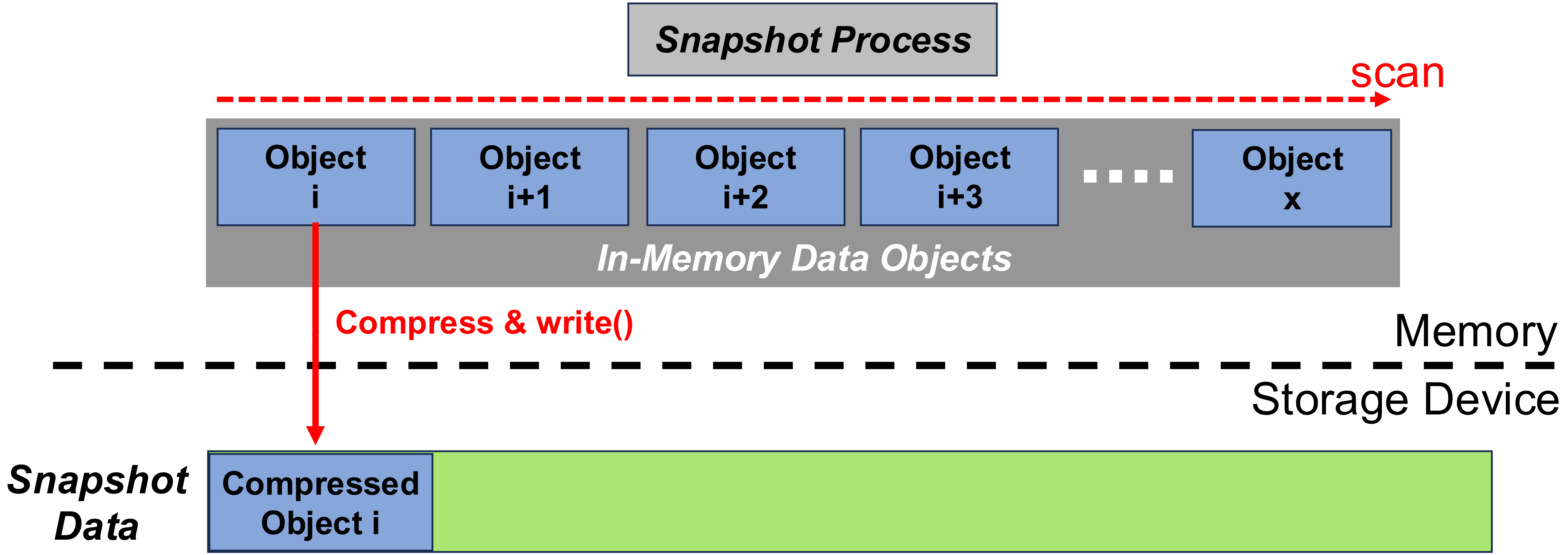
- **Snapshot:**
  Compresses and saves full in-memory dataset to storage device.
- Snapshots in Redis are created two ways:
  - **WAL-Snapshot:**
    Automatically triggered when WAL grows too large to limit its size.
  - **On-Demand-Snapshot:**
    Manually or periodically created by admins for backups, data transfer, or specific points in time (e.g., before a server release or testing).
  - Only one WAL-Snapshot and one On-Demand-Snapshot can exist at once; they cannot be created simultaneously.
- WAL writing and Snapshot creation can occur concurrently.

Redis Persistence - Snapshot



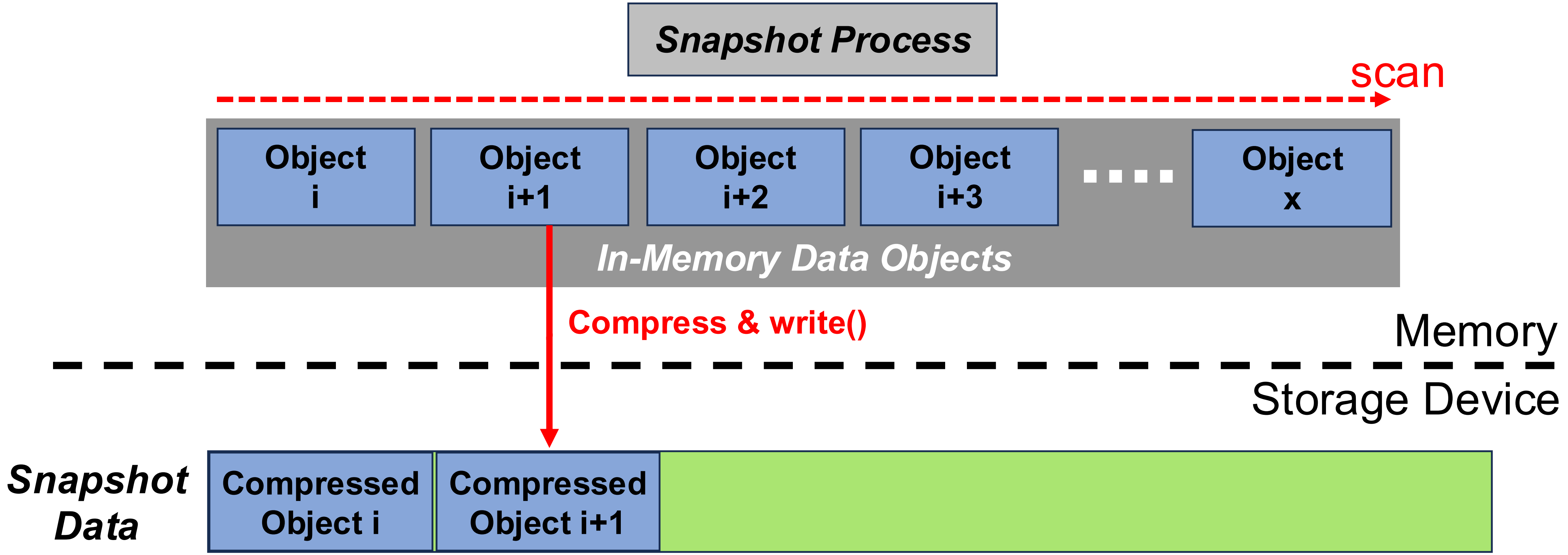
- **Create Snapshot Process:**
  Redis uses fork() to create a child process that handles snapshot I/O, enabling parallel query processing.

Redis Persistence - Snapshot



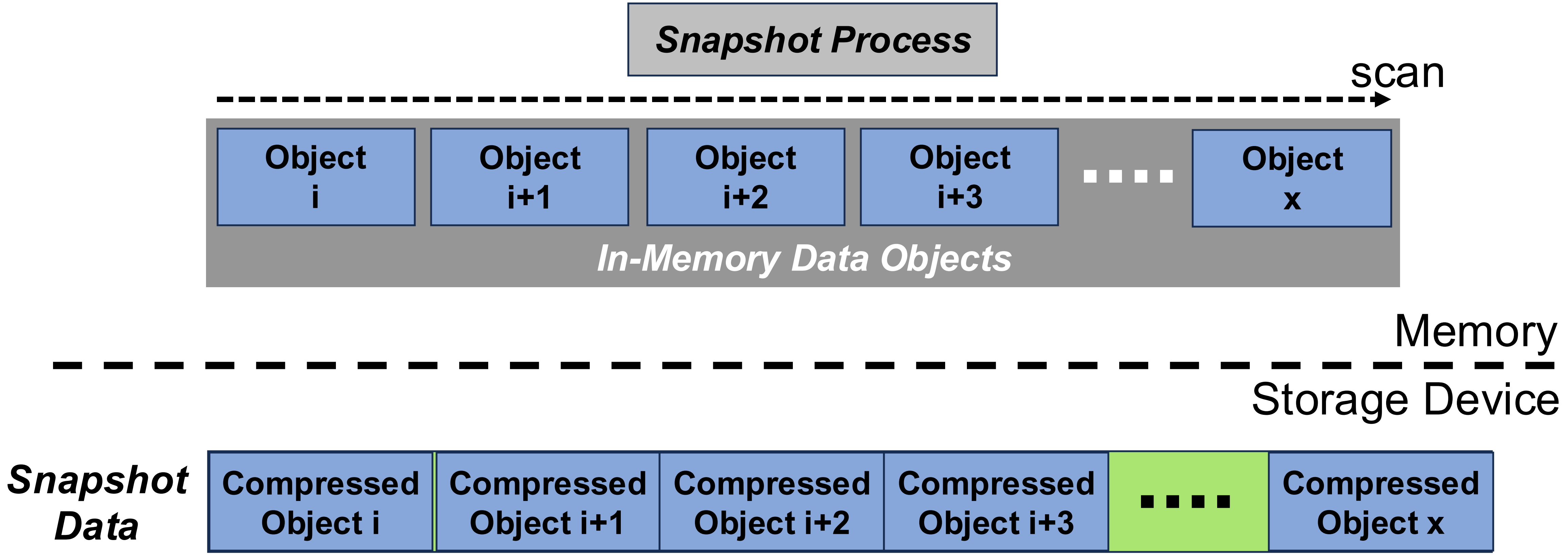
- Sequentially scans in-memory data objects, compresses each, and writes them to storage device using write() system calls.

Redis Persistence - Snapshot



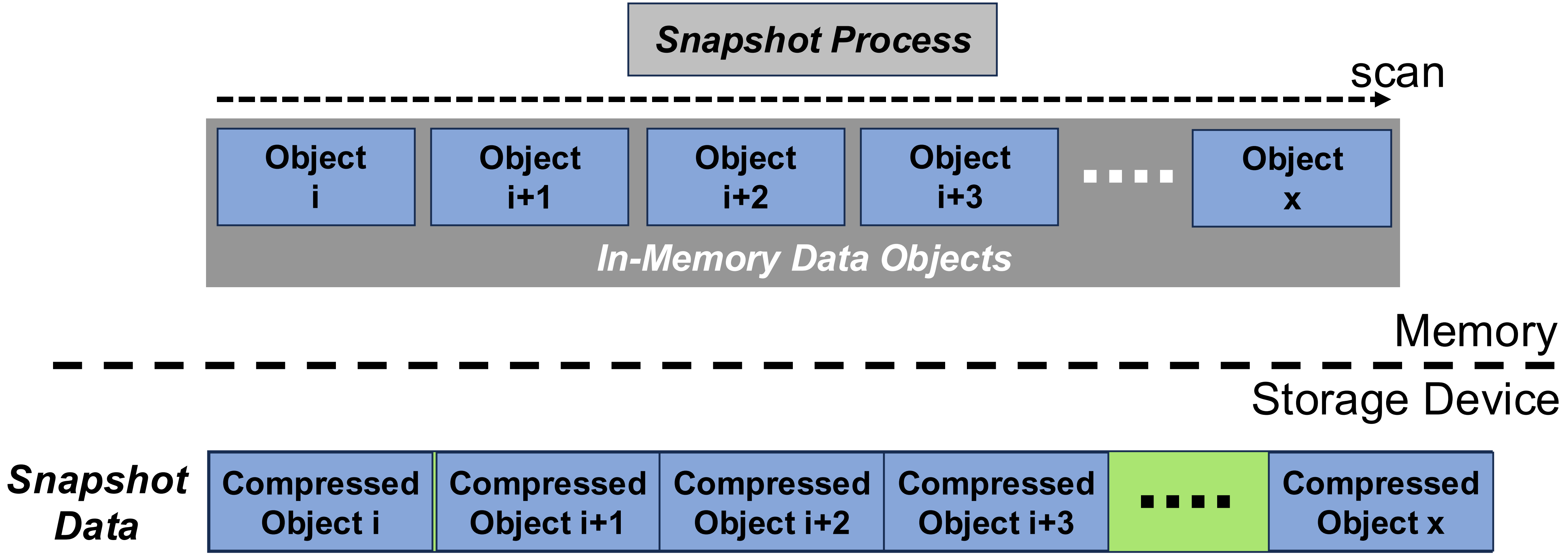
- Sequentially scans in-memory data objects, compresses each, and writes them to storage device using write() system calls.

Redis Persistence - Snapshot



- **WAL-Snapshot case:**
  After completion, existing WAL is deleted after the snapshot is created.

Redis Persistence - Snapshot



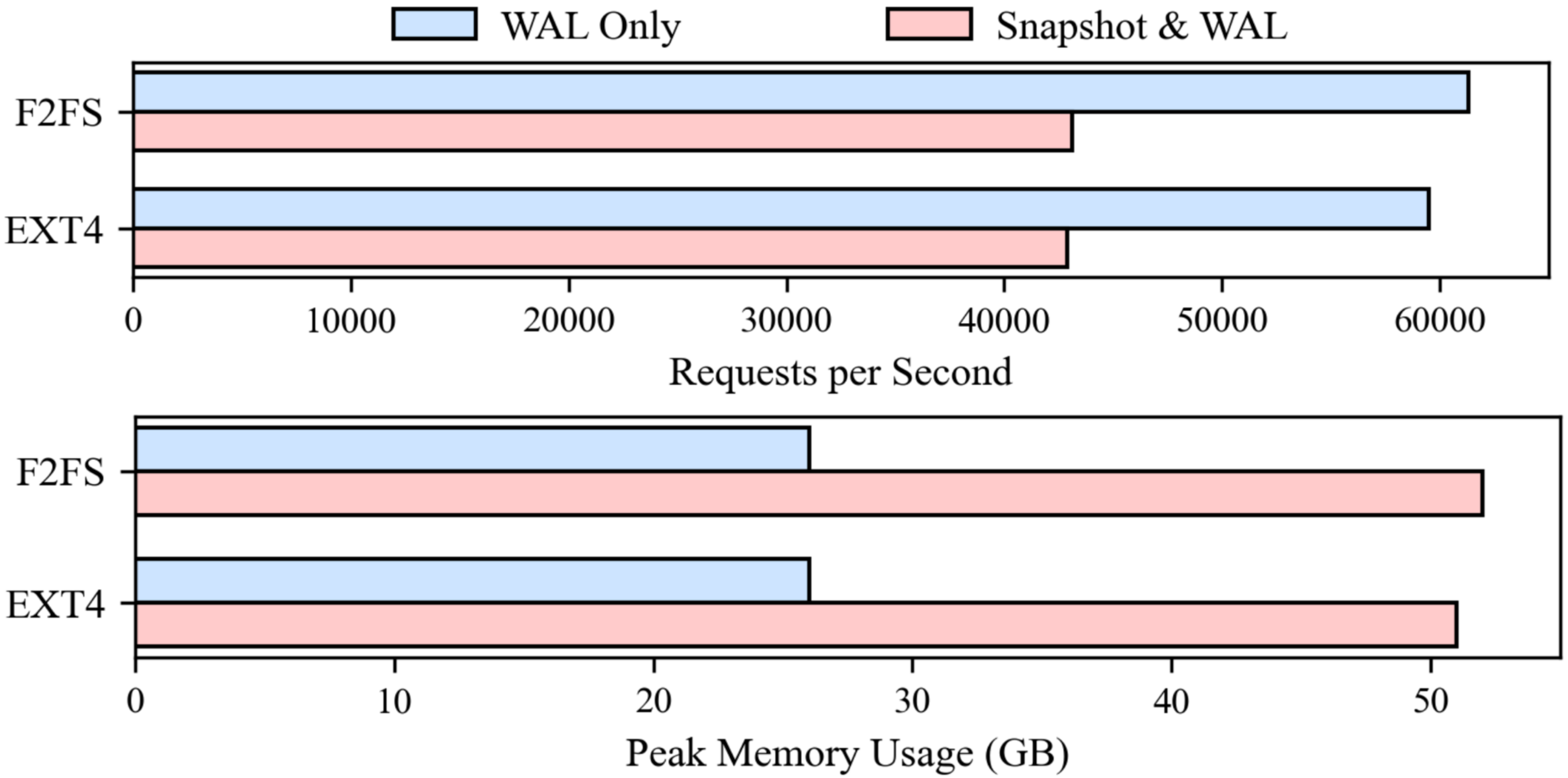
- **Recovery from Failure:**
  If in-memory data objects are lost due to an unexpected failure, Redis loads the snapshot into memory and then replays the WAL.

Redis Persistence - Snapshot

- Snapshot increases memory use and reduces query throughput (fork()'s CoW policy).

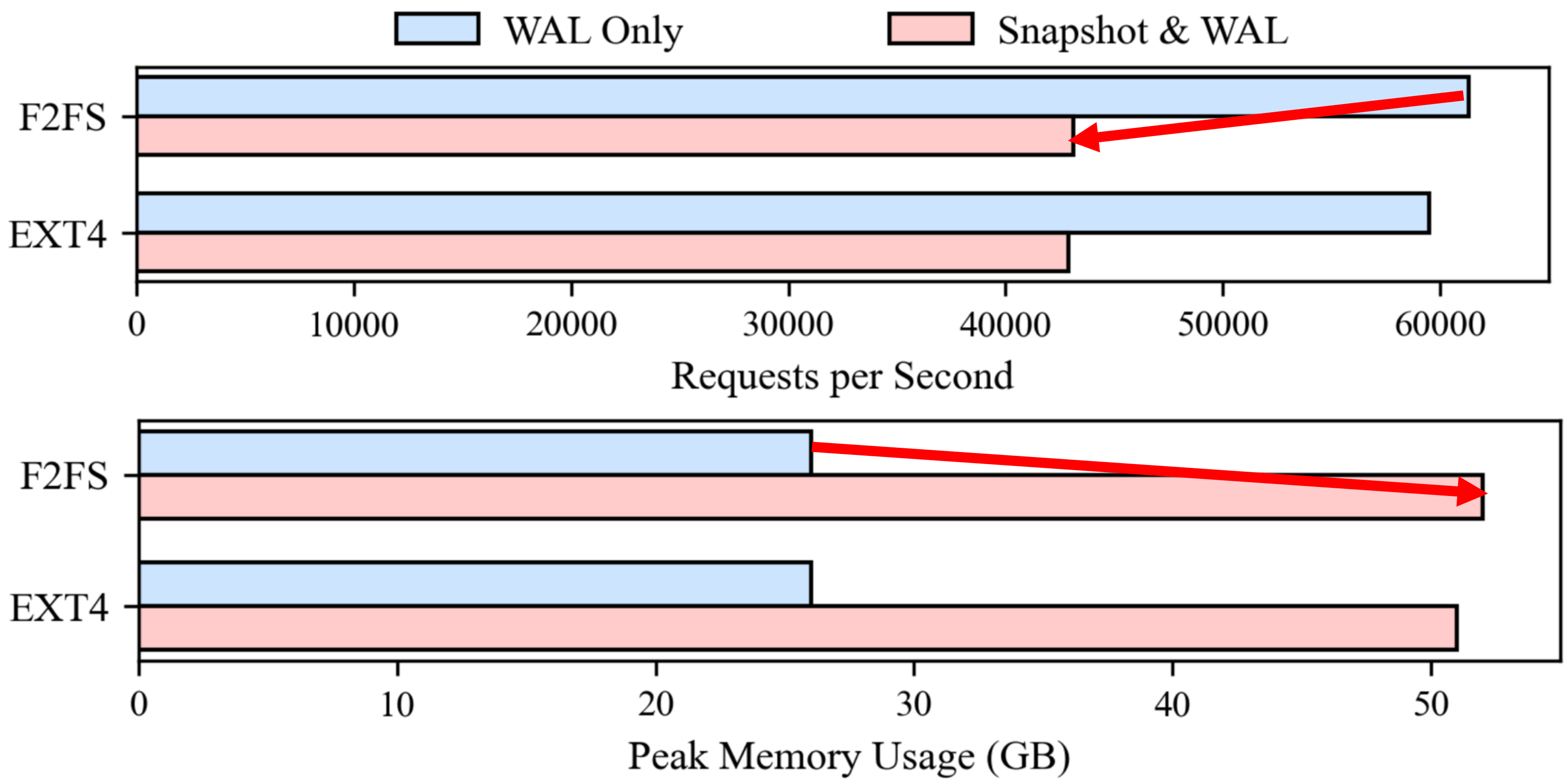
Redis Persistence - Snapshot

- Snapshot increases memory use and reduces query throughput (fork()'s CoW policy).



