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

Yongseok Son
Systems and Storage Lab.
Department of Computer Science and Engineering
Chung-Ang University

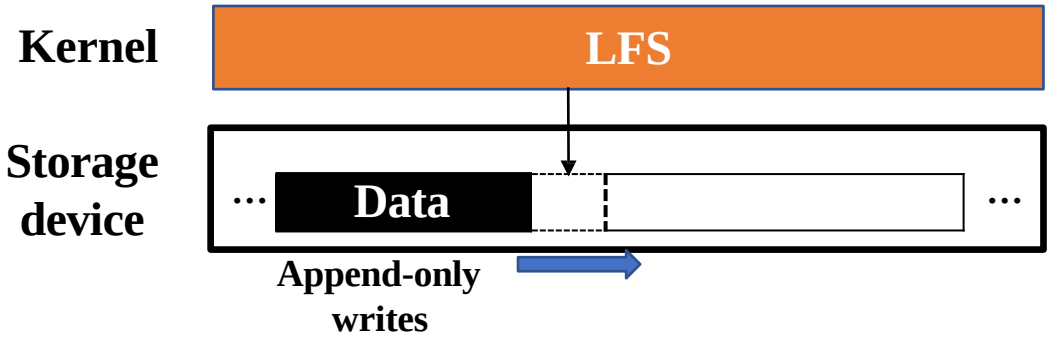
Background and Motivation

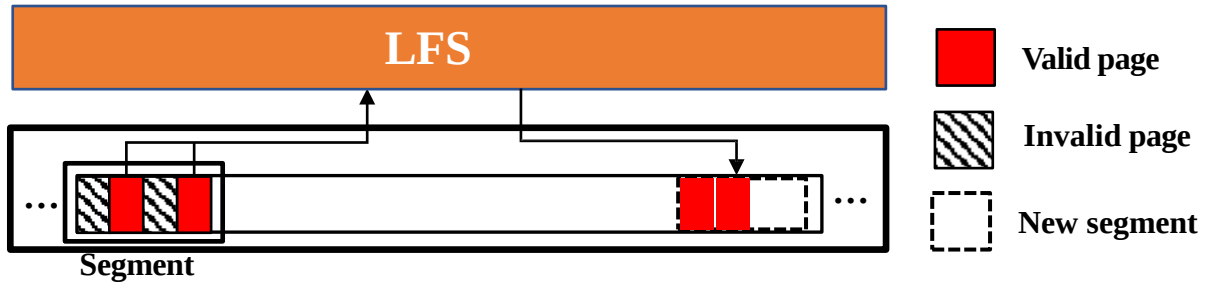
❖ Log-structured File System (LFS)

- Gathers small and random writes to large and sequential writes
- Append-only manner
  - Data are placed to consecutive logical block address (LBA)
  - As LFS ages, it accumulates invalid pages due to overwrites

❖ Garbage collection (GC)

- Reclaiming spaces occupied by invalid pages in a segment unit
- Migrating dispersed valid pages to a new segment





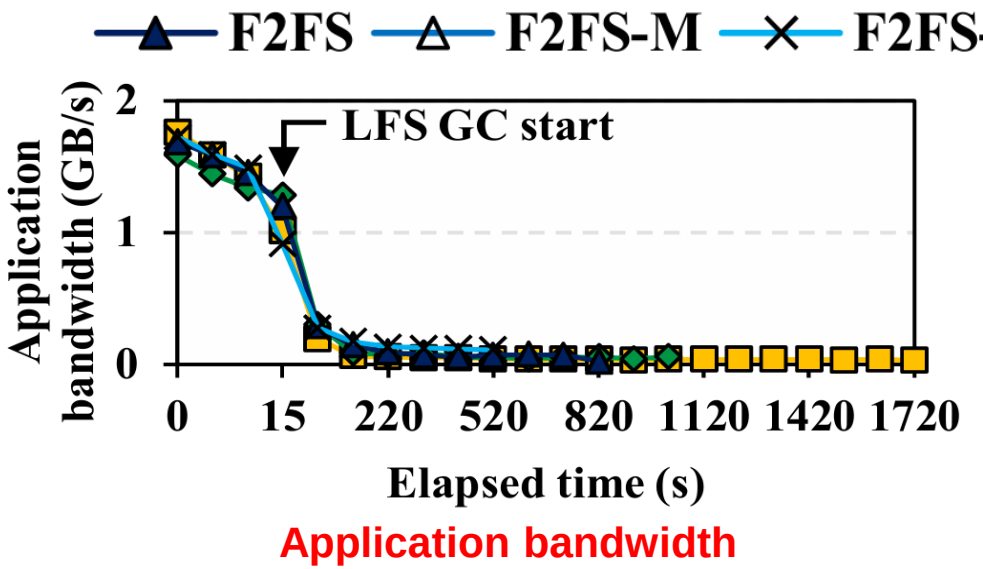
GC: Migrate valid pages and reclaim segment

Background and Motivation

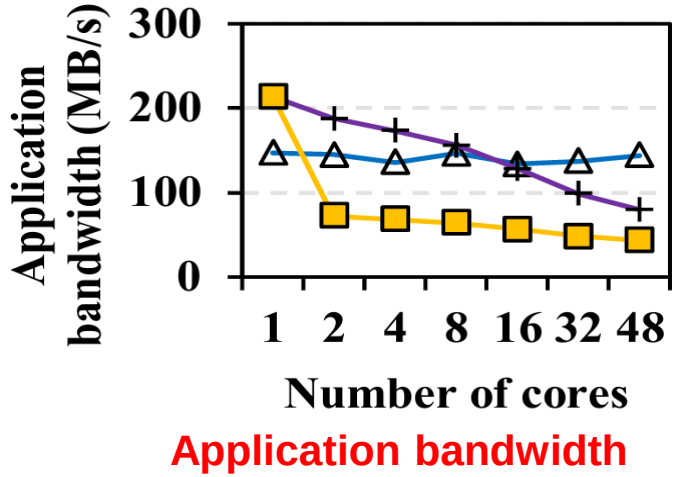
Sustained Performance of LFSs

- Overhead of GC
  - Application throughput decreases by up to 68 times
  - Serialized GC by a single GC thread
    - ✓ F2FS, F2FS-L (LFS mode), MAX (ATC'21)
- Simply increasing GC threads
  - Even performance decreases due to increased lock contention
    - ✓ F2FS-M (Modified with Multiple GC threads), F2FS-LM, P-GC (HotStorage '23)