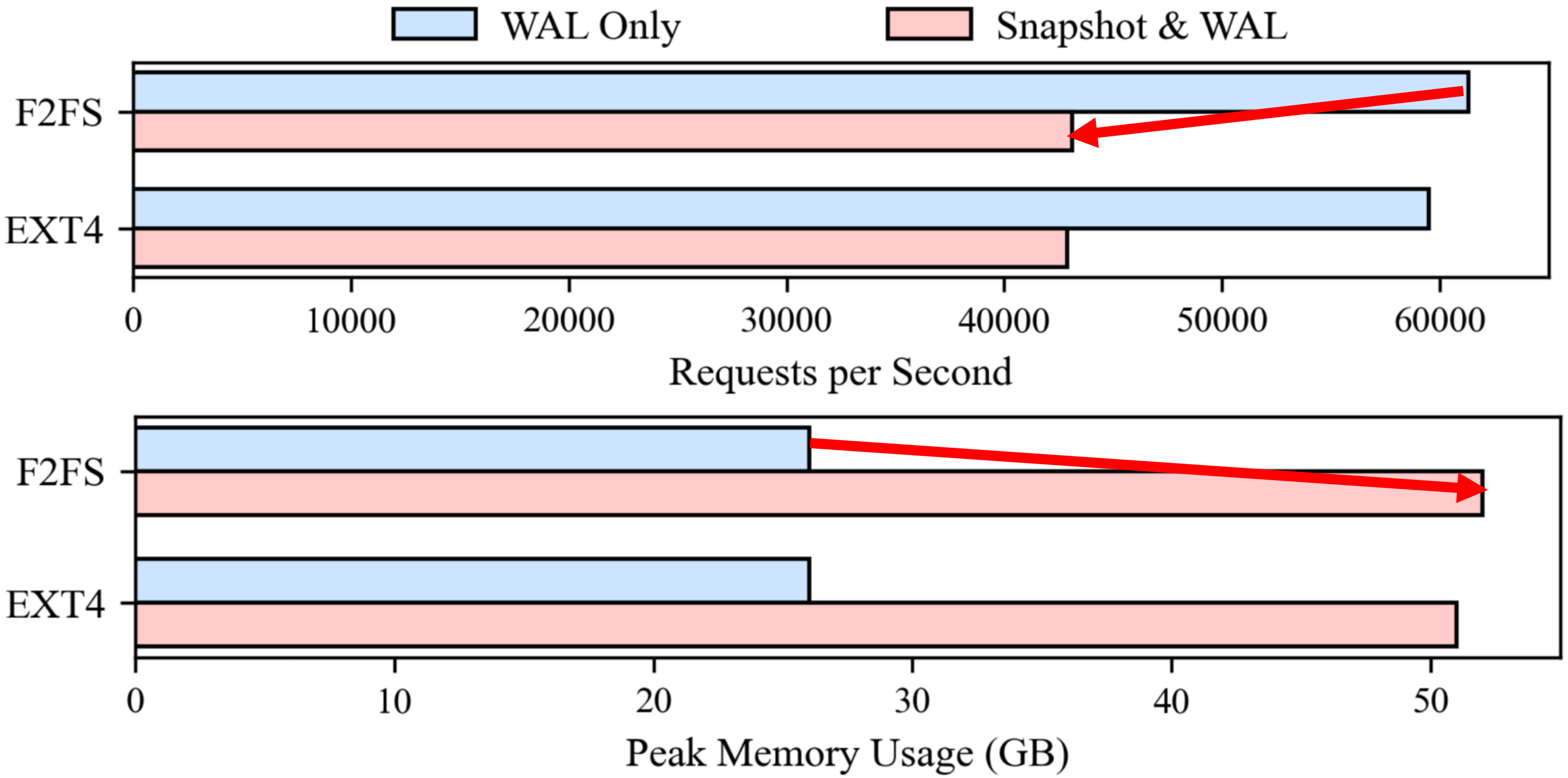
Redis Persistence - Snapshot

- Snapshot increases memory use and reduces query throughput (fork()'s CoW policy).



Redis Persistence - Snapshot

- Snapshot increases memory use and reduces query throughput (fork()'s CoW policy).
- Longer snapshots worsen memory pressure and throughput drops, so minimizing snapshot duration is essential.



Contents

Motiv Experiment: Snapshot Duration Analysis

- **Metrics:** Snapshot Duration, Snapshot Throughput, WAL Throughput
- Three Snapshot Scenarios:

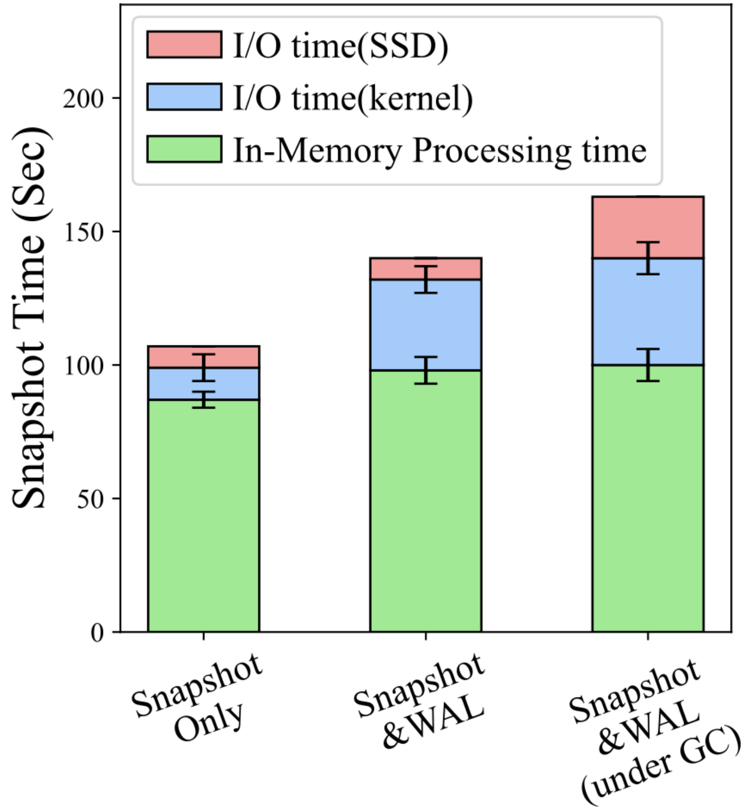
Motiv Experiment: Snapshot Duration Analysis

- **Metrics:** Snapshot Duration, Snapshot Throughput, WAL Throughput
- Three Snapshot Scenarios:
  - **Snapshot Only:**
    On-Demand-Snapshot generation occurs without WAL operations.
  - **Snapshot & WAL:**
    Snapshot and WAL operations occur concurrently.
  - **Snapshot & WAL (under GC):**
    Snapshot and WAL operations occur concurrently while the SSD experiences Garbage Collection (GC) pressure

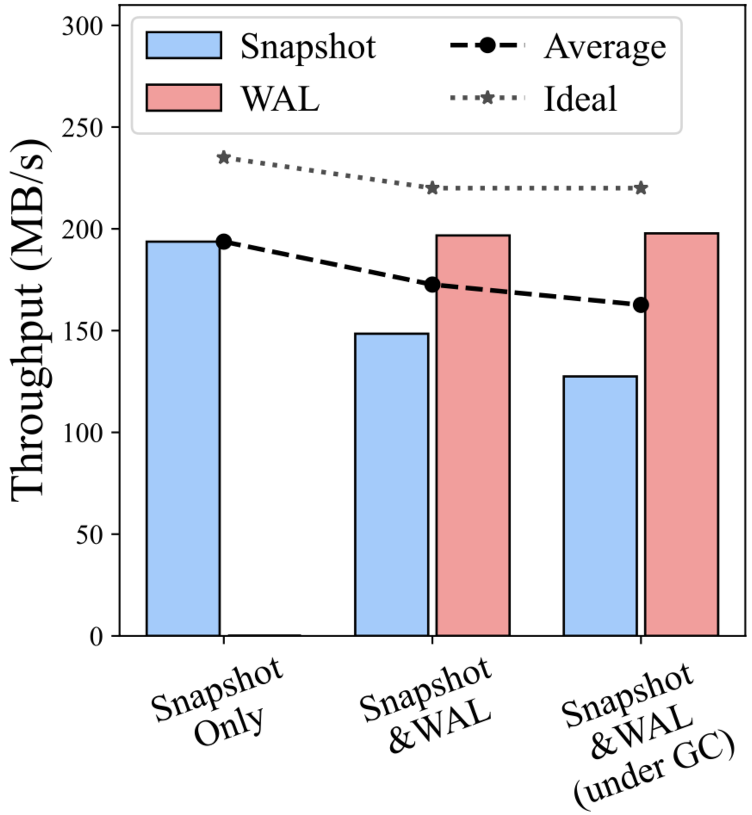
Motiv Experiment: Snapshot Duration Analysis

- **Metrics:** Snapshot Duration, Snapshot Throughput, WAL Throughput
- Three Snapshot Scenarios:
  - **Snapshot Only:**
    On-Demand-Snapshot generation occurs without WAL operations.
  - **Snapshot & WAL:**
    Snapshot and WAL operations occur concurrently.
  - **Snapshot & WAL (under GC):**
    Snapshot and WAL operations occur concurrently while the SSD experiences Garbage Collection (GC) pressure
- ***Ideal Situation:***
  In-memory snapshot tasks such as index search, compression, and memory copying are fully overlapped with kernel and SSD I/O times.

Motiv Experiment: Snapshot Duration Analysis

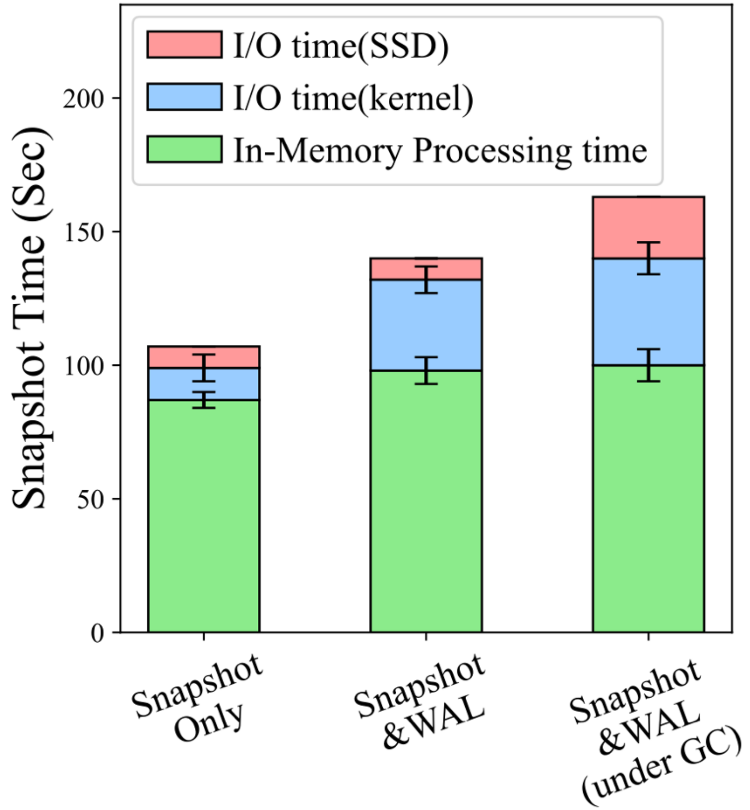


(a) Snapshot Time Distribution



(b) Throughput Analysis

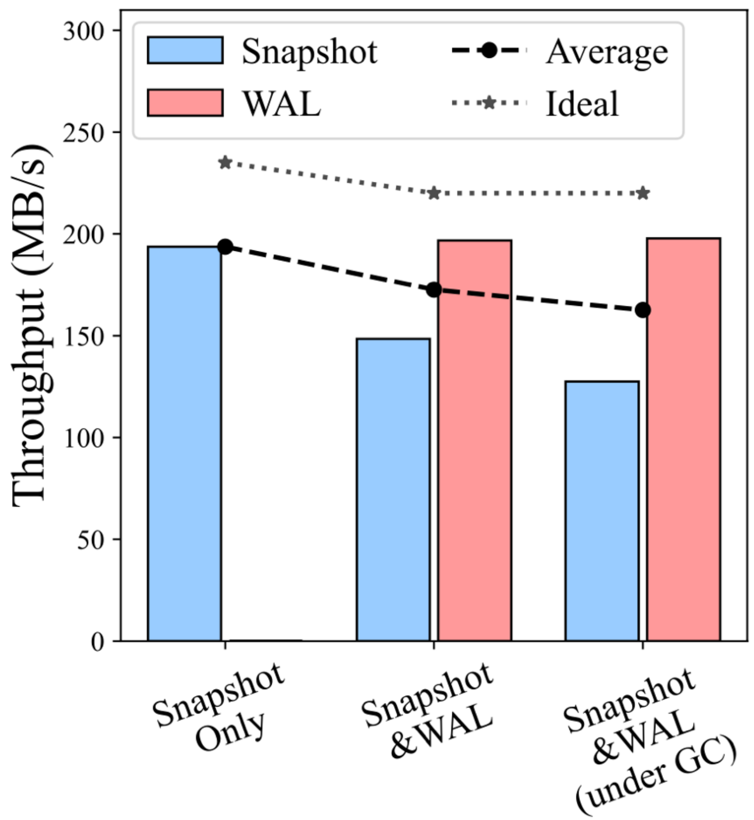
Motiv Experiment: Snapshot Duration Analysis



(a) Snapshot Time Distribution

- Snapshot Time Distribution graph shows the distribution of snapshot durations.
- **Red**:
  time spent inside the SSD
- **Blue**:
  time spent inside the kernel
- **Green**:
  time spent on in-memory operations like compression

Motiv Experiment: Snapshot Duration Analysis



(b) Throughput Analysis

- Throughput Analysis graph shows WAL and Snapshot throughput.
- **Blue:**
  throughput used by Snapshot
- **Red:**
  throughput used by WAL
- **Black dashed line:**
  average WAL and Snapshot throughput
- **Gray dashed line:**
  throughput in the ***ideal situation***

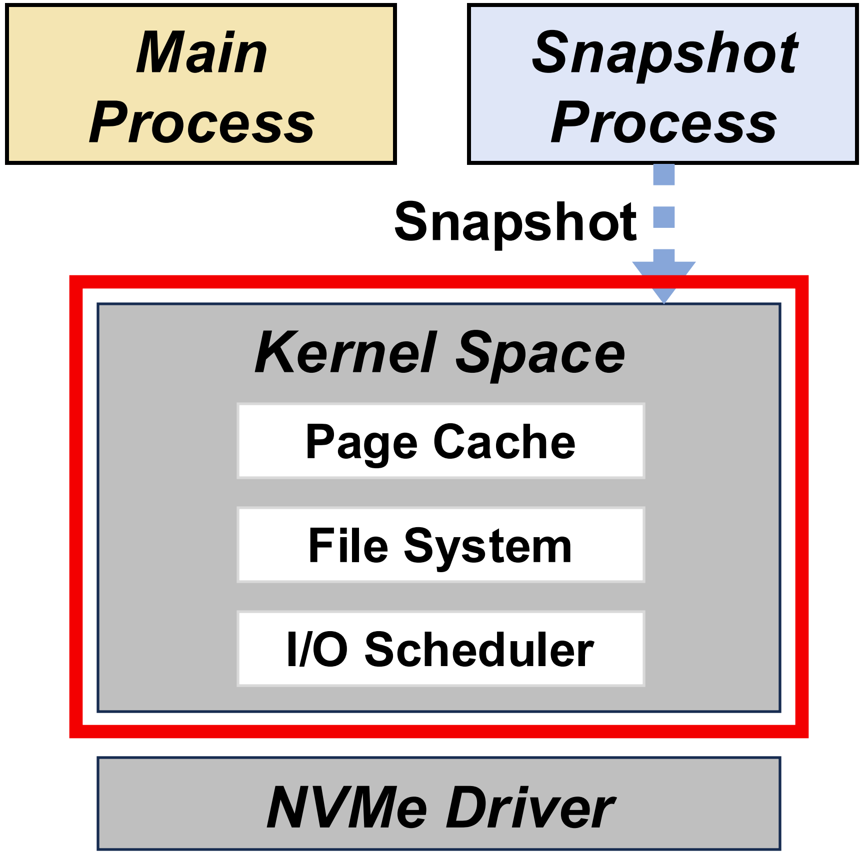
Four Key Observations

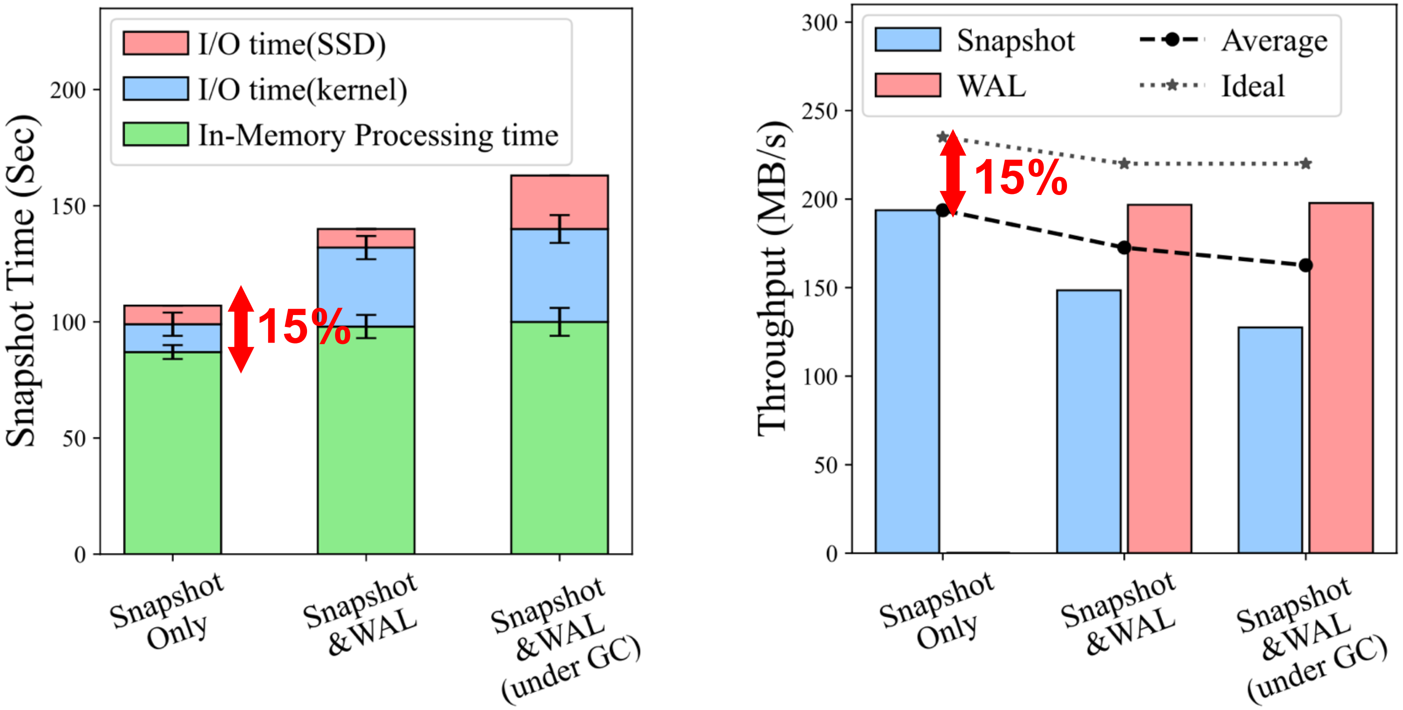
We closely analyzed the results of the motivation experiment.

As a result, we identified **Four Key Observations** in the kernel I/O path that prevent achieving the ideal situation.

- **O1.** Kernel I/O path has high syscall overhead.
- **O2.** Kernel I/O path has a scalability problem.
- **O3.** Kernel I/O path ignores per-process write patterns.
- **O4.** Kernel I/O path lacks sufficient mechanisms to eliminate SSD GC.

O1: Kernel I/O path has high syscall overhead.