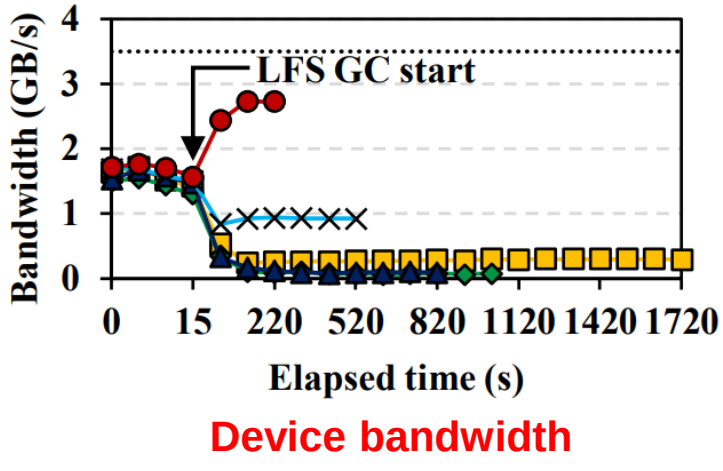
→ New Concurrent and Scalable techniques are required



Application bandwidth



Application bandwidth



Device bandwidth

Background and Motivation

- **Single GC stream and lock contention**
  - Only one cold data GC stream → limited parallel GC I/O
  - Free-space allocation serialized via curseg_mutex (cold)

- **Excessive contention on victim selection**
  - Global dirty-segment bitmap scanned with a mutex lock (seglist_lock)
  - Coarse-grained scan → under-utilization of multi-core parallelism

- **Over-strict synchronization for segment metadata**
  - Segment metadata (valid page bitmap/count) is managed via a semaphore lock (sentry_lock)
  - Frequent updates → this strict synchronization lowers concurrency

- **Coarse-grained file-level protection**
  - File-level lock (i_gc_rwsem) protects the entire file data between GC thread and I/O threads
  - Serializing the access to the victim file, limiting SSD bandwidth and core concurrency.

Overall Architecture of ScaleLFS

Dedicated garbage collector (DGC)

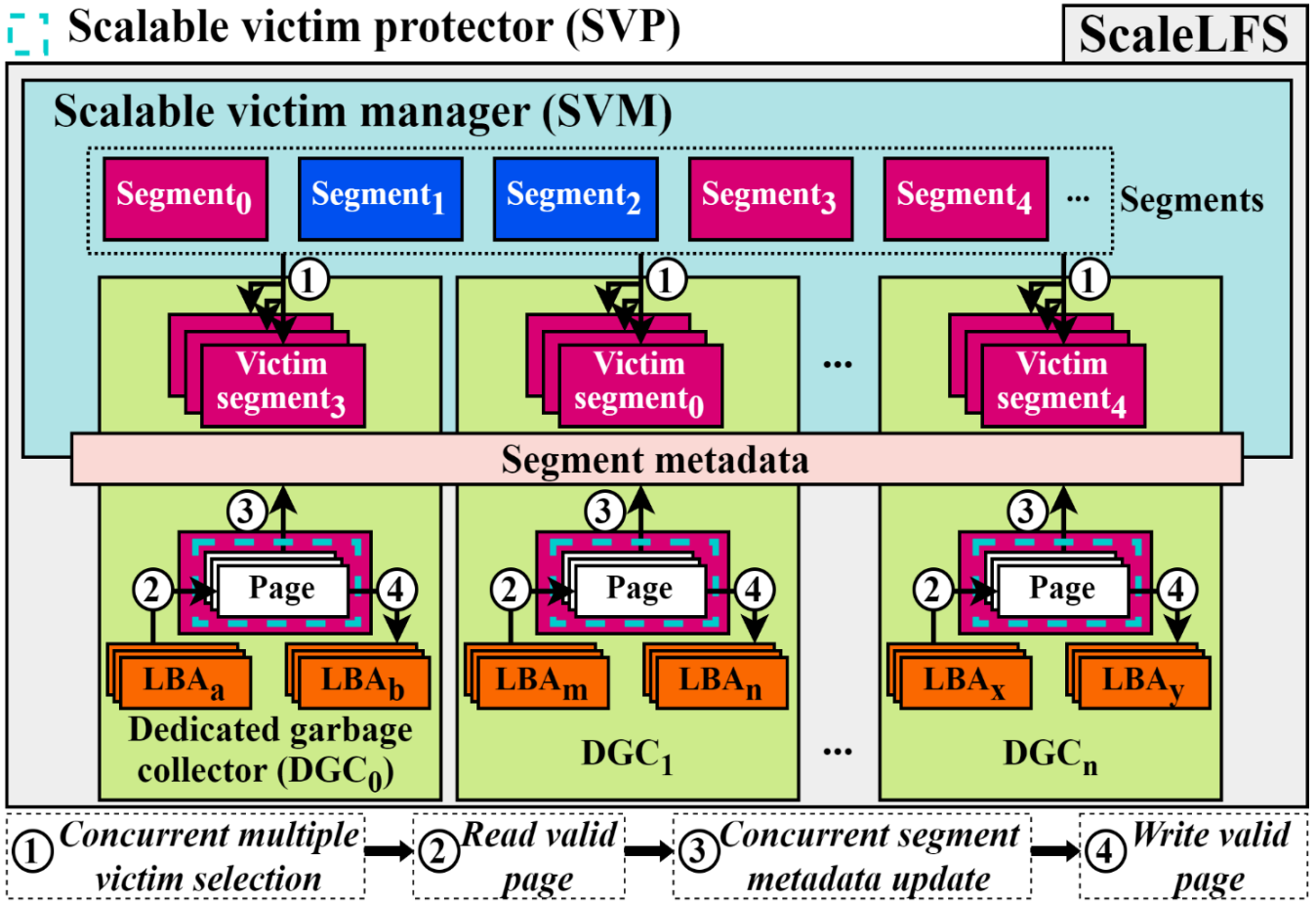
- One-to-one model
  - A per-core and resource dedication strategy
    - ✓ Dedicated page buffer
    - ✓ Dedicated segment (write stream)

Scalable Victim Manager (SVM)

- Concurrent victim selection (CVS)
  - Leveraging atomic bitmaps and instructions
- Loose-synchronization update (LSU)
  - Dividing victim segment metadata

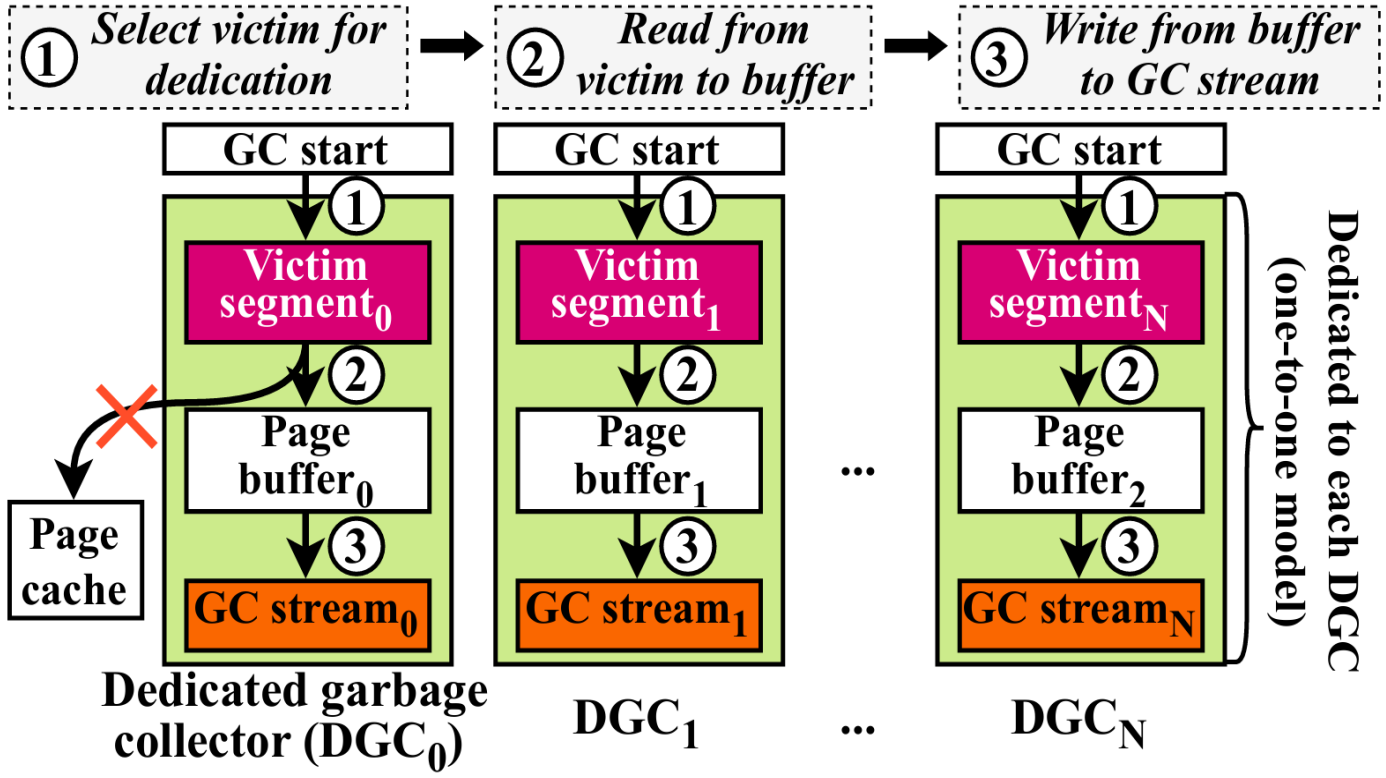
Scalable Victim Protector (SVP)

- Page-level GC management
  - Multiple GC threads cleans a file in parallel
  - Using concurrent hash table per file



Dedicated Garbage Collector (DGC)

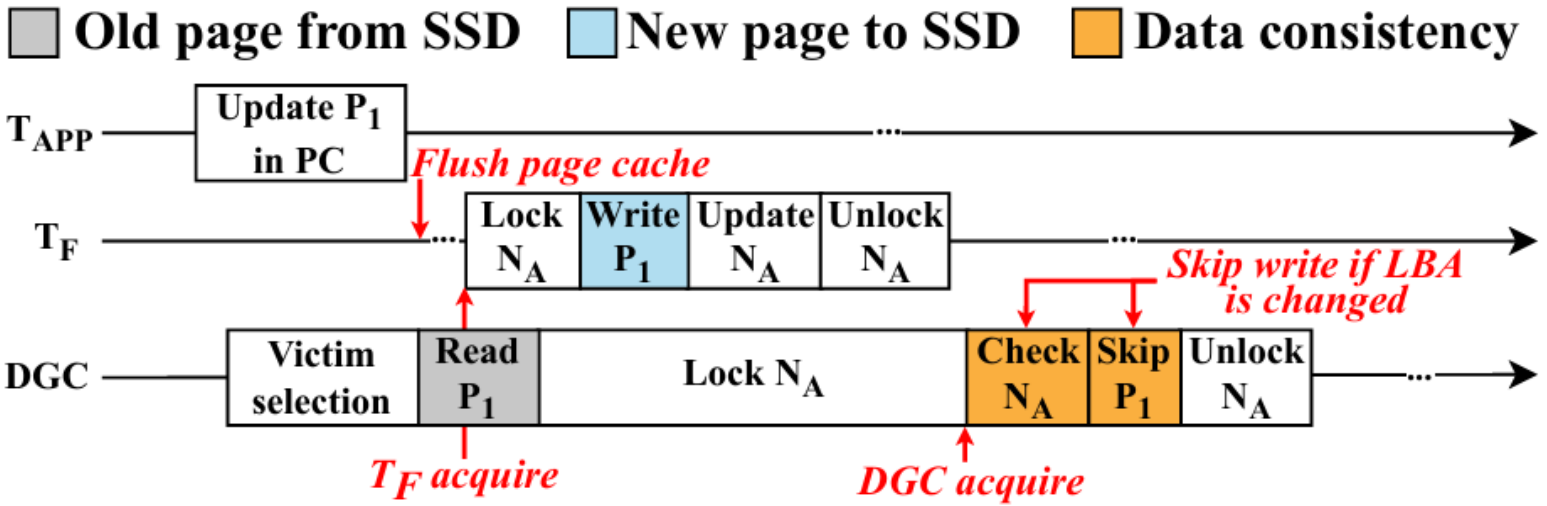
- **DGC enables one-to-one GC model with DPB/DWS**
- **Dedicated page buffer (DPB)**
  - GC data page per DGC
  - **Avoid expensive page cache overhead**
- **Dedicated write stream (DWS)**
  - Write stream per DGC
  - **Avoid lock contention from curseg_mutex (cold)**



Dedicated Garbage Collector (DGC)

❖ Data Consistency with Dedicated Page Buffer (DPB)

- Using the protection from existing node lock
- Case 1: T_F acquires the lock first
  - T_F writes **P_1 (new data)** to the device
  - Updates **N_A (new data's node)** with the **new LBA**
  - When DGC later acquires the lock:
    - ✓ Detects that LBA in N_A has changed
    - ✓ Skips writing its own P_1



T_{APP}: application thread, T_F: flush thread, P_1: page 1, N_A: node A, PC: page cache

Dedicated Garbage Collector (DGC)