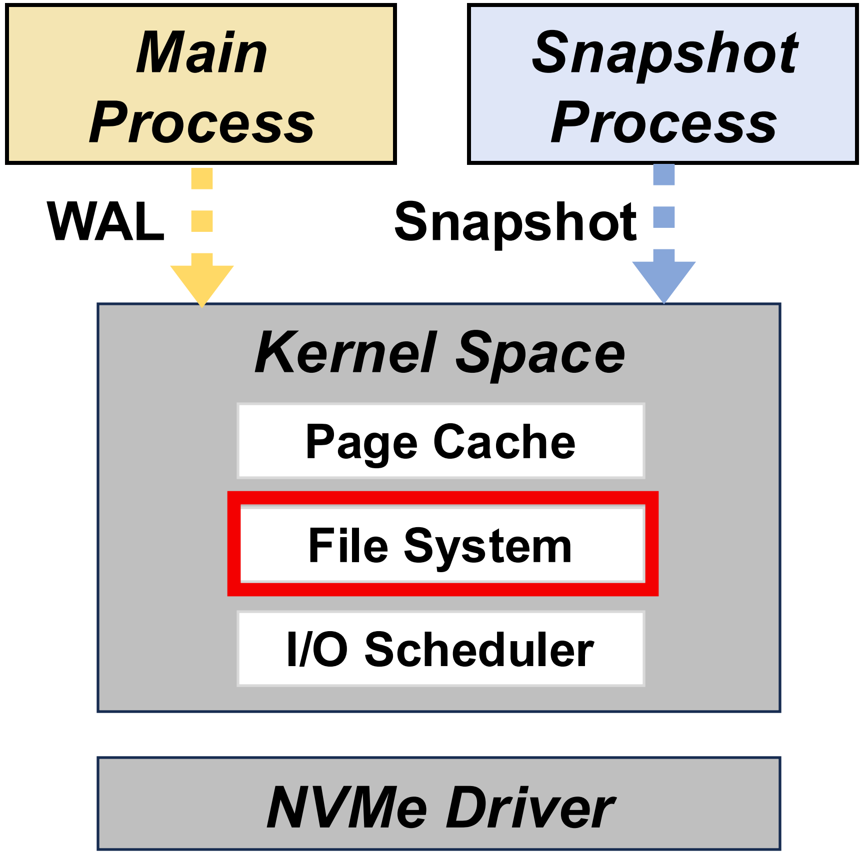


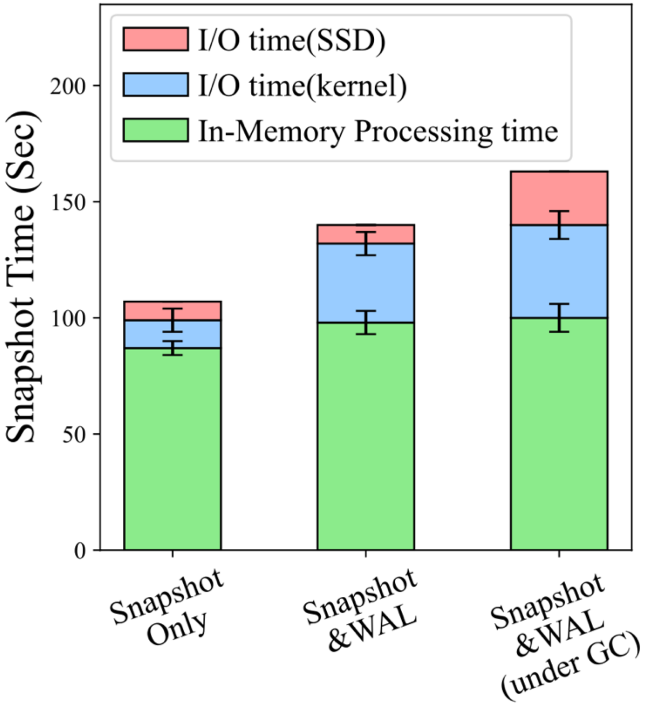


(a) Snapshot Time Distribution **(b) Throughput Analysis**

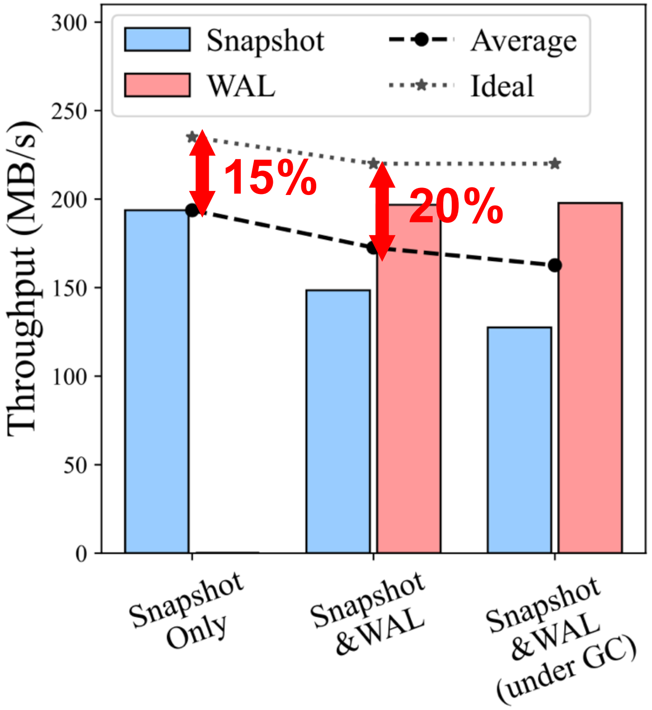
- Left graph shows ~15% of time occurs in the kernel even with only the snapshot process.
- Snapshot processing fails to achieve the ***ideal situation*** where in-memory tasks are fully overlapped with kernel and SSD I/O times.

O2. Kernel I/O path has a scalability problem.





(a) Snapshot Time Distribution

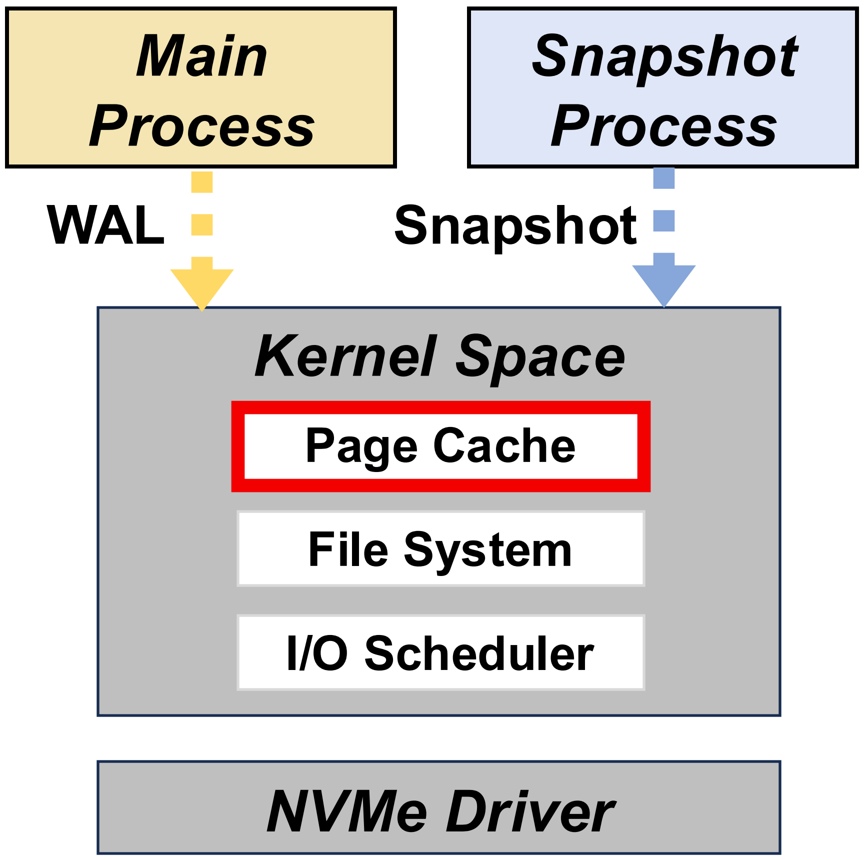


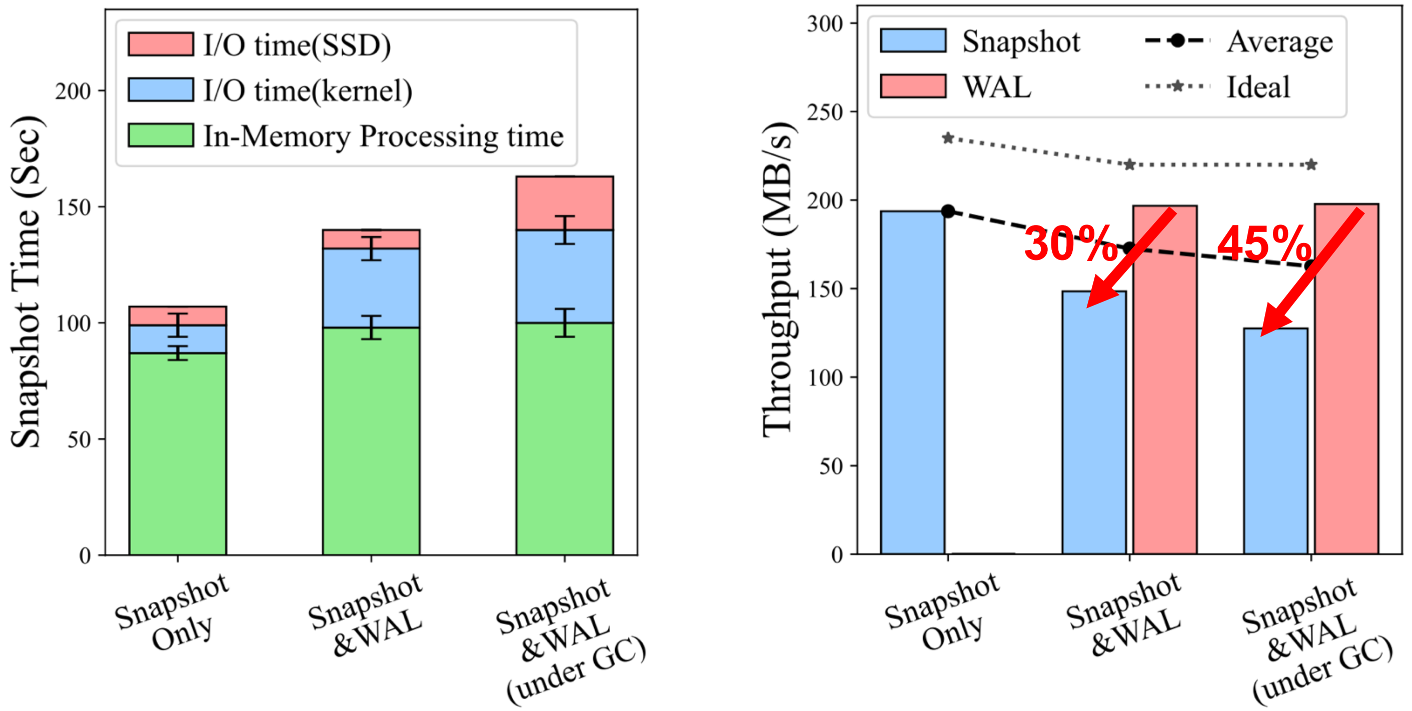
(b) Throughput Analysis

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%

O3: Kernel I/O path ignores per-process write patterns.

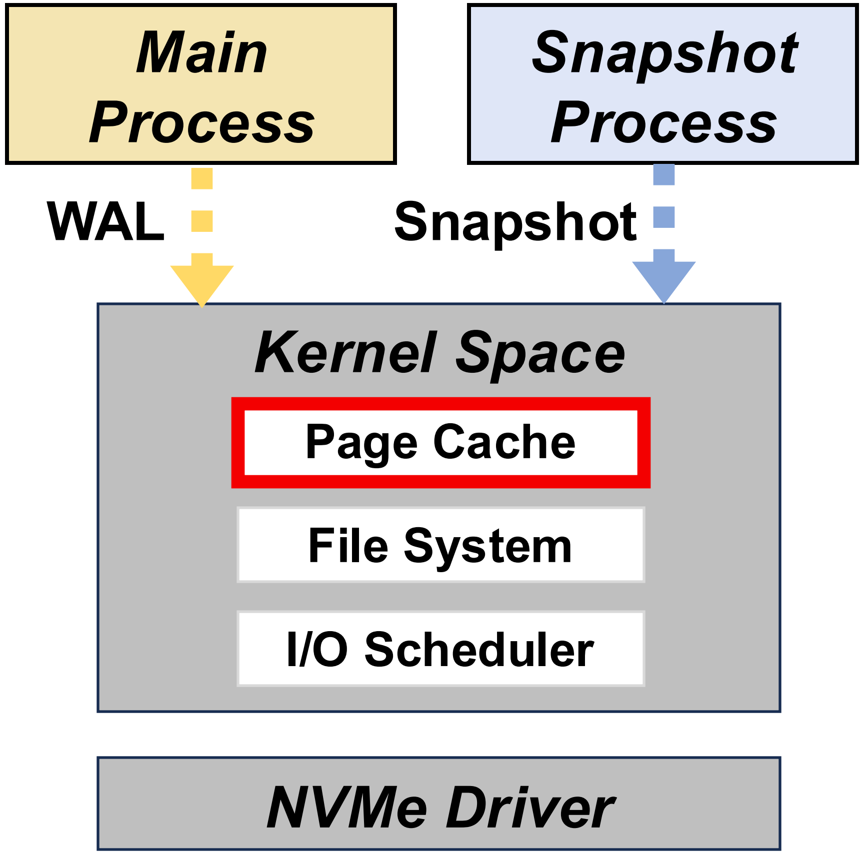


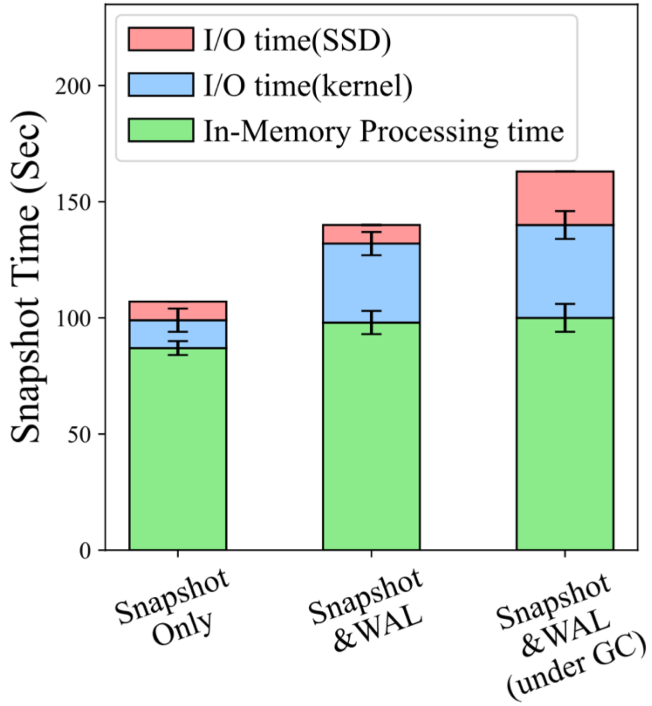


(a) Snapshot Time Distribution (b) Throughput Analysis

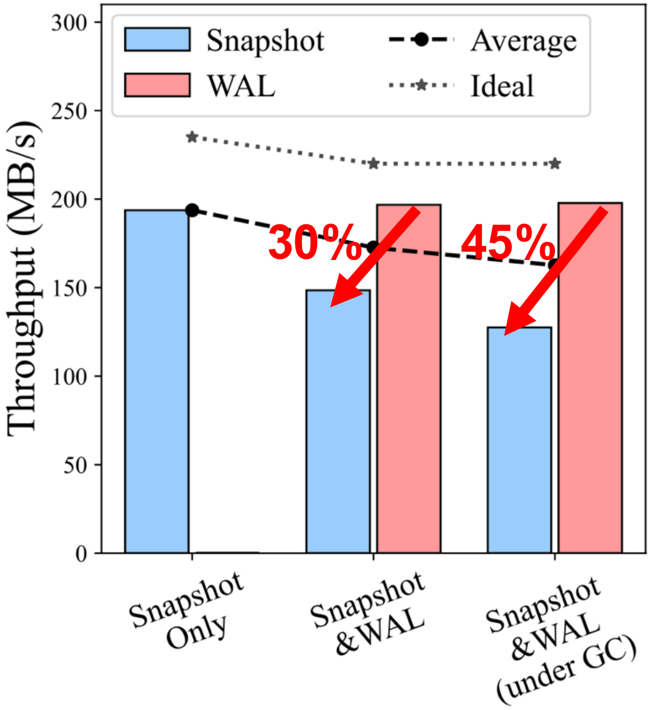
- **WAL**: Writes large buffered data in a single *write()* and *fsync()* call at a time threshold.
  vs
  Snapshot: Compresses each object individually and writes each compressed object with many *write()* calls.

O3: Kernel I/O path ignores per-process write patterns.





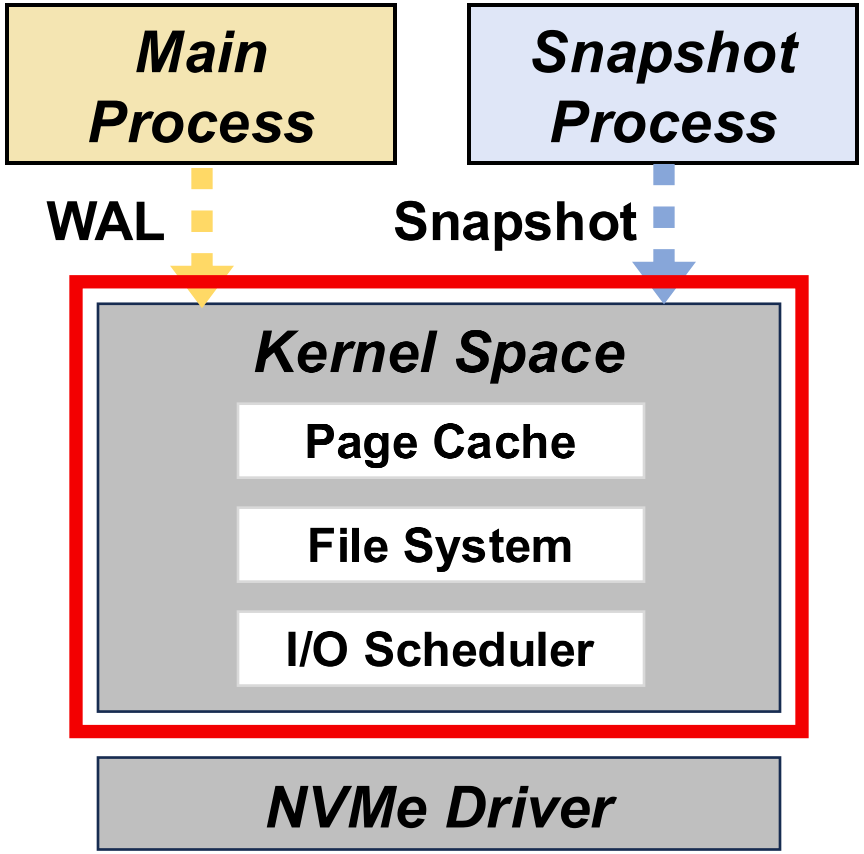
(a) Snapshot Time Distribution

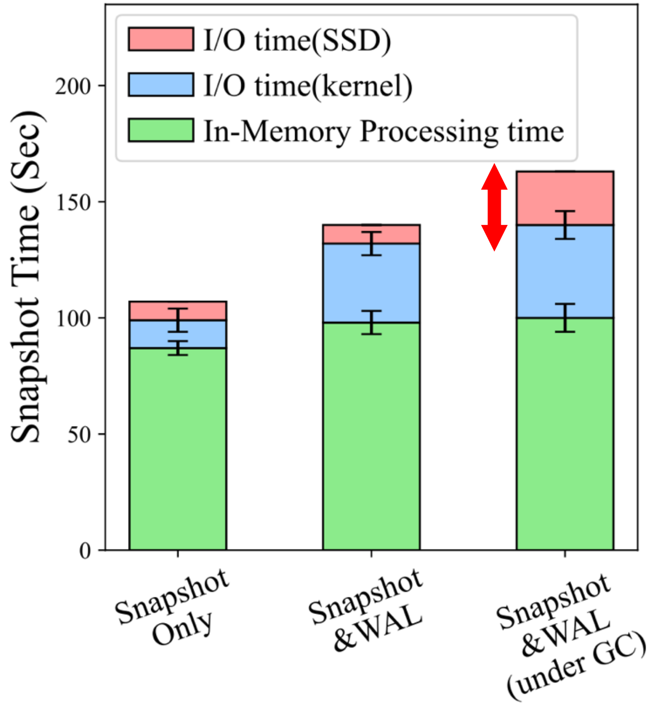


(b) Throughput Analysis

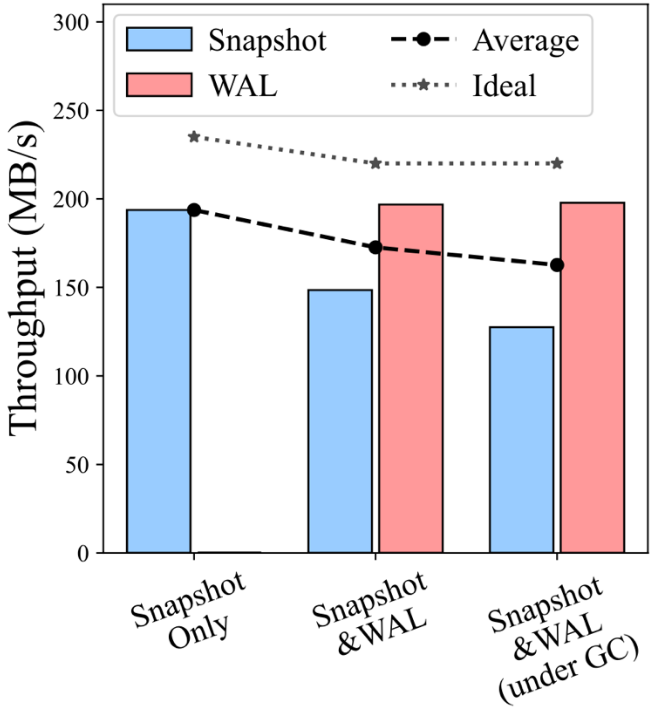
- When the Page Cache is busy (e.g., dirty page flush), a process attempting *write()* can be blocked.
- The kernel I/O path does not recognize these per-process write patterns, causing the snapshot process with its more frequent *write()* calls to be blocked more often.