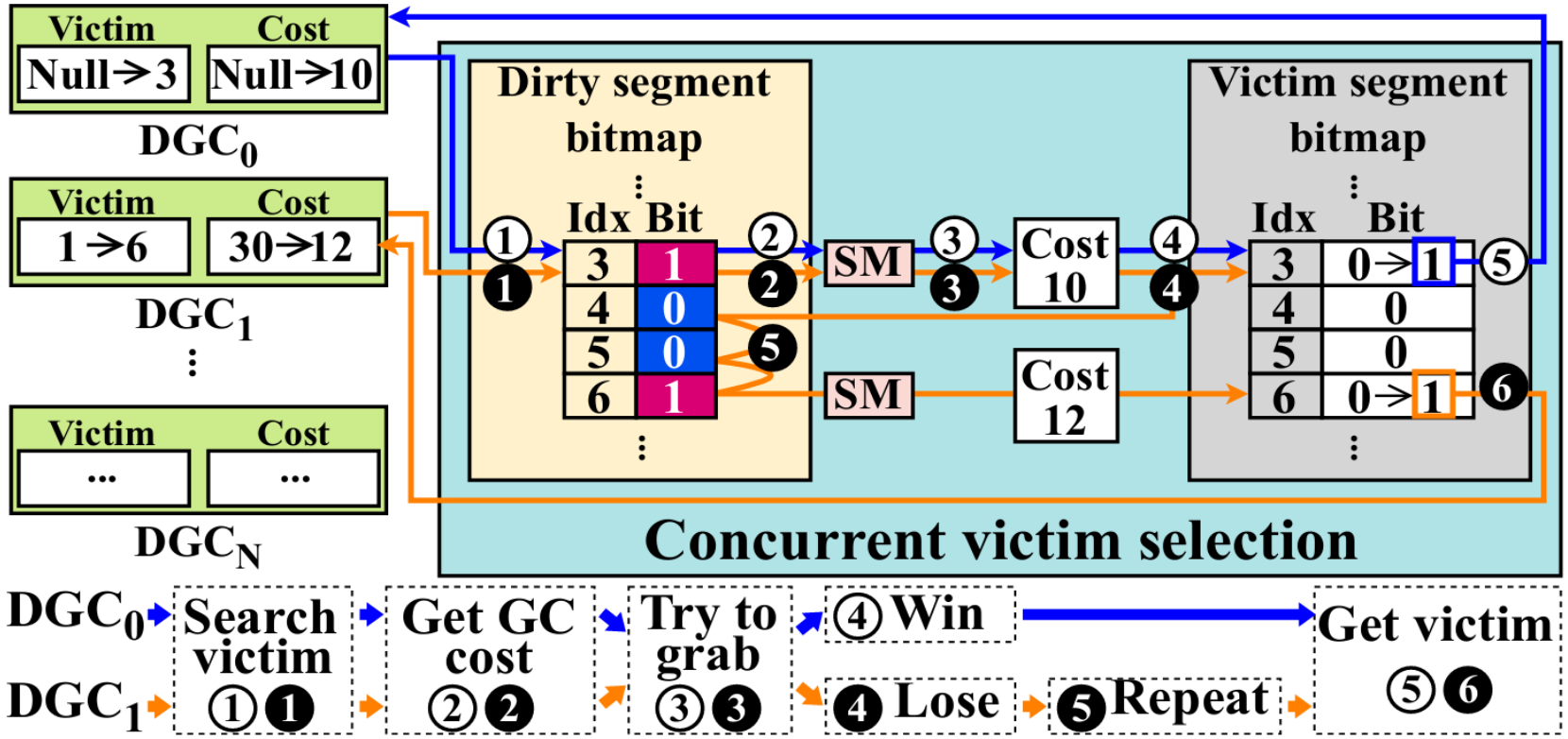
❖ Data Consistency with Dedicated Page Buffer (DPB)

- Case 2: DGC acquires the lock first
  - DGC finds **LBA in N_A** is same as the victim LBA (source address)
    - ✓ T_F has **not yet written** P1
  - DGC writes **P_1** to the device and updates **LBA** in N_A
  - Later, T_F writes the **new P1** from page cache after DGC releases the lock

Scalable Victim Manager (1)

Concurrent Victim Selection (CVS)

- Two atomic bitmaps
  - Dirty segment bitmap (GC victim candidates)
  - Victim segment bitmap (Selected GC victim)
- Concurrent victim selection procedure
  - Concurrently scan dirty segment bitmap
    - ✓ Selecting segment having least valid pages
  - Concurrently mark selected segment to the victim segment bitmap



Idx: segment number, bit 1: dirty segment, SM: segment metadata (segment entry)

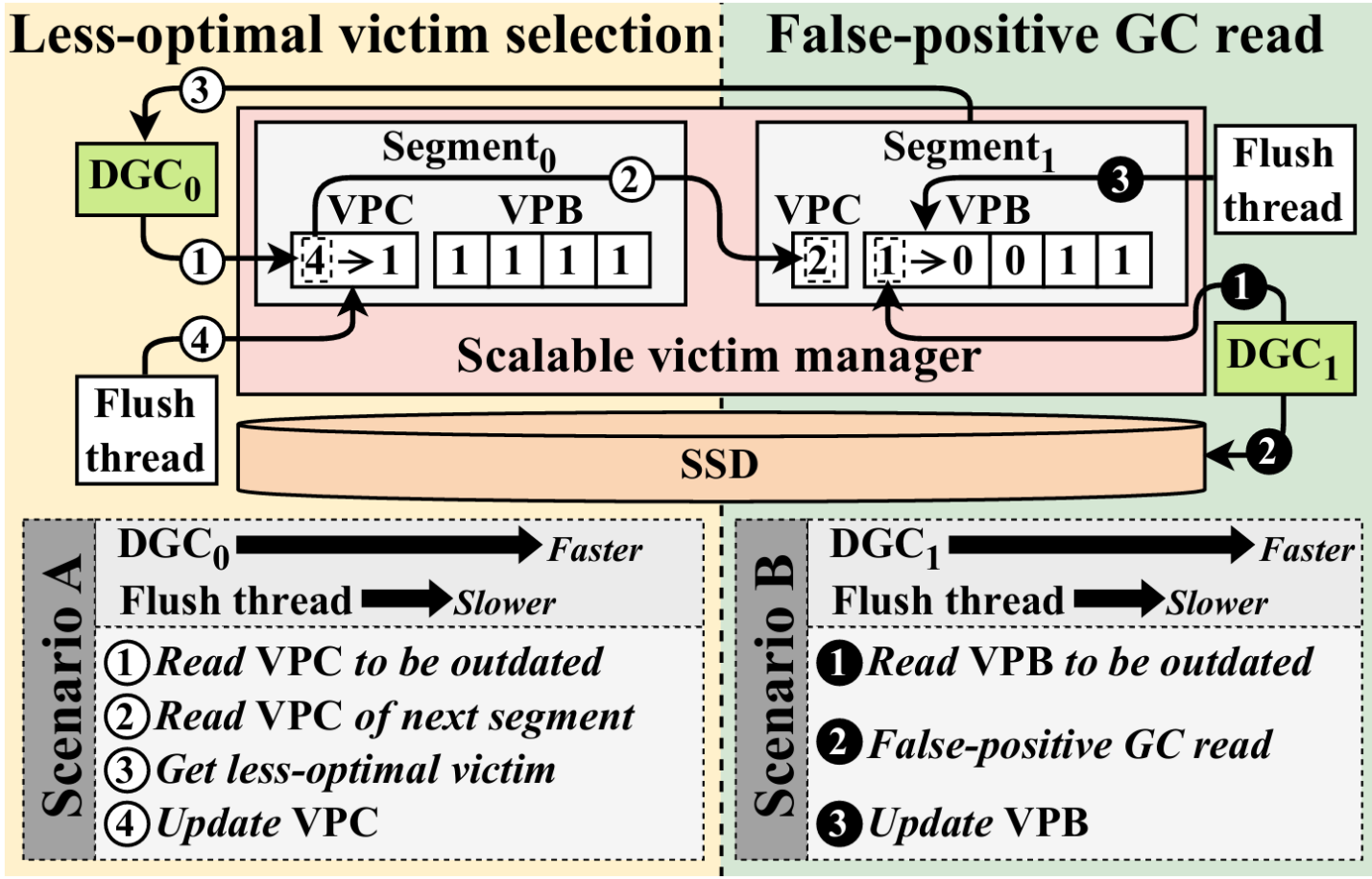
Scalable Victim Manager (2)