O4: Kernel I/O path lacks sufficient mechanisms to eliminate SSD GC.





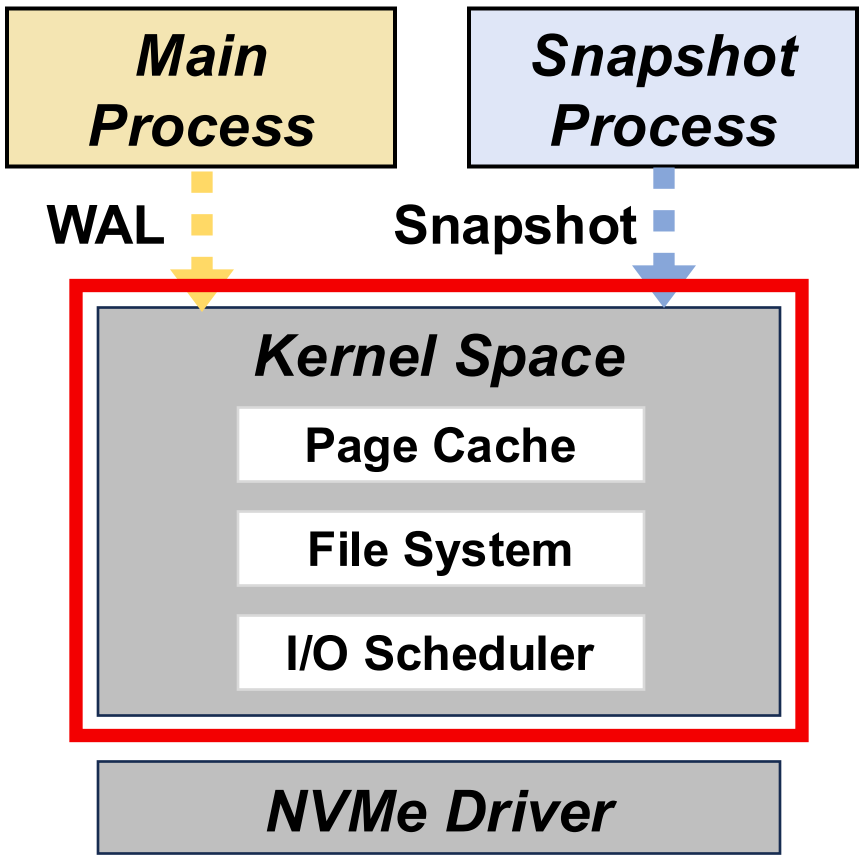
(a) Snapshot Time Distribution

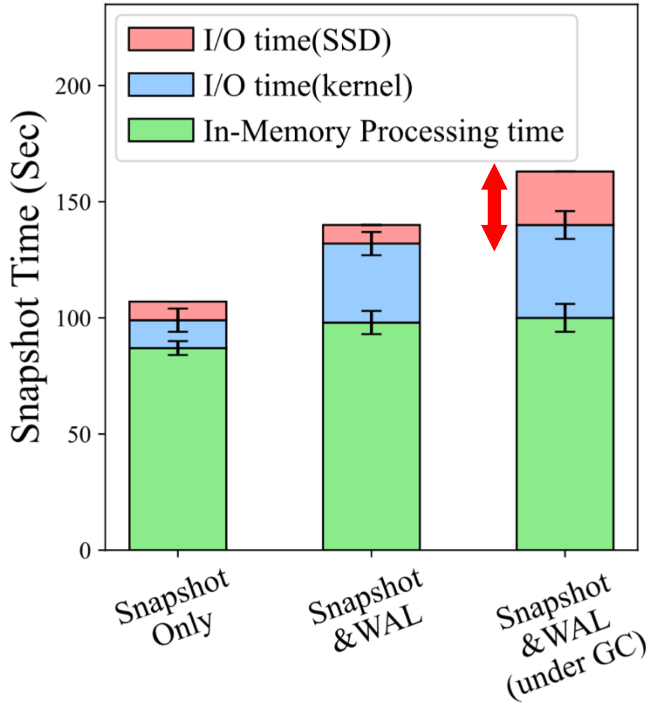


(b) Throughput Analysis

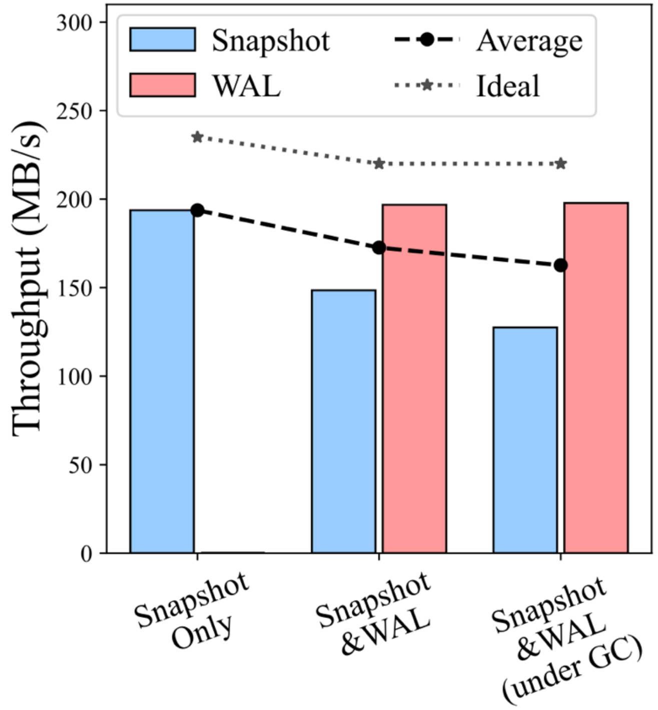
- **WAL & WAL-Snapshot**: Short-lived, frequently replaced in write-heavy workloads.
  vs
  On-Demand Snapshot: Long-lived, created manually by admin or at long intervals to preserve data.

O4: Kernel I/O path lacks sufficient mechanisms to eliminate SSD GC.





(a) Snapshot Time Distribution

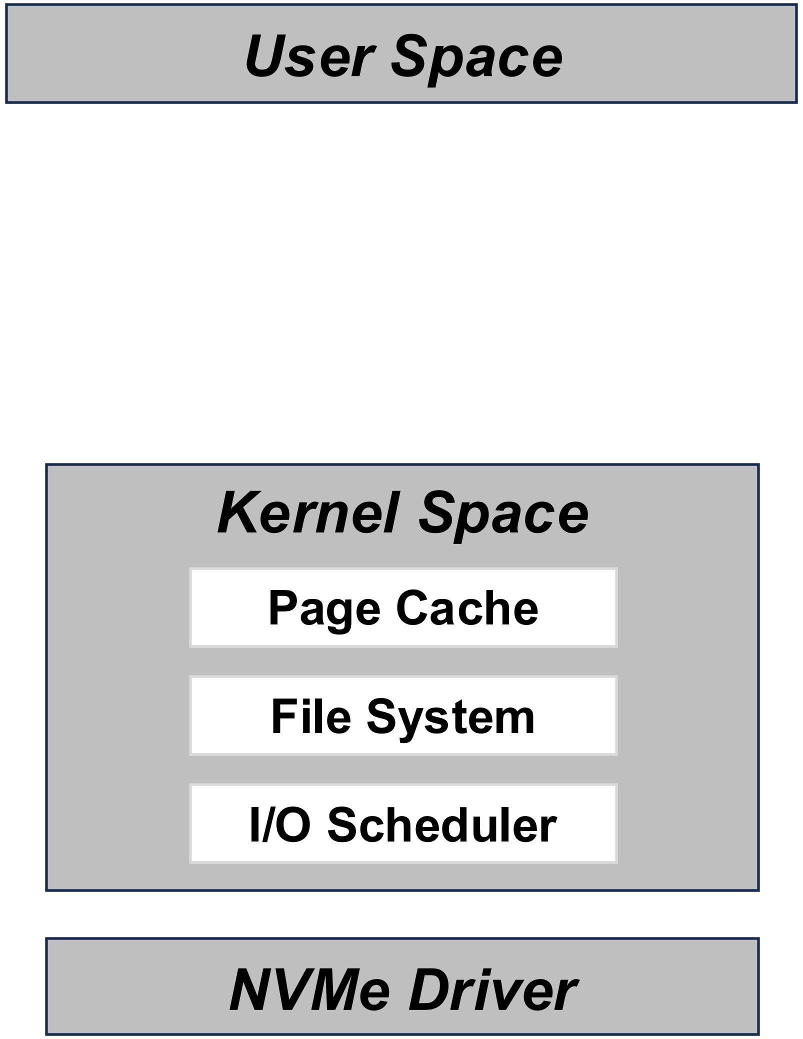


(b) Throughput Analysis

- Although recent XFS updates are attempting to support this issue, using it does not resolve the three issues mentioned earlier.

Contents

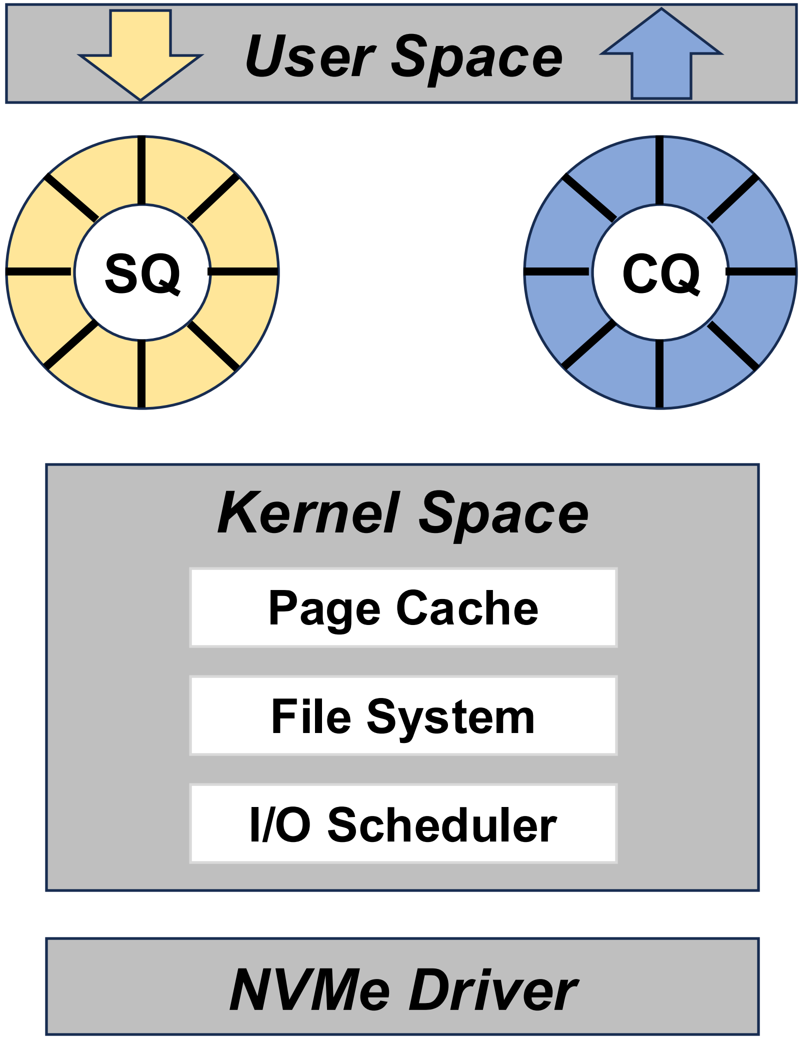
Opportunity 1: I/O Passthru



- I/O passthru[1] is a new I/O path introduced last year.
- I/O passthru is upstreamed in the Linux kernel.

[1] 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).

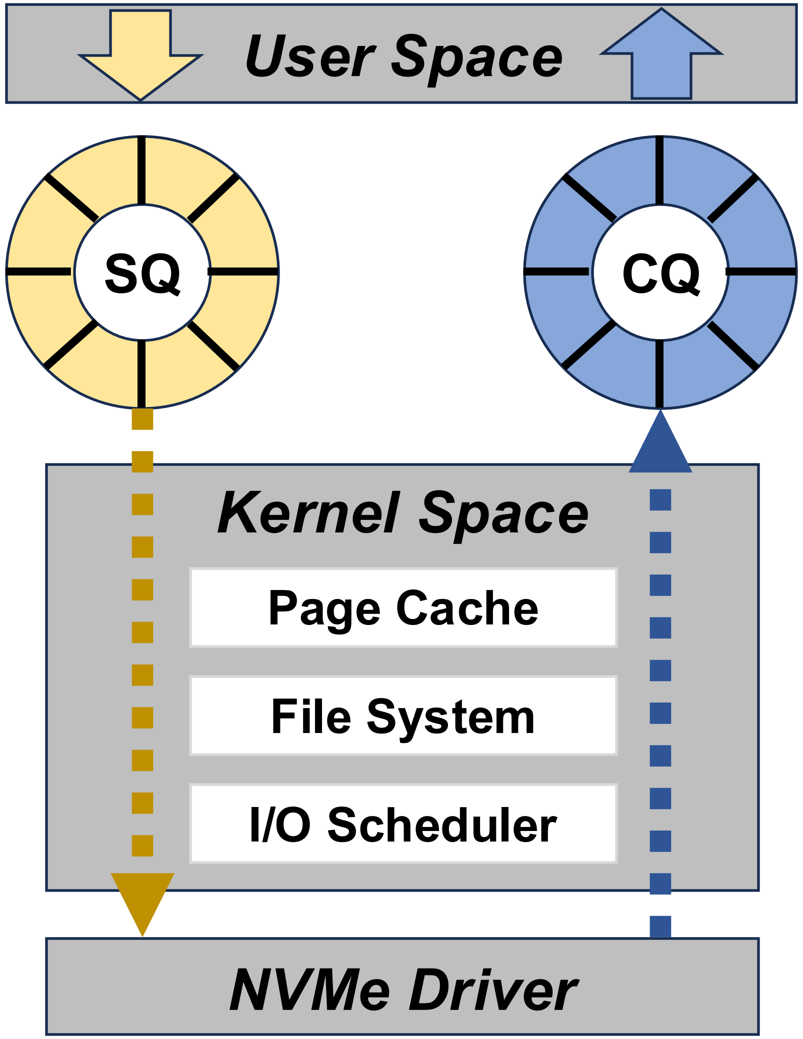
Opportunity 1: I/O Passthru



- I/O passthru[1] is a new I/O path introduced last year.
- I/O passthru is upstreamed in the Linux kernel.
- Runs based on the io_uring API
  - Reduces system call overhead.
  - Allows each process to use independent uring configurations.

[1] 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).

Opportunity 1: I/O Passthru

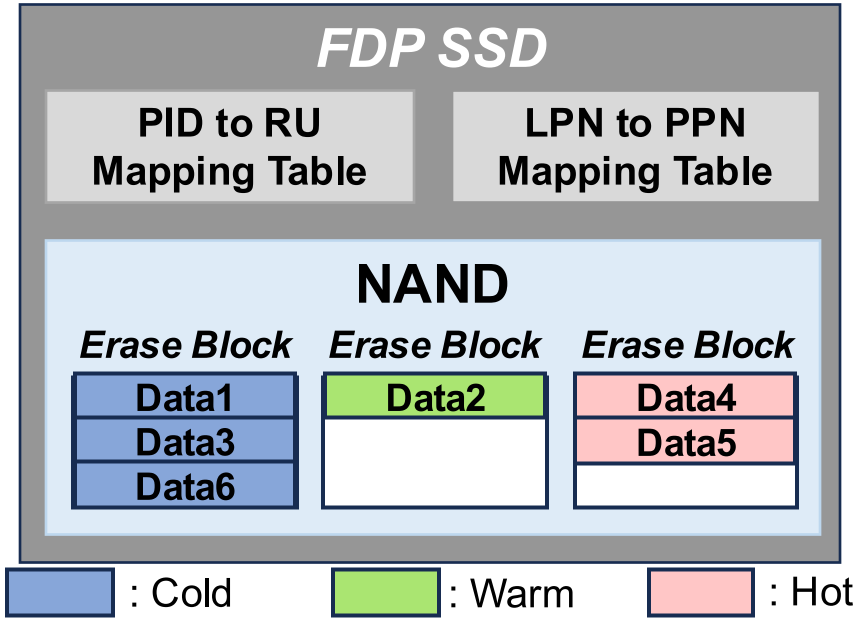


- I/O passthru[1] is a new I/O path introduced last year.
- I/O passthru is upstreamed in the Linux kernel.
- Runs based on the io_uring API
  - Reduces system call overhead.
  - Allows each process to use independent uring configurations.
- Bypasses kernel layers
  - Enables complete separation of I/O paths across processes.
  - Resolves scalability and contention issues.

[1] 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).

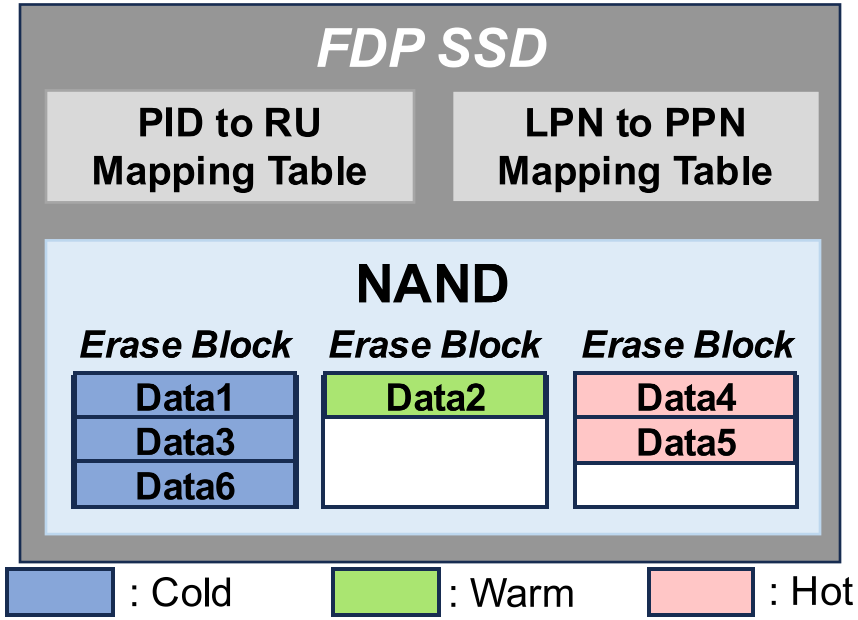
Opportunity 2: FDP SSD

- A way to solve SSD GC problem is to place data in different NAND Erase blocks according to data **lifetime**.



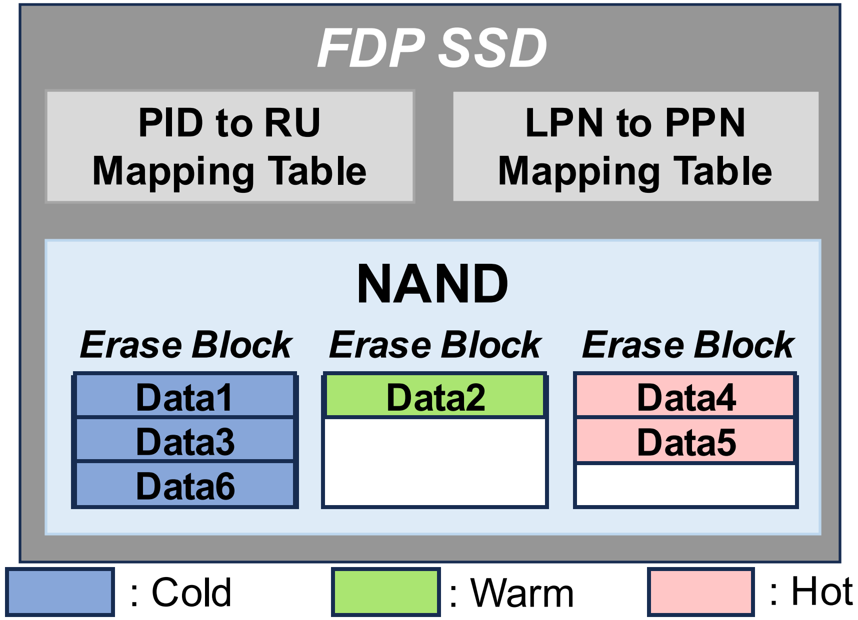
Opportunity 2: FDP SSD

- A way to solve SSD GC problem is to place data in different NAND Erase blocks according to data **lifetime**.
- The recently introduced Flexible Data Placement SSD, or FDP SSD, can address this issue.