- **Loose-synchronization Update (LSU)**
  - Accessing and updating the metadata of victim segments when selecting victims and after copying valid pages
    - Segment entry lock (sentry_lock) incurs high contention among DGCs
  - Manage per-segment metadata
    - Atomic valid page bitmap (VPB)
    - Atomic valid page count (VPC)
  - Concurrent update and access to valid bitmap and valid count

- **Two side effects**
  - Side effect #1: Less-optimal victim segment selection
    - Selecting best victim segment is not guaranteed
    - Write amplification issue
  - Side effect #2: False-positive GC read
    - Unnecessary GC read can be submitted
    - Data consistency issue

Scalable Victim Manager (2)

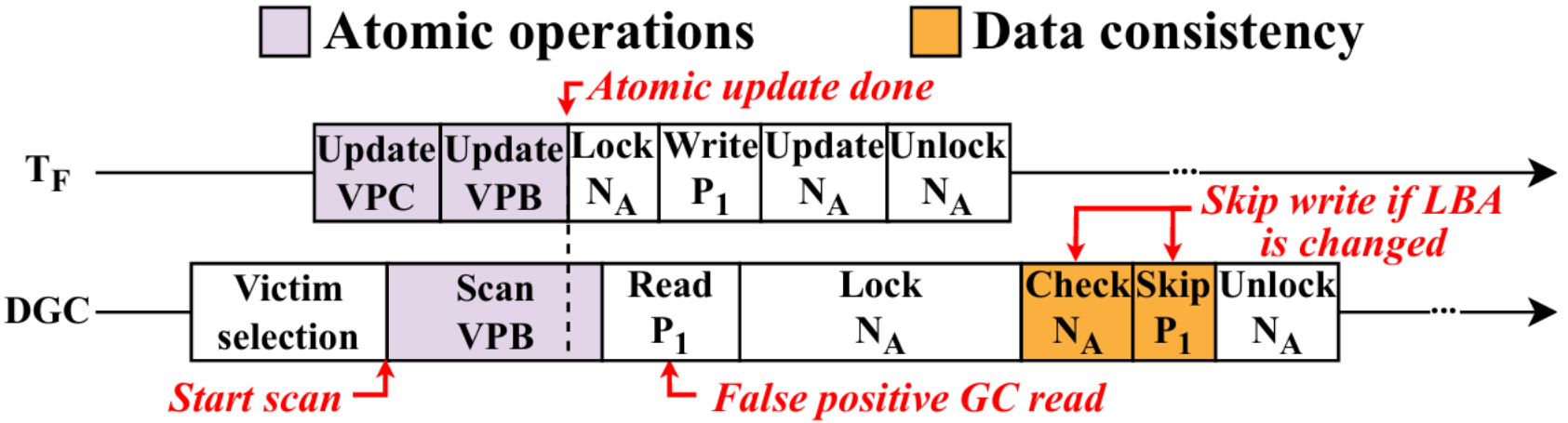
- **Loose-synchronization Update (LSU)**
  - Side effect #1: Less-optimal victim segment selection
  - Side effect #2: False-positive GC read



Scalable Victim Manager (2)

❖ False-positive GC read

- Keeping data consistency using the protection from existing node lock
  - T_F writes **P_1 (new data)** to the device
  - Updates **N_A (new data's node)** with the **new LBA**
  - When DGC later acquires the node lock:
    - ✓ Detects that LBA in N_A has changed
    - ✓ Skips writing its own P_1



T_F: flush thread, P_1: page 1, N_A: node A, VPC: valid page count, VPB: valid page bitmap

Scalable Victim Protector (SVP)

❖ Existing LFSs perform GC at file granularity

- Application threads (e.g., **direct I/O, TRIM,** etc) can directly access or update **victim pages** of a file under GC.
- To prevent conflicts, LFSs employ a file-level GC lock (i_gc_rwsem) to serialize access to the victim file.
- Serializing the access to the victim file and limiting concurrency

❖ SVP enables a page-level GC procedure

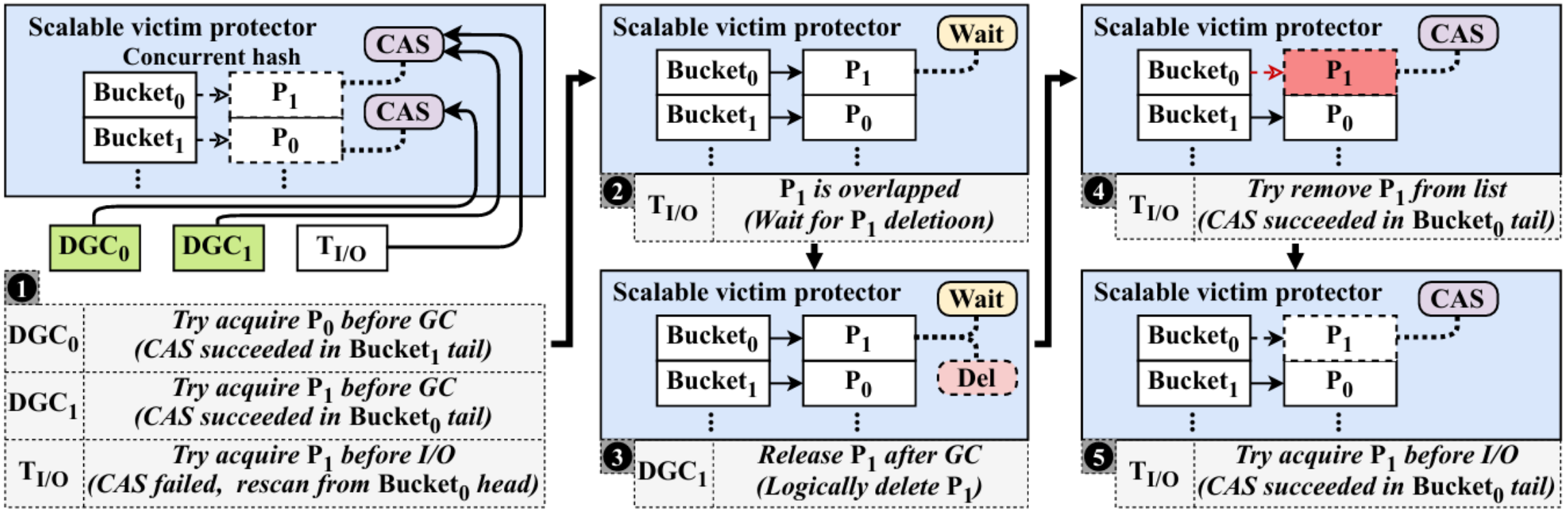
- Protecting individual victim pages instead of a file in a scalable manner
- Using a concurrent protection technique [1] based on a concurrent hash table
  - It allows threads to simultaneously access or update the different pages in a file

[1] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20,

Scalable Victim Protector (SVP)

❖ Per-file concurrent hash table

- Concurrently insert/remove page before/after access
- Page index-based hash bucket selection to reduce contention



Evaluation

Experimental Setup

OS	Ubuntu 22.04 LTS
kernel	Linux 6.0.0
CPU	Intel Xeon E5-2650 CPU (2.2 GHz, 24 physical cores and up to 48 logical cores)
Memory	160 GB of DRAM
SSD	Samsung 9A3 SSD (7.68 TB)

Workloads

- Micro-benchmark: FIO benchmark
- Macro-benchmarks
  - Filebench workloads: Fileserver, Varmail, and OLTP
- Real-world Application: MySQL with YCSB (A, B)

Evaluation

GC Experiment Setup

- Partition
  - 8 GB of DRAM and 30GB partition
  - Understanding impact of filesystem-level GC without device-level GC
- Full SSD capacity
  - 160 GB of DRAM and 7.68TB capacity
  - This shows how much ScaleLFS is effective under device-level GC

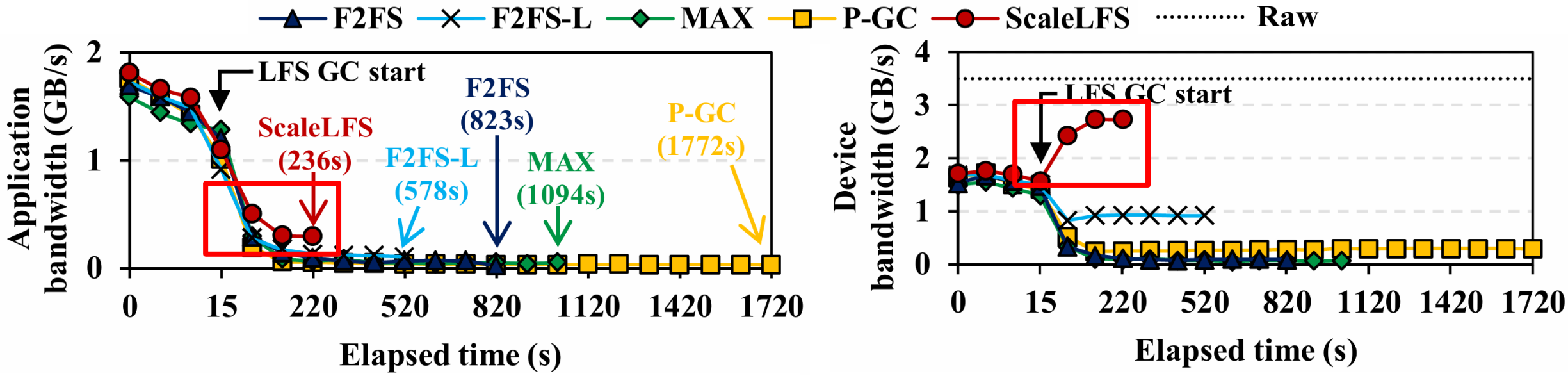
Comparison

- F2FS
- F2FS-L: F2FS in LFS mode
- MAX: Scalable LFS (USENIX ATC'21)
- P-GC: Parallel GC (HotStorage'23)

Evaluation: Micro-benchmark (Partition)

Timeline of Application and Device Bandwidth

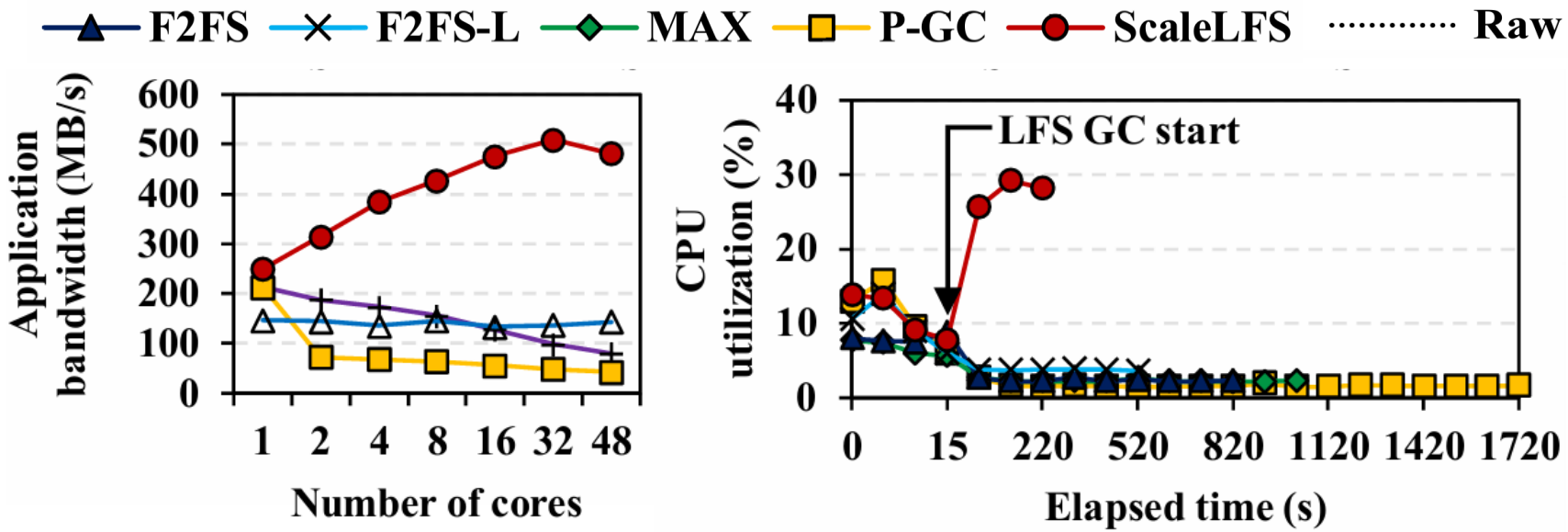
- **Device-level GC and pure file system GC performance**
- Application-level improvement compared with F2FS, F2FS-L, MAX, and P-GC
  - **Bandwidth improvement**: **3.5x**, **2.4x**, **4.6x**, and **7.0x**
  - **Execution time reduction**: 71.1%, 58.1%, 78.1%, and 85.8%
- Maximizing utilization of device-level bandwidth
  - ScaleLFS utilizes more device-level bandwidth by up to **19.6x**
  - ScaleLFS achieves **2.9GB/s**, reaching **near raw device bandwidth limit** (**3.4GB**)