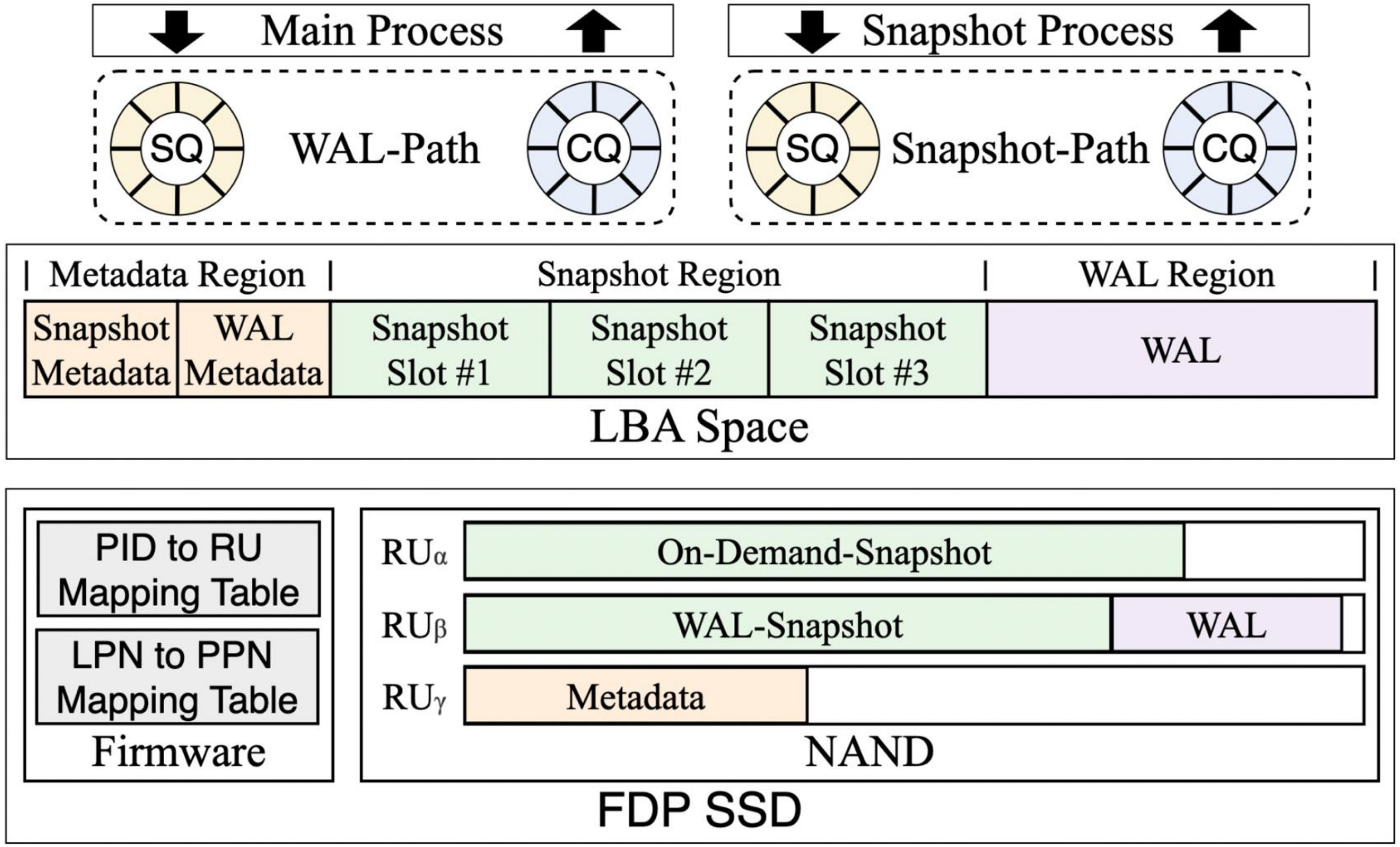


Opportunity 2: FDP SSD

- A way to solve SSD GC problem is to place data in different NAND Erase blocks according to data **lifetime**.
- The recently introduced Flexible Data Placement SSD, or FDP SSD, can address this issue.
- I/O Passthru is compatible with the latest NVMe commands that can utilize FDP SSD.
- By using an FDP SSD, we can separate the **WAL and WAL-Snapshot** from the **On-Demand Snapshot**, thereby eliminating garbage collection.

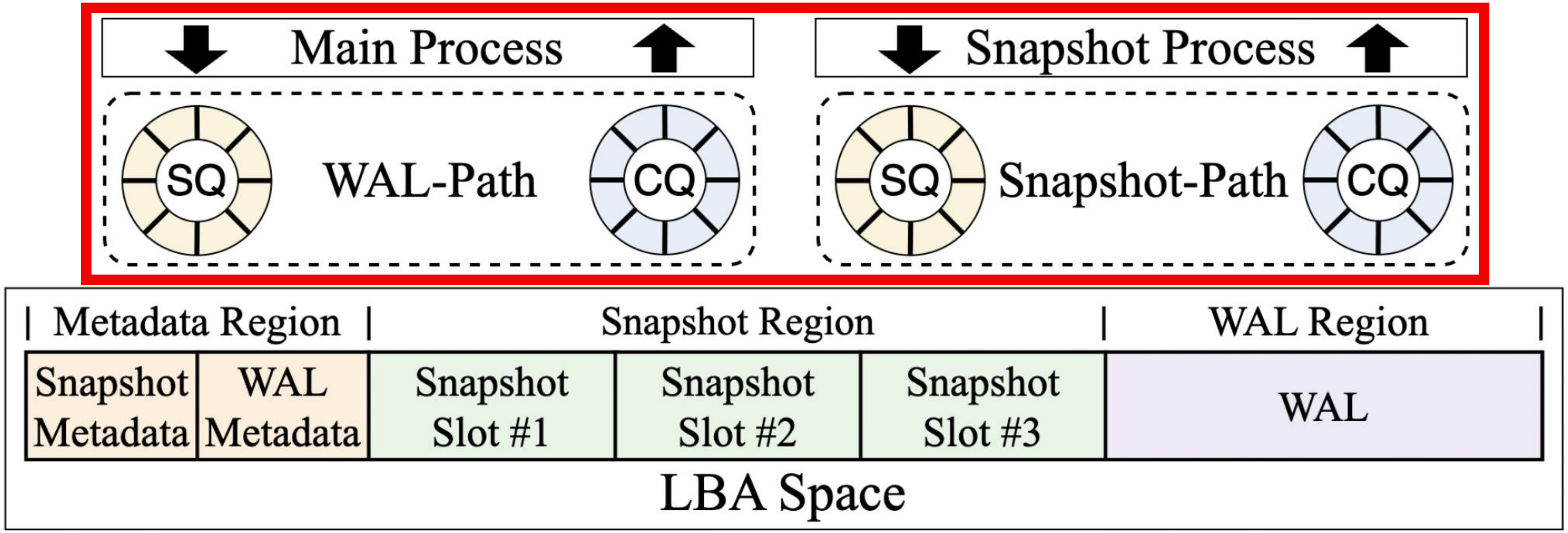


Overall Design of SlimIO



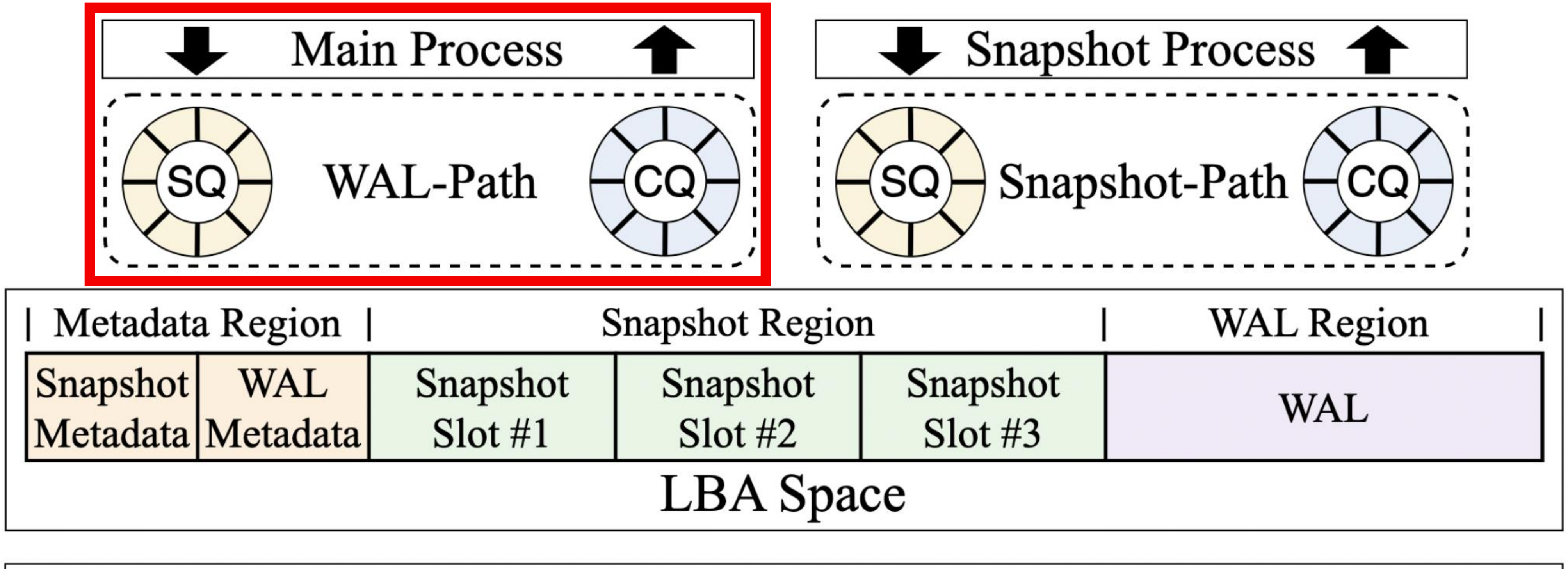
Snapshot–WAL Separation via I/O Passthru

- Processes sharing same I/O path, leading to contention and scalability issues.
- Therefore, we separate WAL and Snapshot I/O paths via I/O Passthru.
- I/O Passthru is based on io_uring, it reduces syscall overhead.



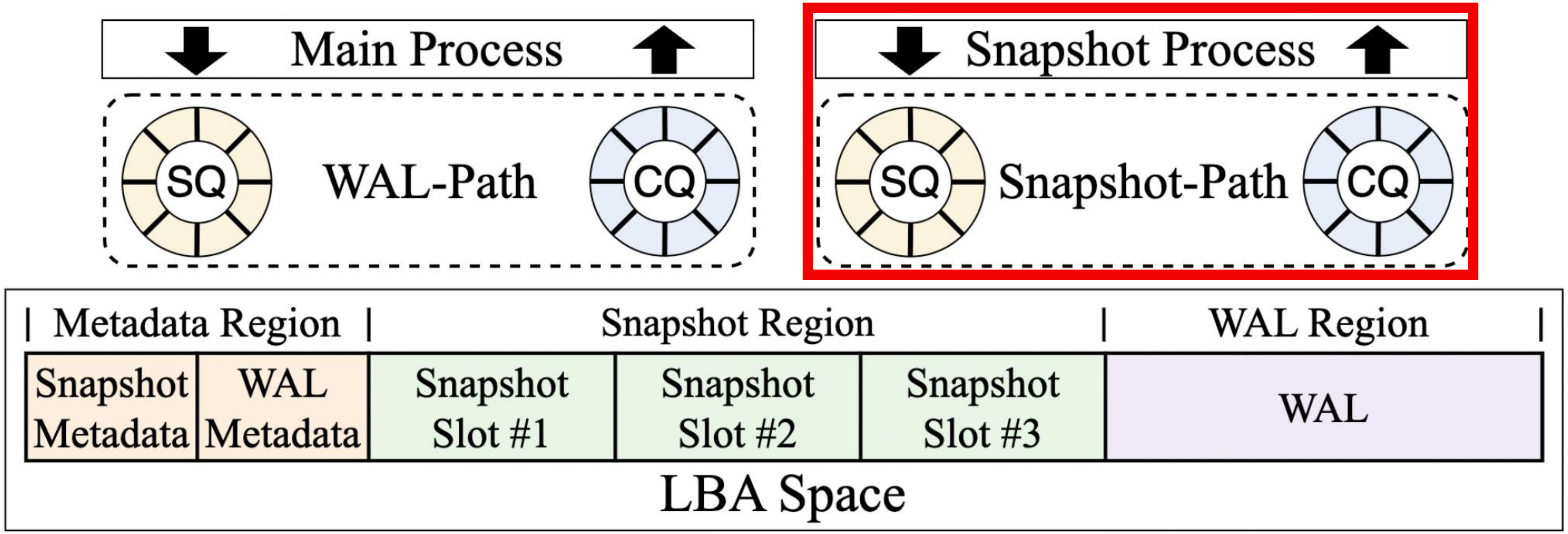
Snapshot–WAL Separation via I/O Passthru

- Redis initializes SQ and CQ for WAL via I/O passthru at startup.
- A dedicated CQ handling thread is also spawned to process completions.
- SlimIO preserves the original Redis WAL logging policy without modification.



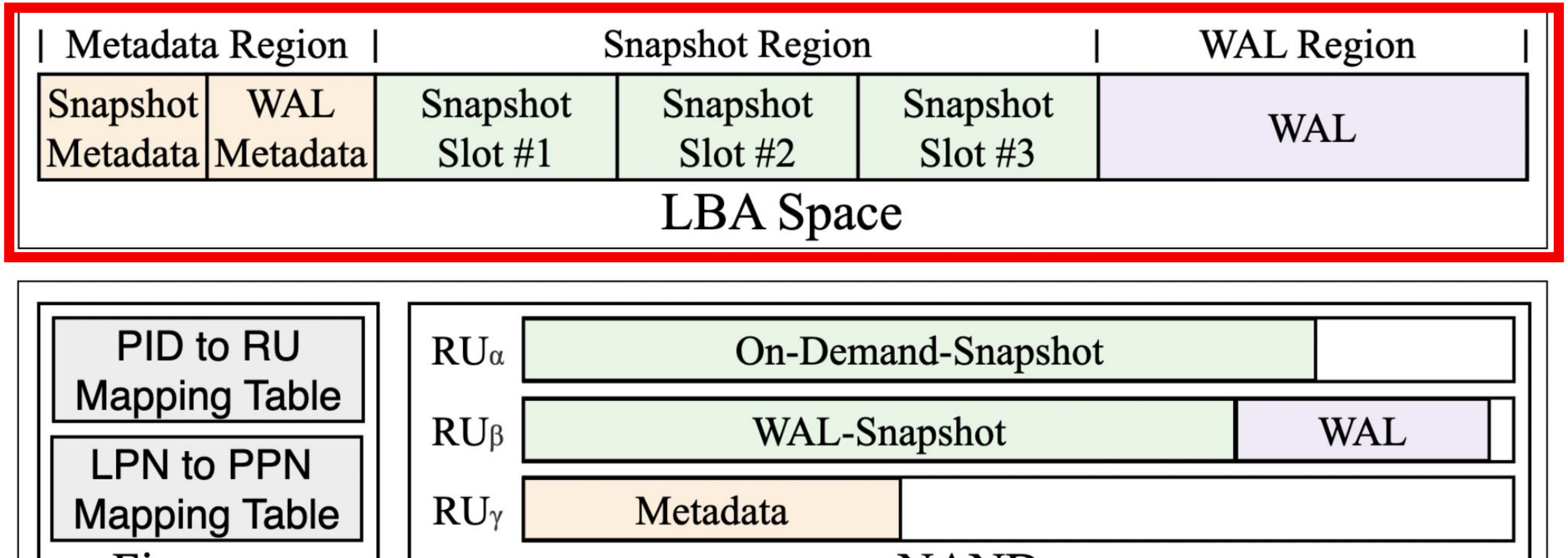
Snapshot–WAL Separation via I/O Passthru

- Snapshot Process initializes its own SQ and CQ via I/O passthru.
- This Snapshot-Path runs in *SQPOLL* mode.
- SlimIO preserves the original Redis Snapshot module without modification.



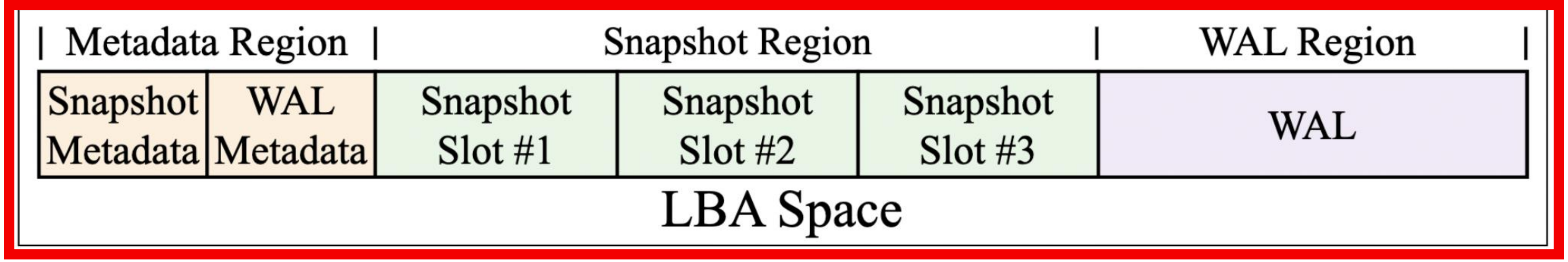
LBA Space Management

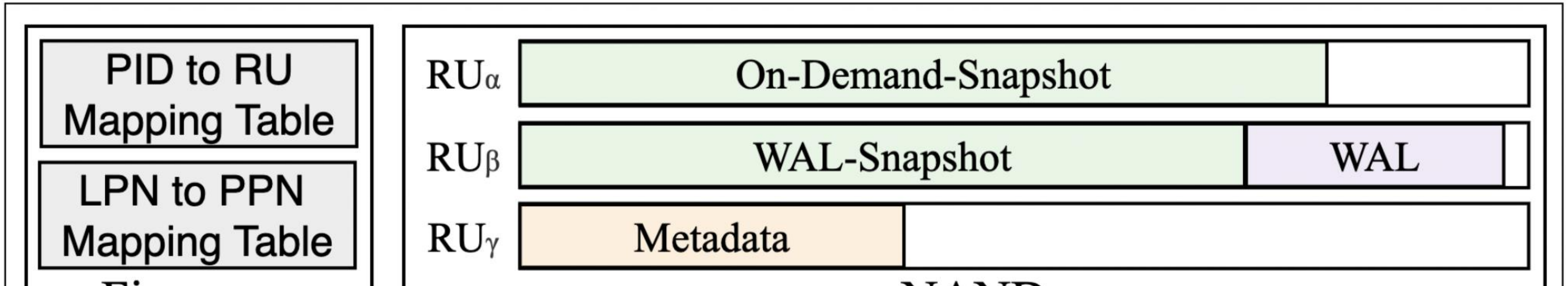
- **Key Challenge:** I/O Passthru bypasses the file system, requiring explicit LBA space management.
- However, Redis mainly use sequential writes, simplifying LBA management.