Evaluation: Micro-benchmark (Partition)

Core Scalability and CPU utilization

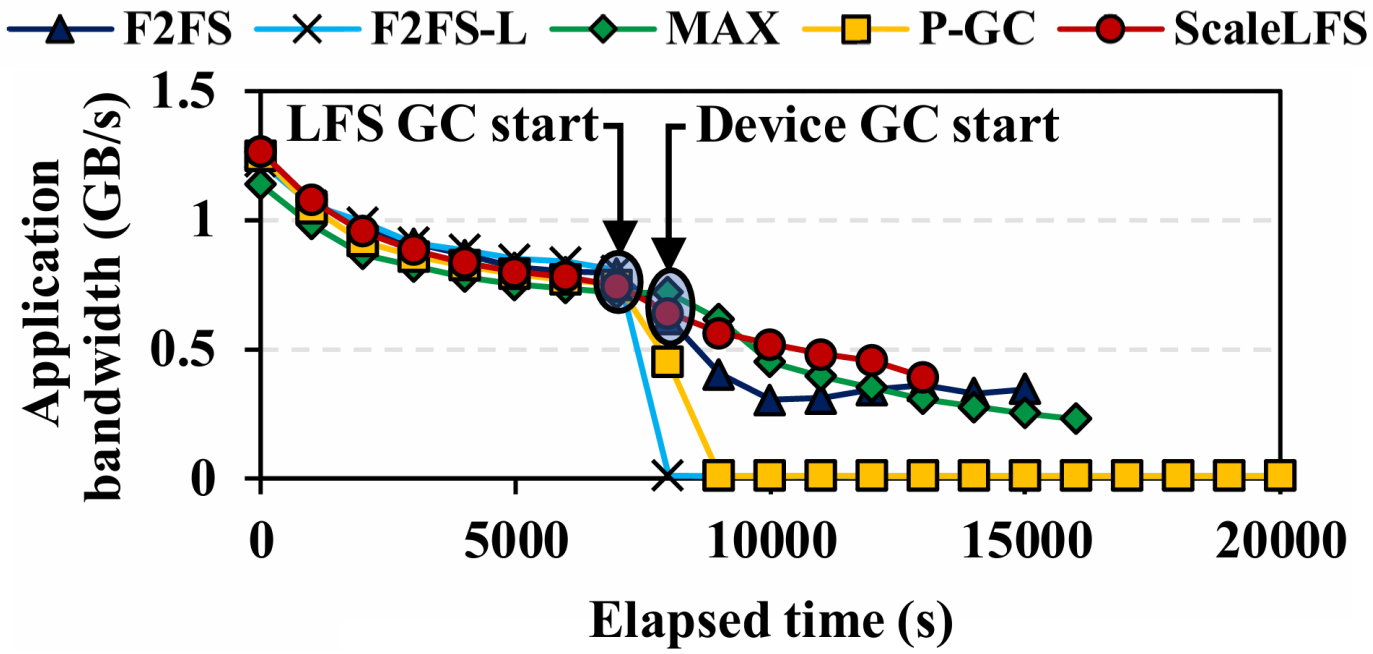
- Core scalability
  - ScaleLFS scales the performance by up to 3.4x, 3.7x, 3.9x, 4.4x, 5.2x, and 5.8x, in the 2, 4, 8, 16, 32, and 48 cores
- CPU utilization
  - F2FS, F2FS-L, and MAX use one core by serialized GC
  - P-GC's GC threads are blocked frequently due to the file-level locking
  - CPU utilization of ScaleLFS increases by up to 29.1% when GC is triggered



Evaluation: Micro-benchmark (Full Capacity)

❖ Execution time

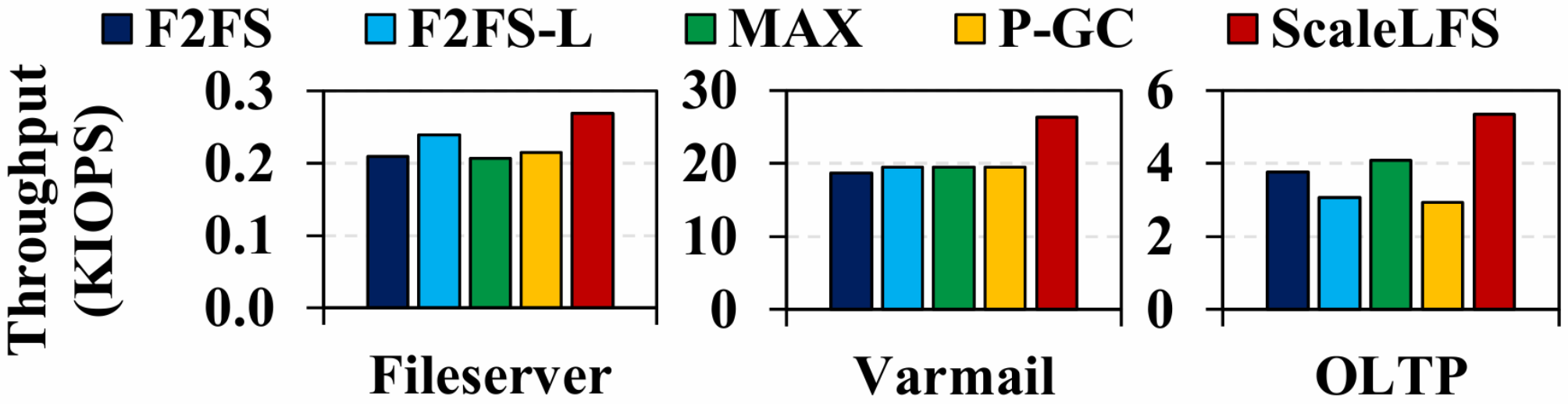
- ScaleLFS completes the I/O operations about 37 minutes faster than F2FS and 56 minutes faster than MAX
- F2FS-L, P-GC
  - They cannot terminate the I/O operations even when their execution time exceeds 20,000 seconds (11 MB/s).



Evaluation: Macro-benchmark

Filebench

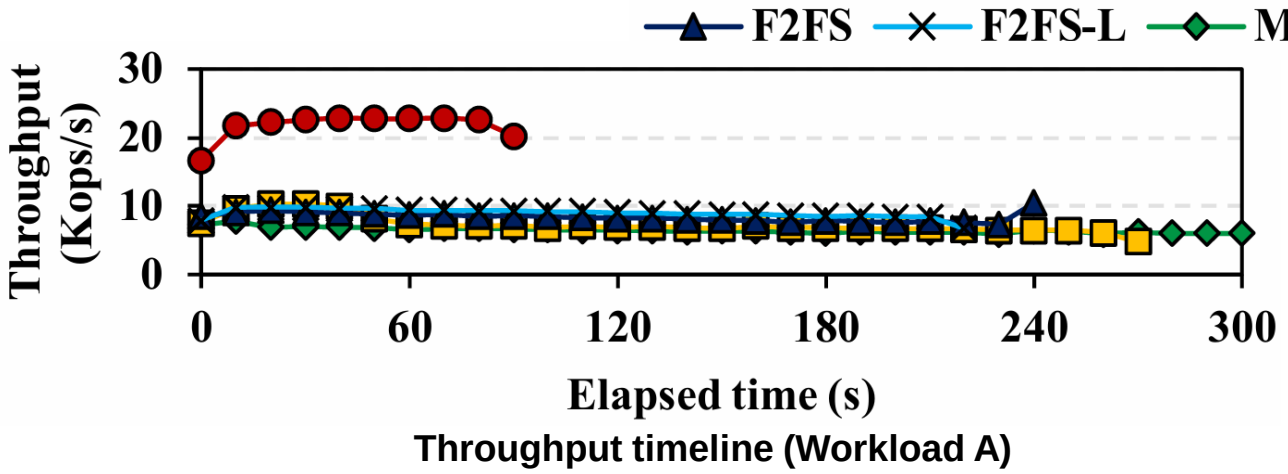
- Improvement: 30.3% (fileserver), 40.6% (Varmail), 83.1% (OLTP)
- Less improvement than FIO
  - Fileserver/Varmail: frequent deletion → infrequent GC
  - OLTP: reader and writer threads block each other → less frequent write and GC



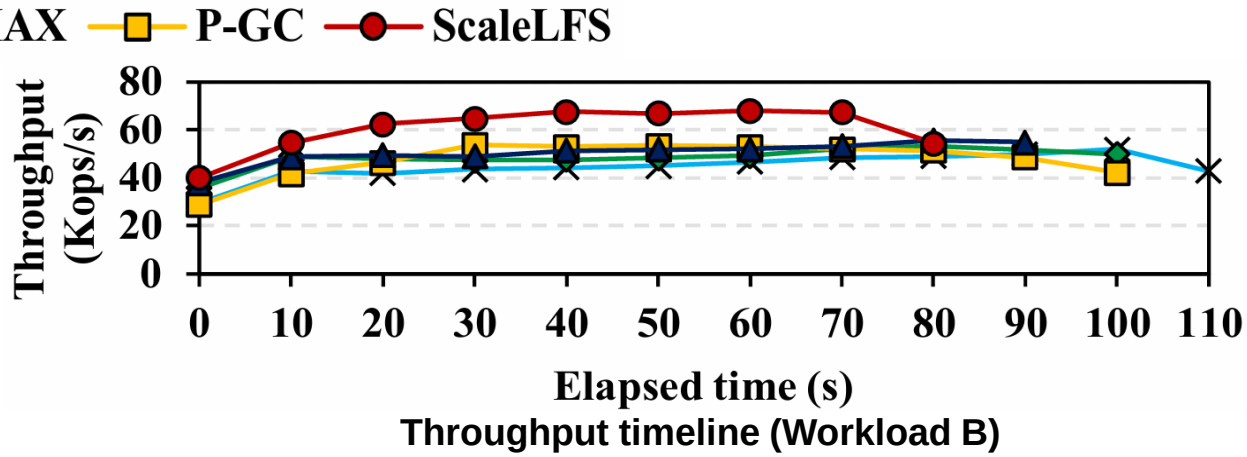
Evaluation: Real-world Application

MySQL with YCSB benchmark

- Workload A (Write-intensive)
  - Update 50%, Read 50%
  - Throughput: 3.38x, Execution time: 70.4% reduction
- Workload B (Read-intensive)
  - Update 5%, Read 95%
  - Throughput: 1.37x, Execution time: 27.3% reduction



Throughput timeline (Workload A)



Throughput timeline (Workload B)

Evaluation

❖ Performance Breakdown and Lock Overhead

- **SPGC**: Simple per-core garbage collector
- **SVP**: Scalable victim protector (target lock: i_gc_rwsem)
- **DWS**: Dedicated write stream (target lock: curseg_mutex)
- **LSU**: Loose-synchronization update (target lock: sentry_lock)
- **CVS**: Concurrent victim selection (target lock: seglist_lock)
- **DPB**: Dedicated page buffer

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

Side Effect of ScaleLFS

❖ Two side effects observed in ScaleLFS

- Less-optimal GC victim selection
- False-positive GC reads

❖ Less-optimal Victim Selection

- Loose synchronization slightly increases WAF but improves scalability of GC.
- Write Amplification Factor (WAF)
  - **Fileserver:** 1.0036 in F2FS → 1.0040 in ScaleLFS (**+0.3%**)
  - **YCSB:** 9.05 in F2FS → 9.37 in ScaleLFS (**+3.6%**)

❖ False-positive GC Reads

- Frequency is low and performance impact is negligible.
  - **Fileserver:** None
  - **YCSB:** 5,890 false-positive reads / 1M I/Os → **0.006 per I/O request**

Conclusion

- **ScaleLFS**
  - Parallelizing the GC procedure with a per-core and dedicated resource-based mechanism
  - Concurrent selecting victim segments and updating the metadata of the segments via a concurrent data structure and loose synchronization updates
  - Enabling page-level GC and protection with a scalable data structure

- **Please refer to the paper for further details**
  - https://dl.acm.org/doi/10.5555/3724648.3724652

- **ScaleLFS** is open source now:
  - https://github.com/syslab-cau/ScaleLFS

Q&A

Thank you for your attention