LBA Space Management

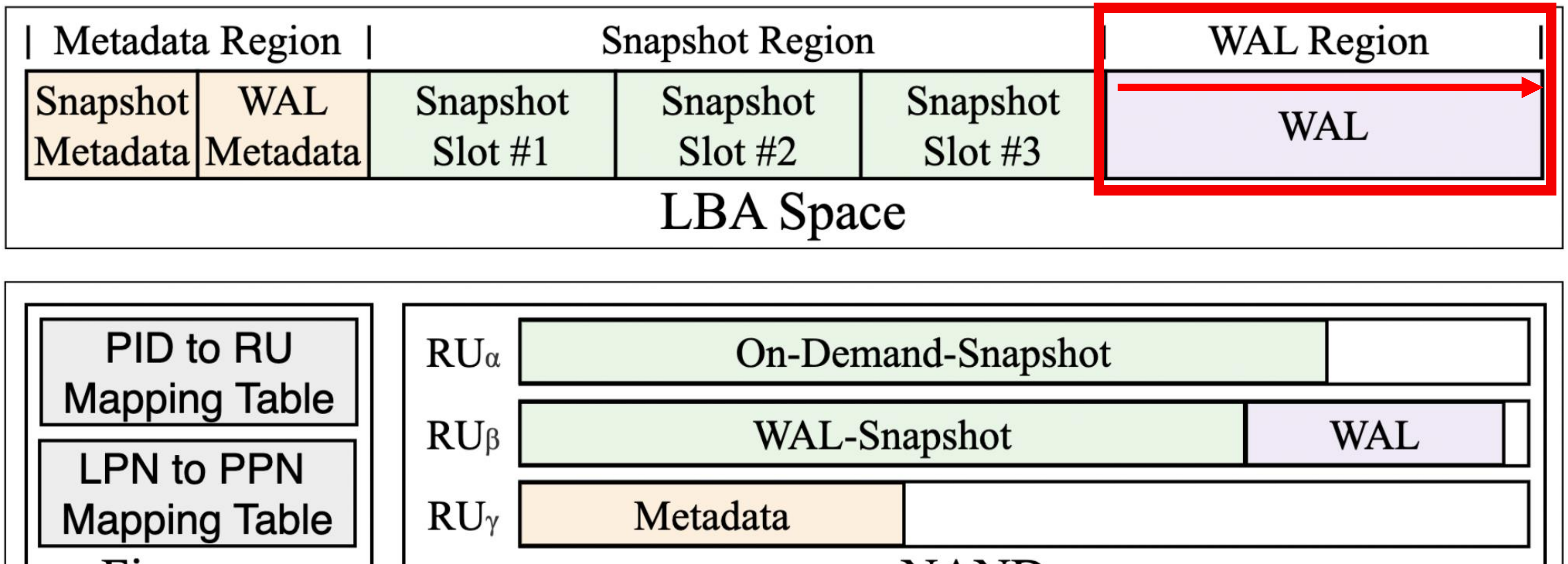
- To remain fully compatible with Redis, SlimIO divides the LBA space into three regions:
  - Metadata Region
  - Snapshot Region
  - WAL Region





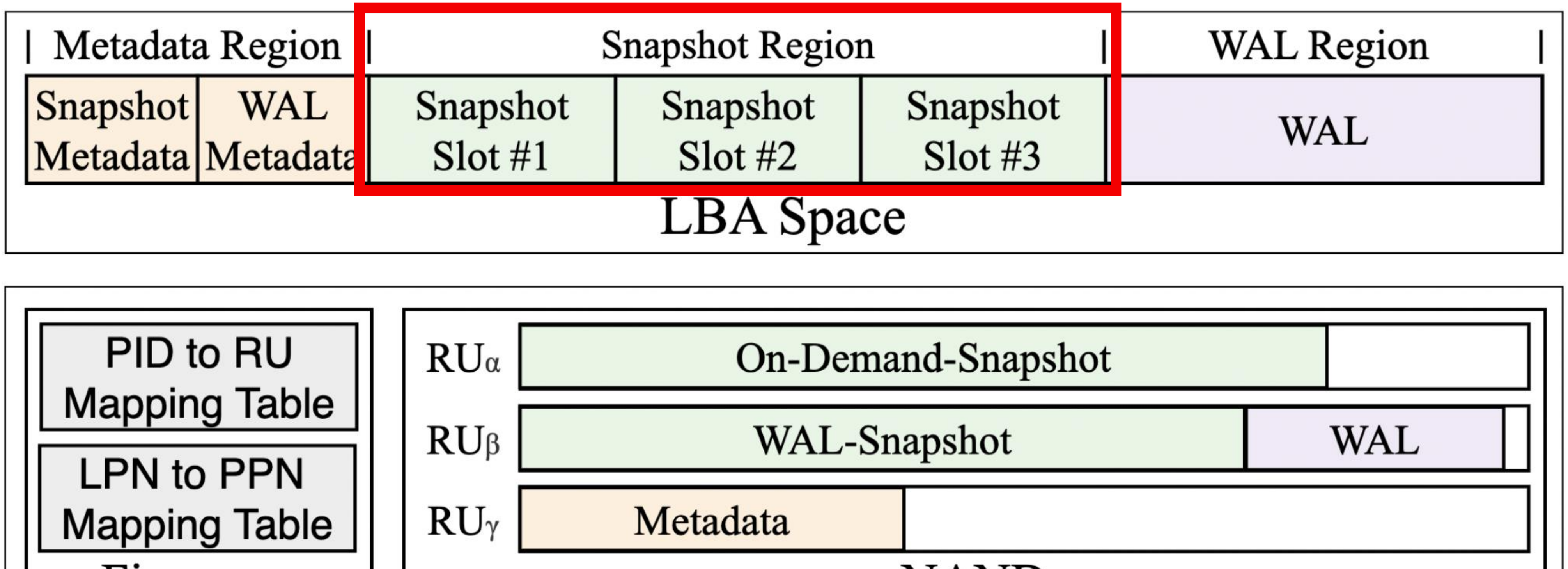
LBA Space Management

- In the WAL Region, WAL entries are written sequentially using a sliding window manner.
- The previous WAL is only deallocated after a new WAL-Snapshot generation is successful.



LBA Space Management

- The snapshot region has two slots for WAL and On-Demand snapshot, plus a reserve slot for failures.
- This design mirrors Redis's behavior of deleting old data only after a new snapshot is safely completed.



LBA Space Management

- Initial state:
  - Slot #1 = WAL-Snapshot slot,
  - Slot #2 = reserve slot,
  - Slot #3 = On-Demand-Snapshot slot.

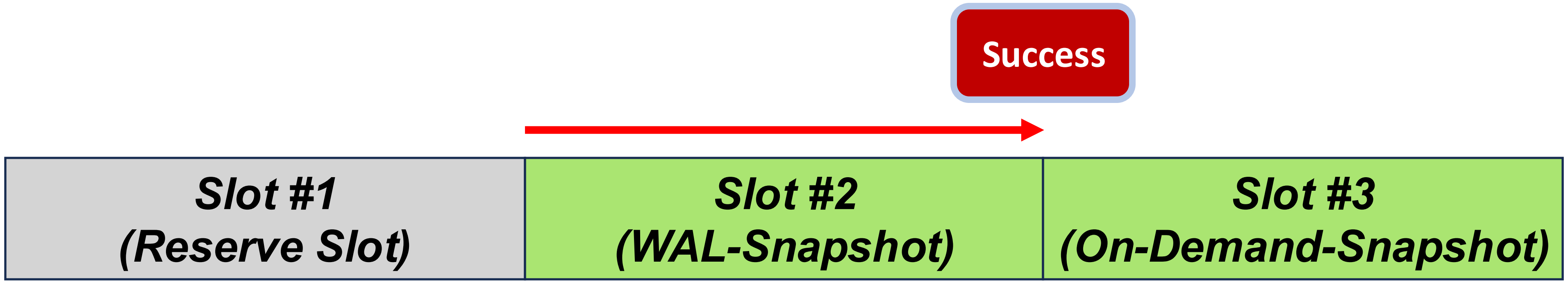
Slot #1 (WAL-Snapshot)	***Slot #2 (Reserve Slot)***	***Slot #3 (On-Demand-Snapshot)***

LBA Space Management

- Initial state:
  - Slot #1 = WAL-Snapshot slot,
  - Slot #2 = reserve slot,
  - Slot #3 = On-Demand-Snapshot slot.
- A new WAL-snapshot is first written to the reserve slot.

LBA Space Management

- Initial state:
  - Slot #1 = WAL-Snapshot slot,
  - Slot #2 = reserve slot,
  - Slot #3 = On-Demand-Snapshot slot.
- A new WAL-snapshot is first written to the reserve slot.
- If successful, it becomes a valid slot, and the old one is reused as the new reserve slot.



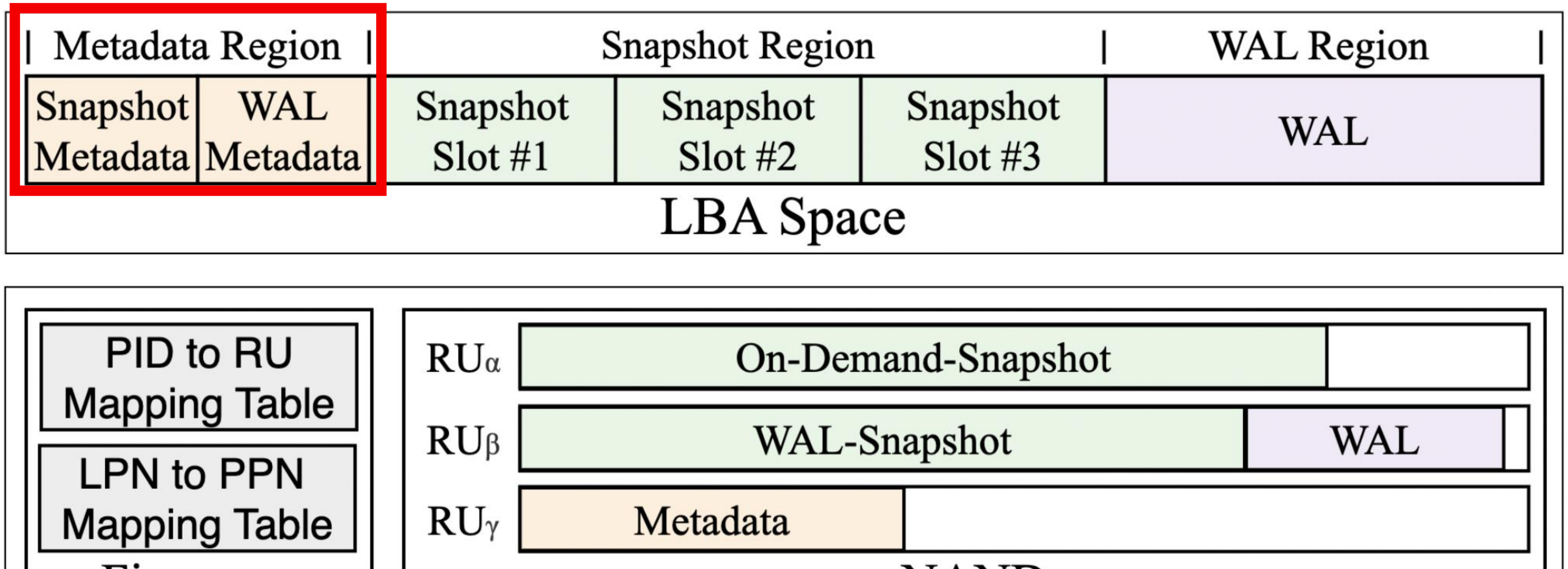
LBA Space Management

- Initial state:
  - Slot #1 = WAL-Snapshot slot,
  - Slot #2 = reserve slot,
  - Slot #3 = On-Demand-Snapshot slot.
- A new WAL-snapshot is first written to the reserve slot.
- If successful, it becomes a valid slot, and the old one is reused as the new reserve slot.
- Each of WAL-Snapshot and On-Demand Snapshot can exist only once, and since only one snapshot runs at a time, three slots are sufficient.

Slot #1 (Reserve Slot)	***Slot #2 (WAL-Snapshot)***	***Slot #3 (On-Demand-Snapshot)***

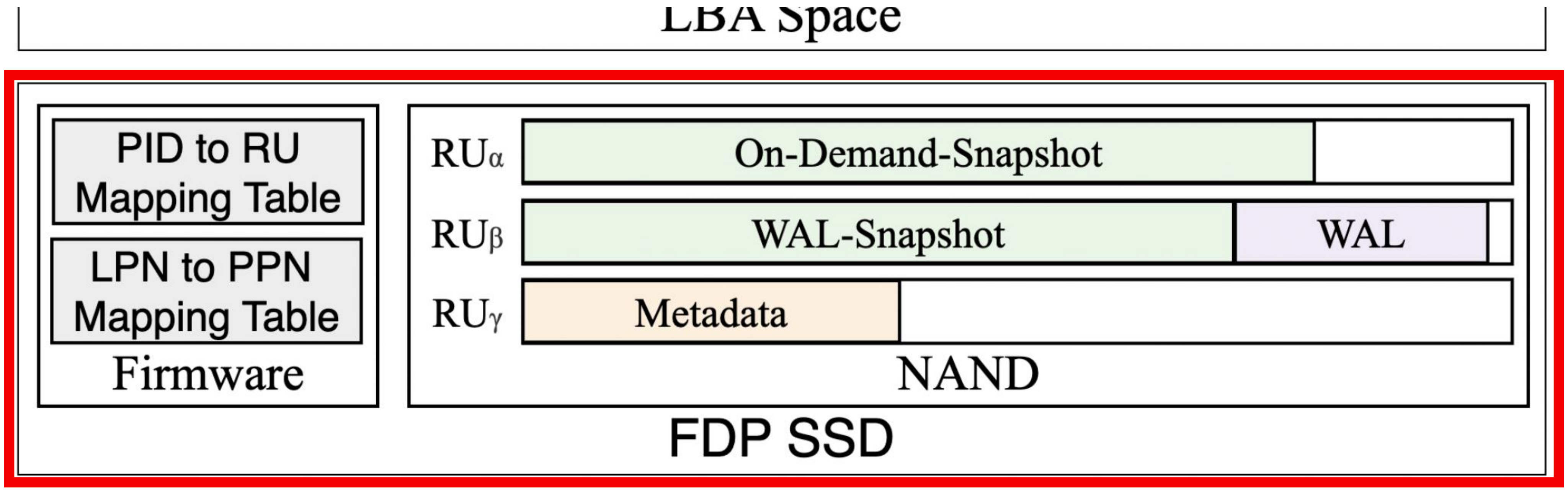
LBA Space Management

- All state information like the current WAL position and the roles of snapshot slots is stored in the Metadata Region, ensuring consistency and reliability of the LBA space.

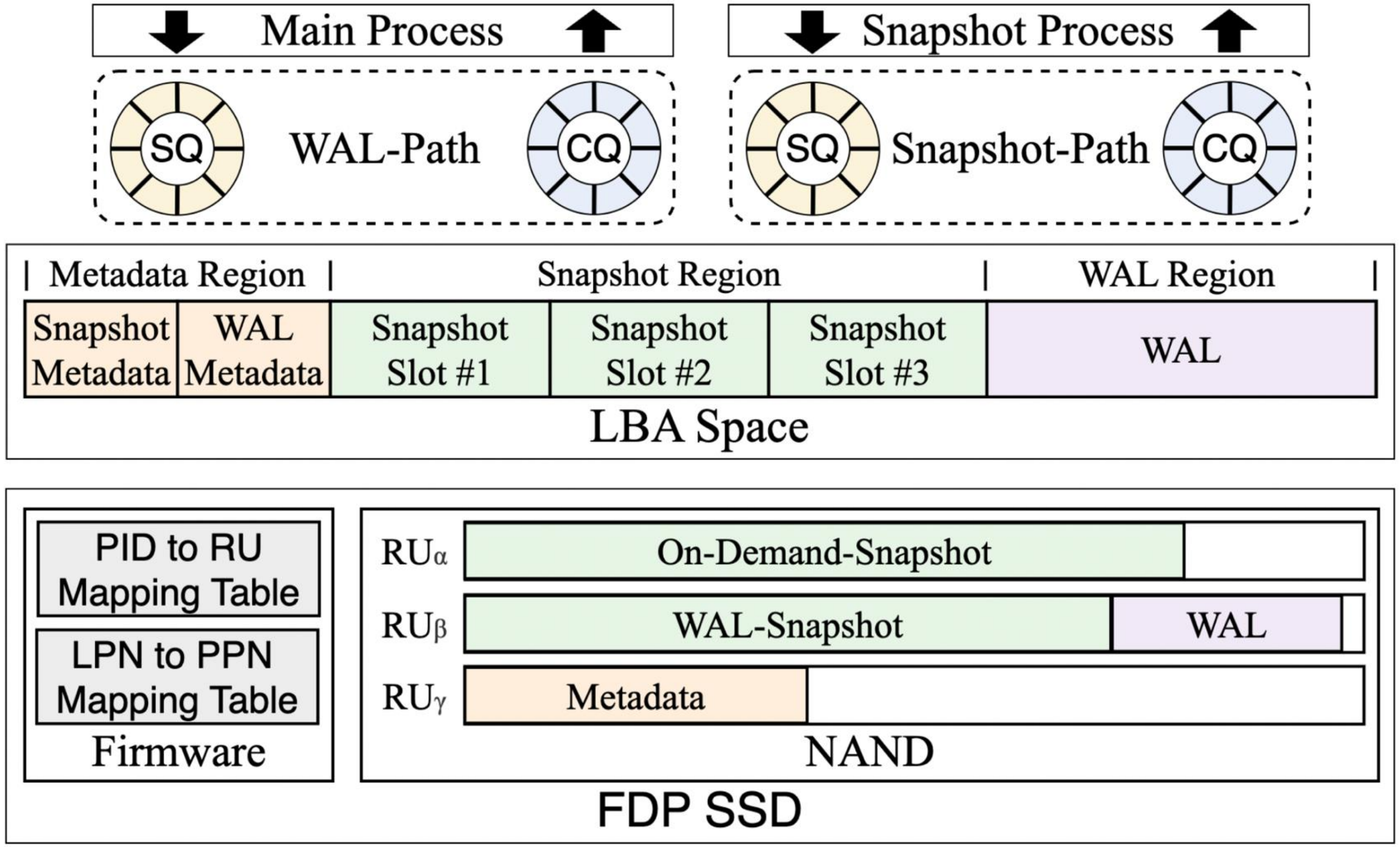


LBA Space Management

- I/O Passthru is compatible with the latest NVMe commands that can utilize FDP SSD.
- We use this to assign different Erase Block based on data lifetimes



Overall Design of SlimIO



Contents

Experimental Setup

HOST

- Intel Xeon Gold 5218R (32 cores)
- 377GB DRAM

GUEST

- 12 cores
- 55GB DRAM
- Linux Kernel 6.7.9
- Redis v.7.4.2
- Baseline: F2FS

FDP SSD

- FEMU
- R/W/E Latency: 40/200/2000 us
- 1GB Reclaim Unit
- 8 CH x 8 WAY
- 180GB Total Capacity

Workloads

- **Common Setting:**
  - WAL Size Limit: ~50GB
  - Two WAL time-threshold–based flush policies (default):
    - Periodical-Log – flush every 1 sec
    - Always-Log – flush on every write
  - Both policies are evaluated in all experiments.
  - Baseline I/O scheduler: ‘none’

Workloads

- **Redis Benchmark:**
  - 50 concurrent clients (default)
  - 8-byte keys and 4096-byte values
  - Each experiment issues 28M SETs, repeated 5 times (total 140M SETs).
  - An On-Demand Snapshot is triggered after each run.
  - In total, 15 Snapshots are generated.
- **YCSB – A:**
  - 8 threads
  - 8-byte keys and 2048-byte values
  - 115M operations (0.5 : 0.5 = GET : SET), executed once.
  - In total, 2 Snapshots are generated.

Overall Evaluation

- ***"WAL Only"*** represents the period where no snapshot is executed.
- ***"WAL & Snapshot"*** represents the period where a snapshot is being executed.

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

Overall Evaluation

- ***"RPS"*** denotes Requests per Second.
- ***"Average RPS"*** represents the overall throughput, including both WAL Only and WAL & Snapshot phases.

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

Overall Evaluation

- ***"Snapshot Time"*** represents the average time to complete a single snapshot.
- Redis benchmark runs 15 snapshots → total time = 15 × snapshot time
- YCSB – A runs 2 snapshots → total time = 2 × snapshot time

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

Overall Evaluation

- ***"SSD WAF"*** indicates the extent of valid copies made during garbage collection.
- A WAF of 1 means that no valid copies are generated.

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

Overall Evaluation – Snapshot Time

- **Redis benchmark:** ~25% reduction in snapshot time
- **YCSB - A:** ~10% reduction (more small values → longer compression time)

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

Overall Evaluation – RPS

- Due to reduced system call overhead and shorter snapshot time, the overall RPS significantly increased.

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

Overall Evaluation – RPS

- WAL & Snapshot phase: minimal performance gap
- RPS drop → fork() Copy-on-Write (memory copy + lock contention)
- SlimIO shortens snapshot duration, reducing the impact period

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

Overall Evaluation – Latency

- Better write performance + shorter snapshot time
  → lower SET & GET tail latency
- YCSB-A: 8 threads, no SSD GC, smaller values → lower latency

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

Recovery Performance

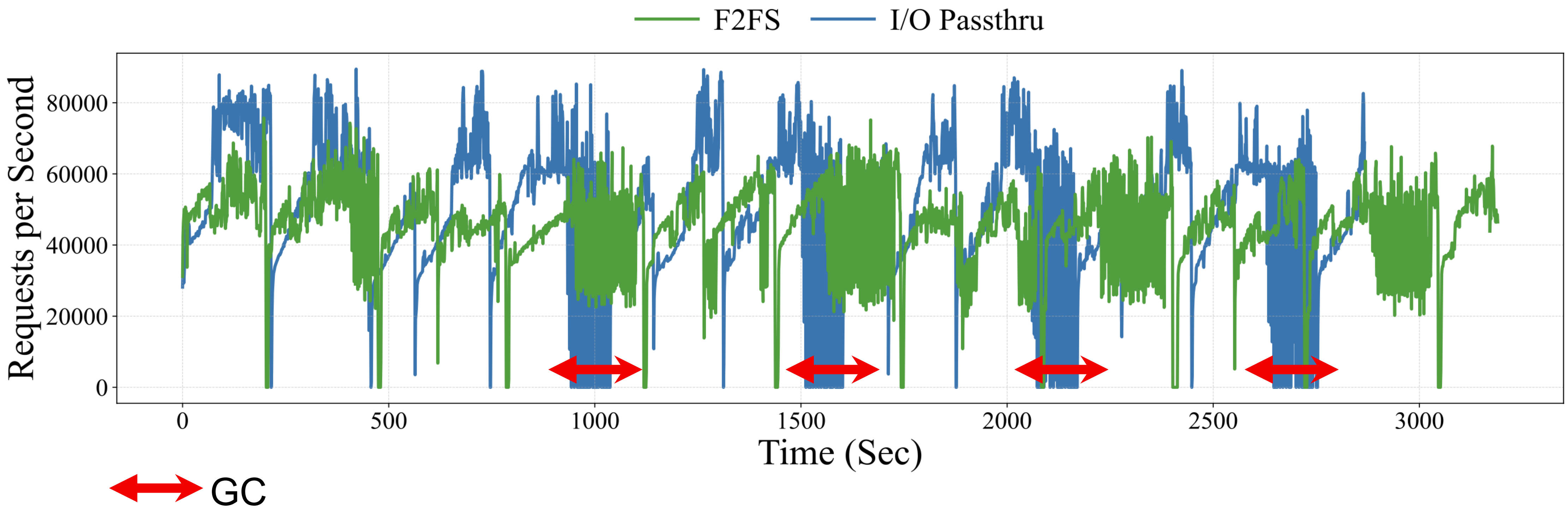
- The baseline uses the page cache for fast reads but still has high syscall overhead.
- SlimIO's read-ahead buffer, optimized for sequential I/O, removes syscall overhead for faster recovery.

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

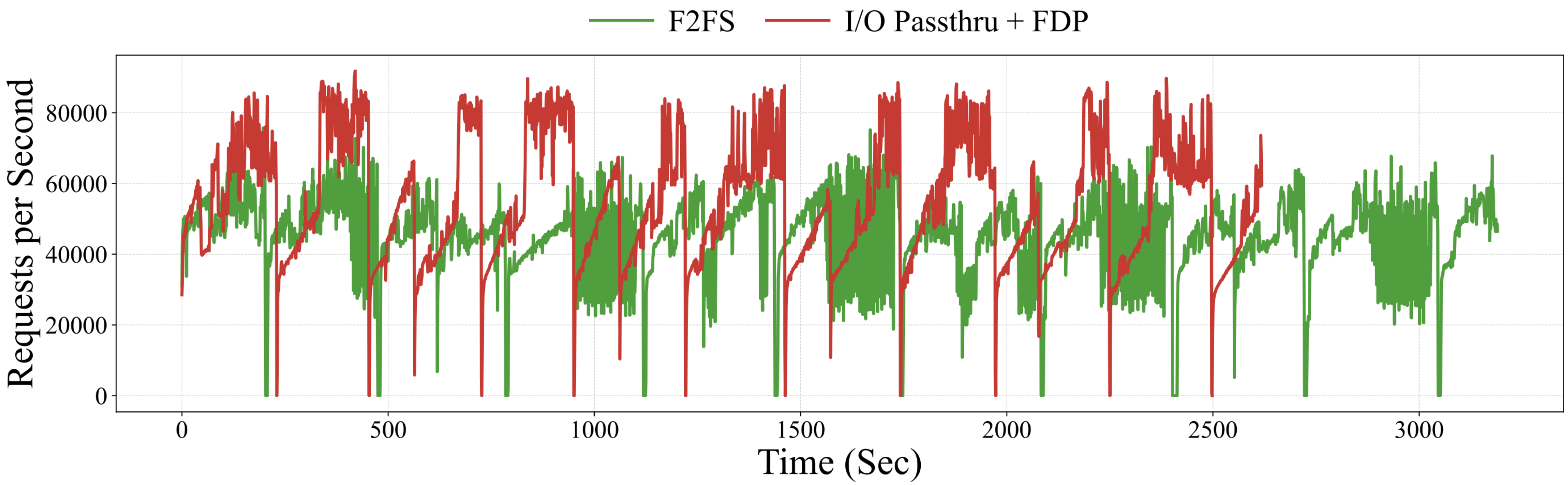
Microscopic Analysis on FDP SSD

- With minimal syscall overhead, SlimIO without FDP shows higher throughput under no-GC periods.
- However, during GC, RPS of SlimIO without FDP drop to zero.



Microscopic Analysis on FDP SSD

- Snapshot time reduced from 140–180 sec (F2FS) to 100 sec (SlimIO).
- Overall RPS also increased by 30%.



Contents

Conclusion

- This paper investigates prolonged snapshot times in Redis, caused by high syscall overhead, I/O interference between snapshot and WAL processes, and SSD GC.
- The traditional kernel I/O path cannot resolve these issues.
- We propose SlimIO, using io_uring-based I/O passthru to reduce syscall overhead and I/O interference, and an FDP SSD to eliminate GC-induced slowdown.
- Experiments show SlimIO significantly shortens snapshot duration and improves overall performance even during non-snapshot periods.

Thank you!

- Contact
  - Sangyun Lee / tkddbs9801@sogang.ac.kr
  - Youngjae Kim / youkim@sogang.ac.kr / https://sites.google.com/site/youkim/home
  - Data-Intensive Computing & Systems Laboratory https://discos.sogang.ac.kr/

<Camera-ready paper>

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, United States, 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

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767511

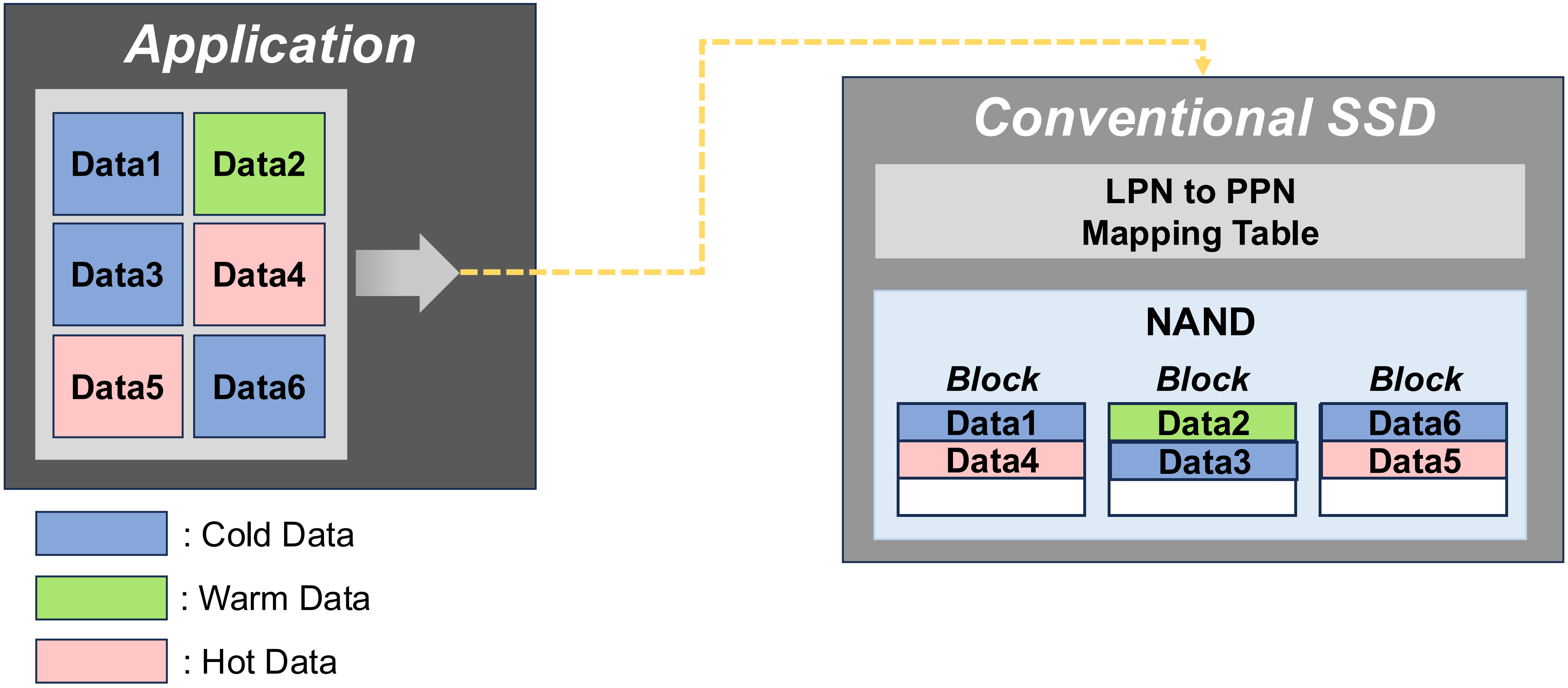
1 Introduction

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

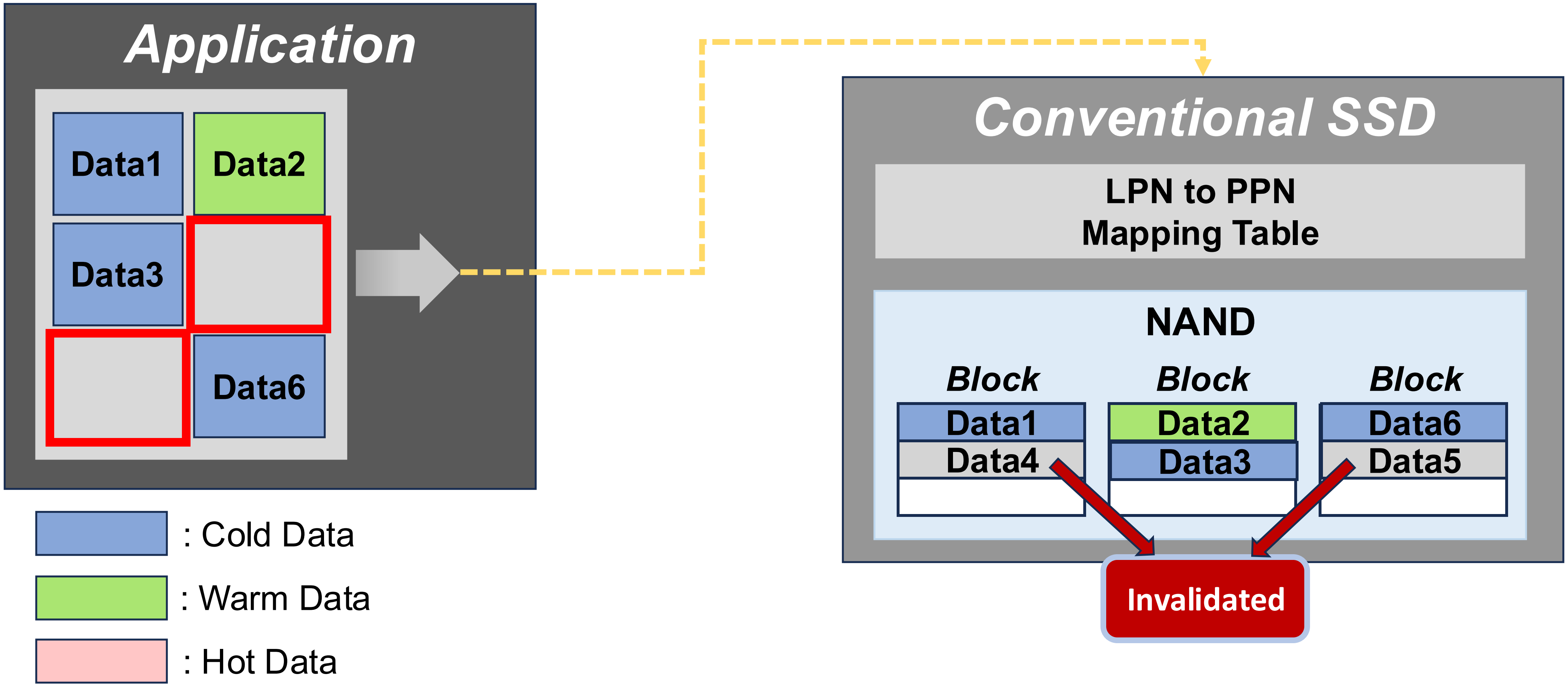
GC overhead in conventional SSD

- In conventional SSDs, data is mixed in NAND regardless of its lifetime.



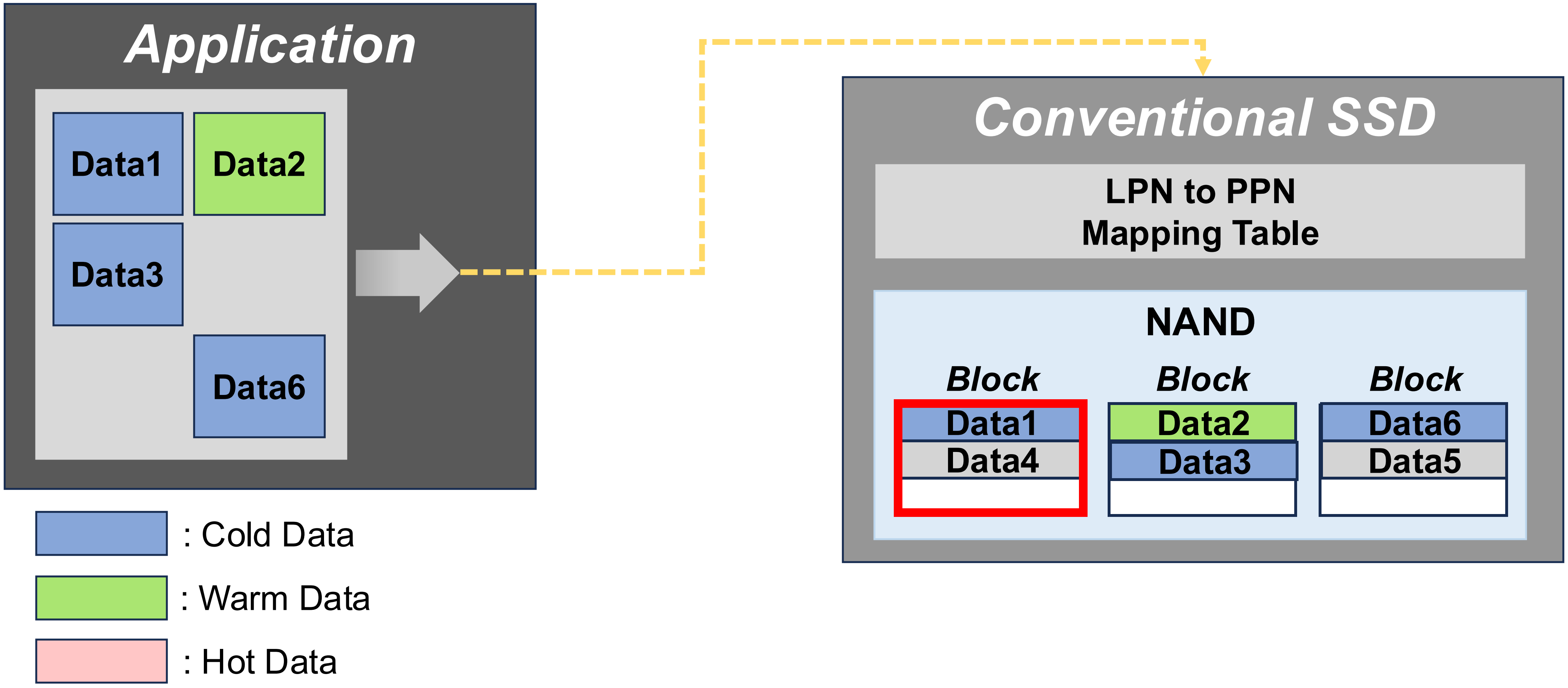
GC overhead in conventional SSD

- In conventional SSDs, data is mixed in NAND regardless of its lifetime.



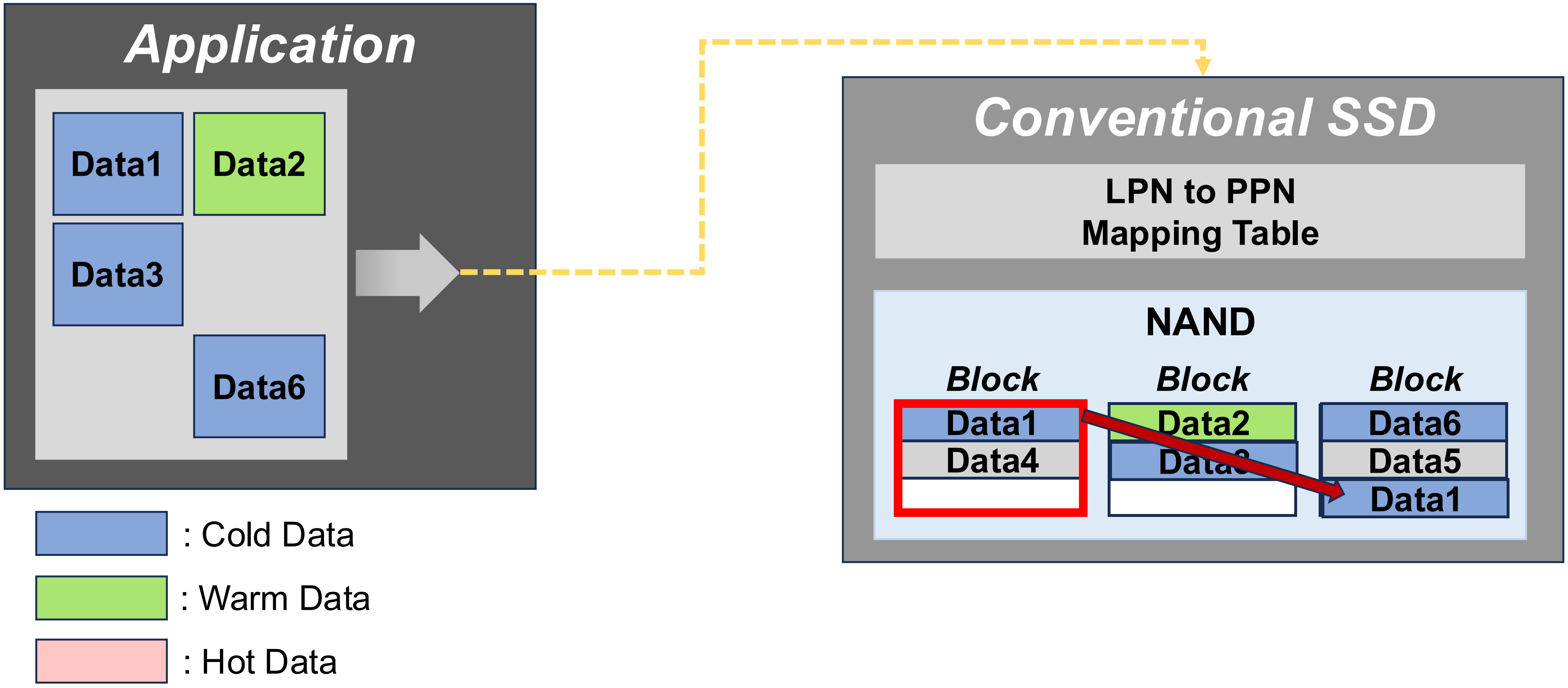
GC overhead in conventional SSD

- If the first block is erased, Data 1 must be copied to another block.



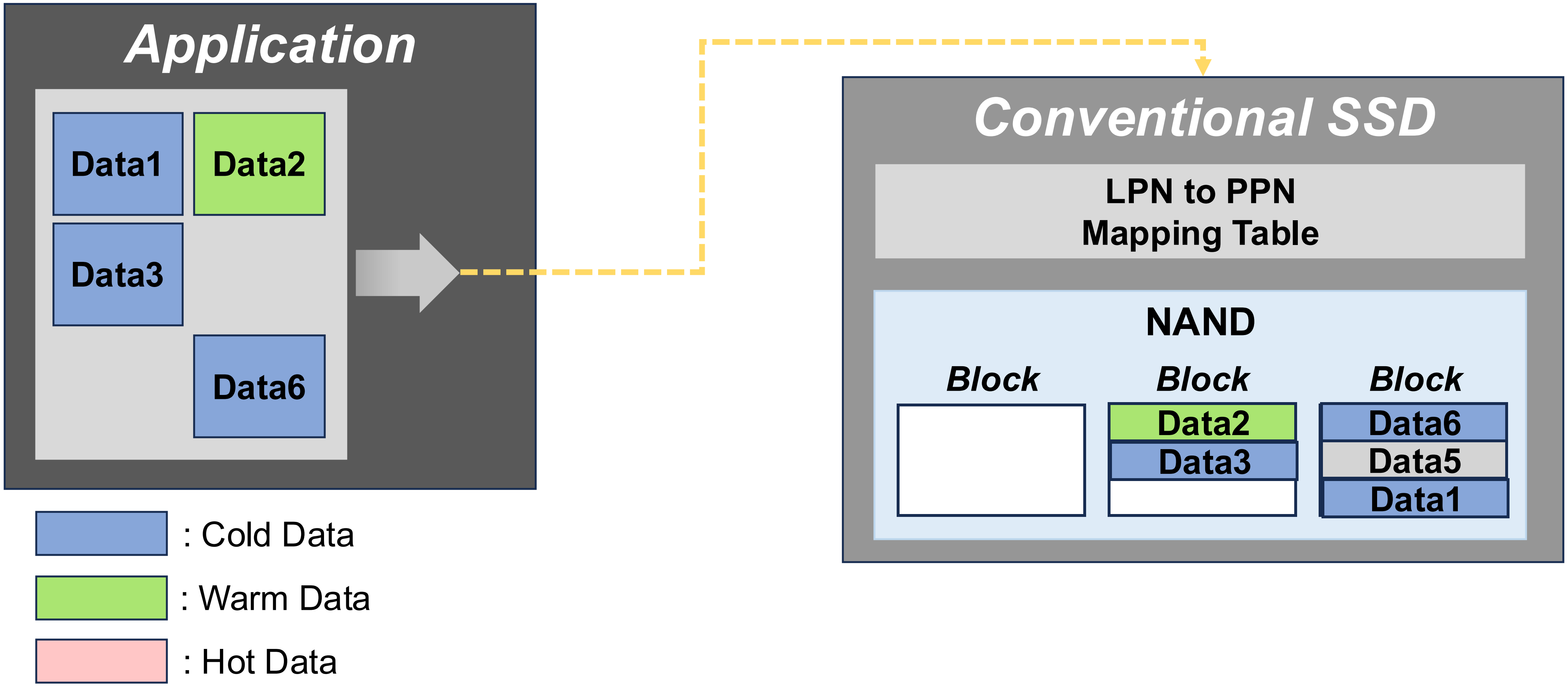
GC overhead in conventional SSD

- If the first block is erased, Data 1 must be copied to another block.



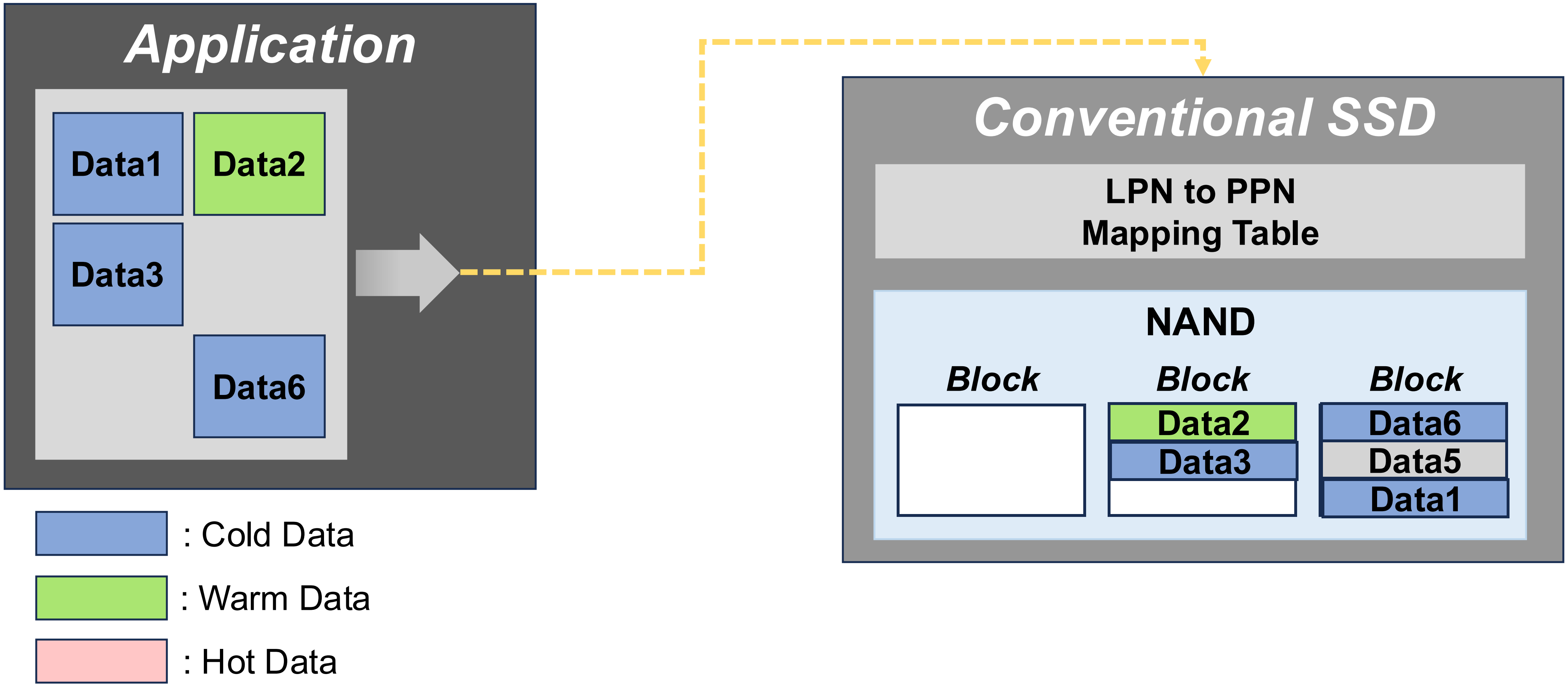
GC overhead in conventional SSD

- If the first block is erased, Data 1 must be copied to another block.



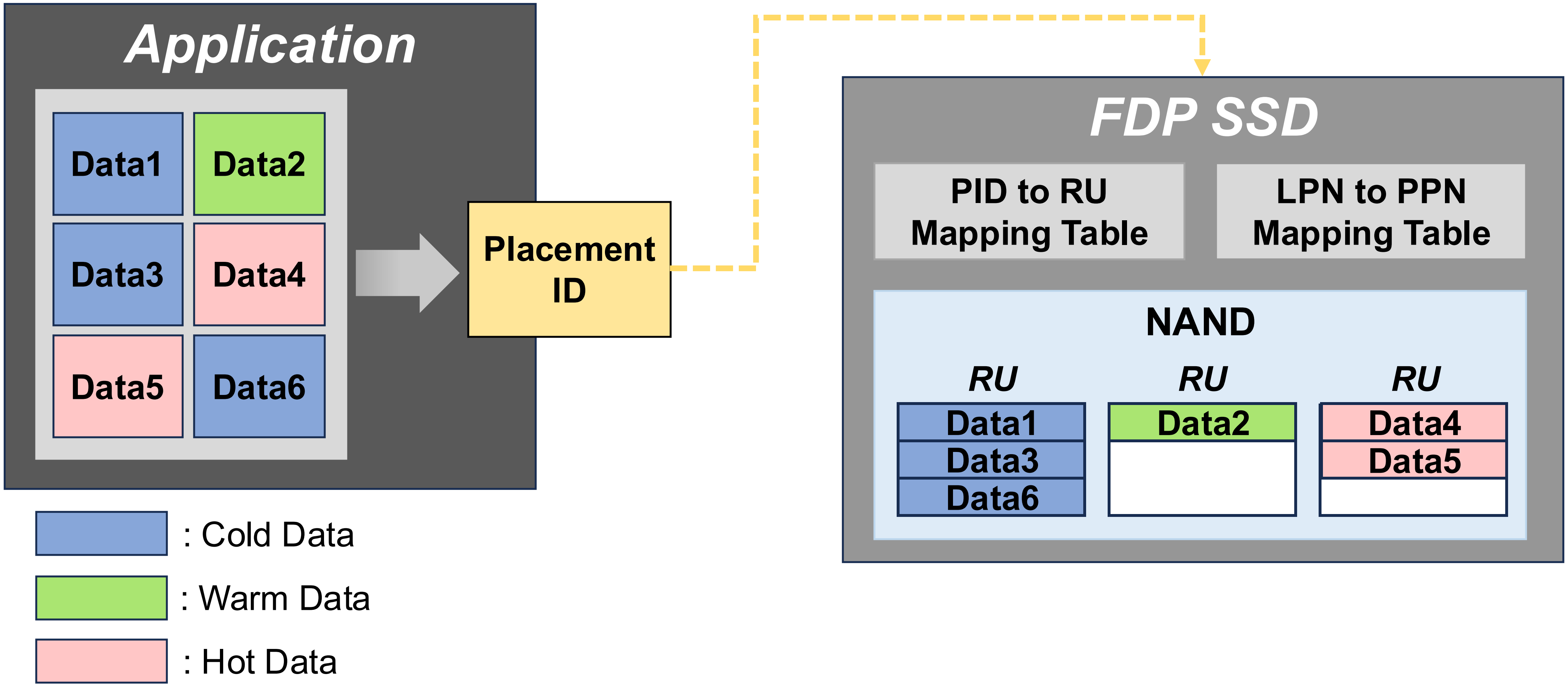
GC overhead in conventional SSD

- However, GC causes SSD wear and performance degradation.

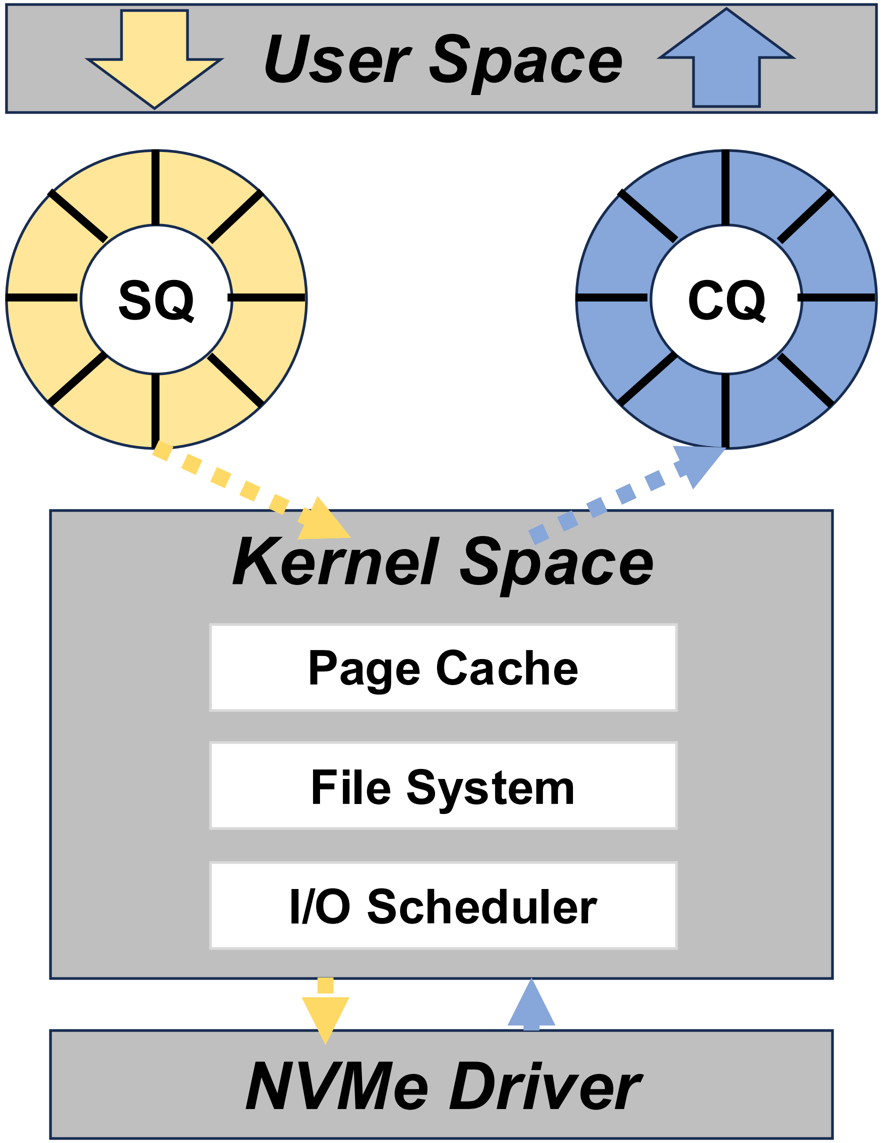


Flexible Data Placement SSD (FDP SSD)

- RU = Reclaim Unit
- PID = Placement ID

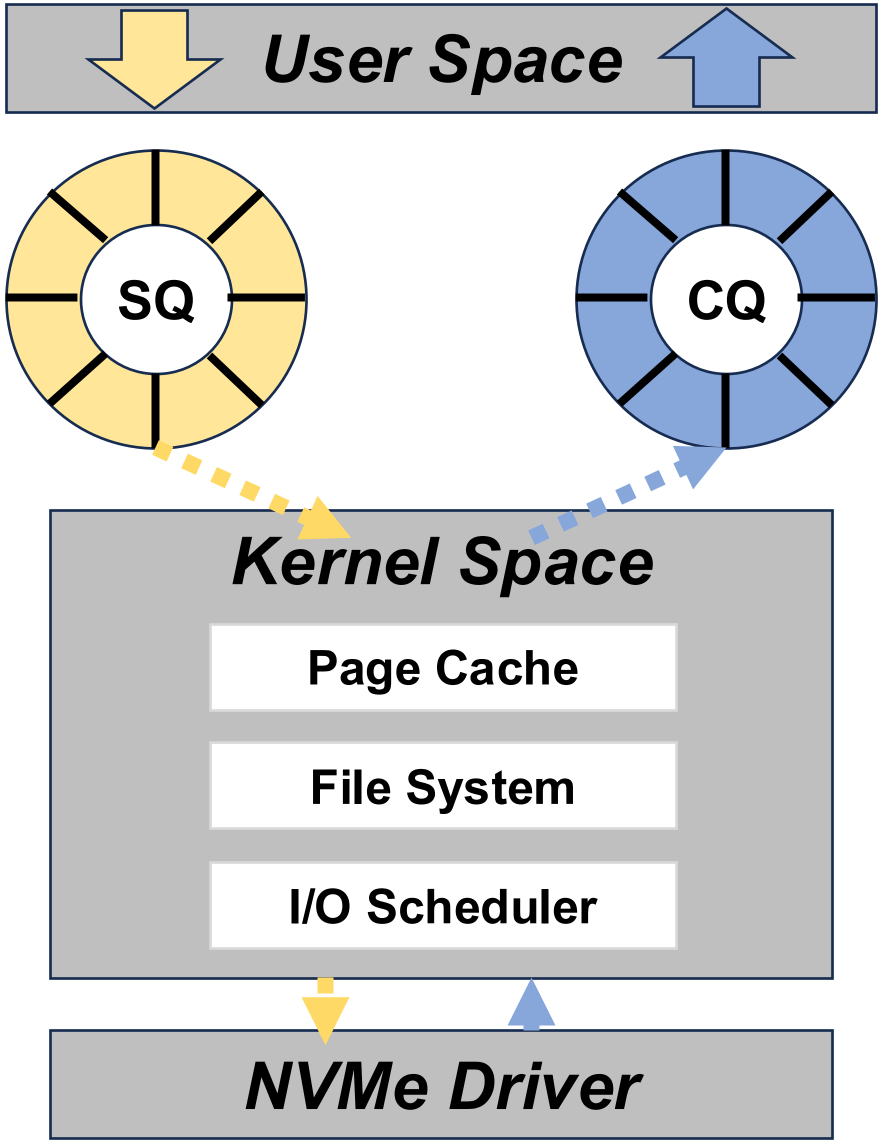


io_uring



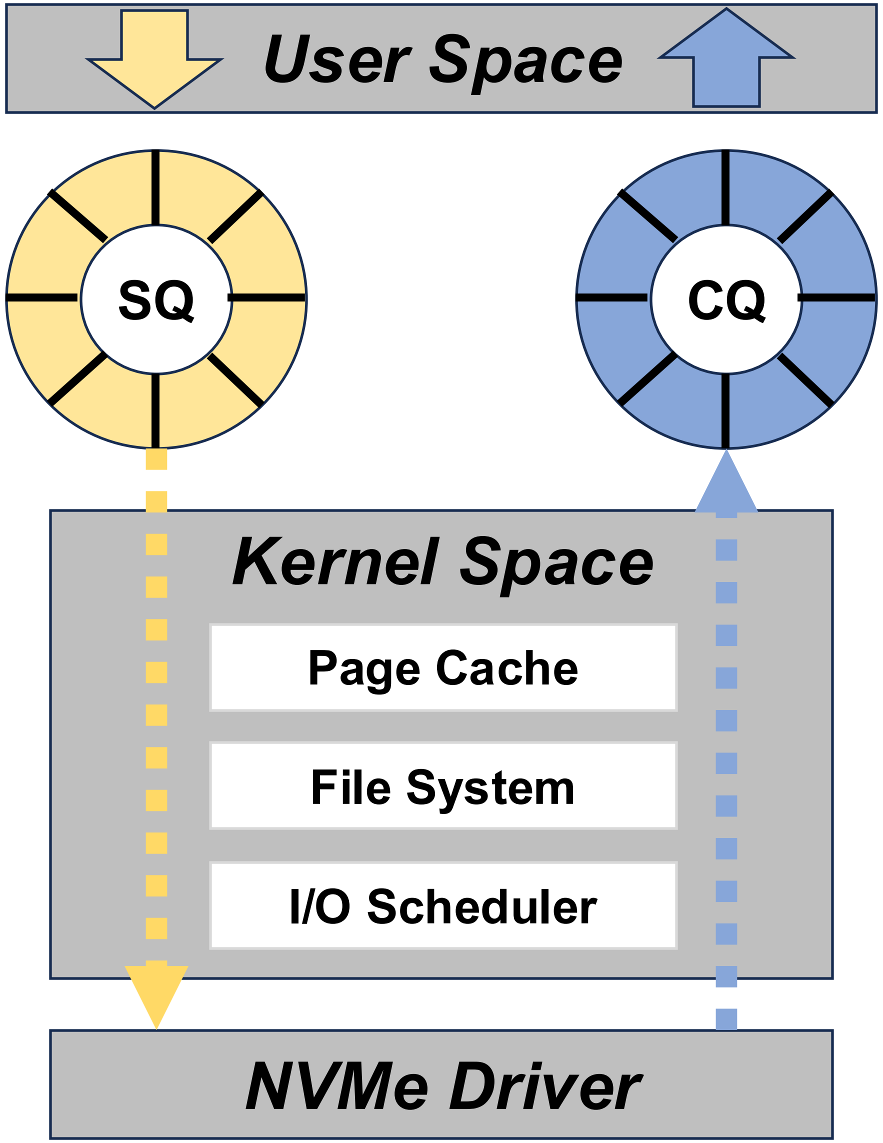
- io_uring: Async I/O API (Linux 5.1~)
- Uses two ring buffers: Submission Queue (SQ) & Completion Queue (CQ)
- SQ: user's I/O requests → CQ: completion results
- SQ & CQ shared by user and kernel
- SQ & CQ shared by user and kernel → reduces request and response structure (i.e., SQE, CQE) copying overhead.

io_uring



- Can **batch** serveral requests in one go:
  - Multiple I/O requests are queued as Submission Queue Entries (SQEs).
  - A single *io_uring_enter()* system call submits them all at once.
- **SQPoll:** Syscall-free submissions. The application can offload the submission of I/O to a kernel thread that io_uring creates.

I/O Passthru



- I/O Passthru
  (Joshi et al., FAST '24, Efficient Linux I/O path supporting advanced NVMe commands)

- Runs based on the io_uring API

- Bypasses kernel layers
  (page cache, file system, scheduler)

- Supports advanced NVMe commands
  → enables use of FDP SSD