Overlapping Aware Zone Allocation for LSM Tree-Based Store on ZNS SSDs

Jingcheng Shen[1], Lang Yang[1], Linbo Long[1], Renping Liu[1], Zhenhua Tan[1], Congming Gao[2] , Yi Jiang[1]

[1]Chongqing University of Posts and Telecommunications, China

[2]Xiamen University, China

Outline

- Background
  - ➢ZNS Interface
  - ➢Work Related to LSM Tree-Based Store on ZNS SSDs
- Motivation & Proposed Method
- Evaluation
- Conclusion

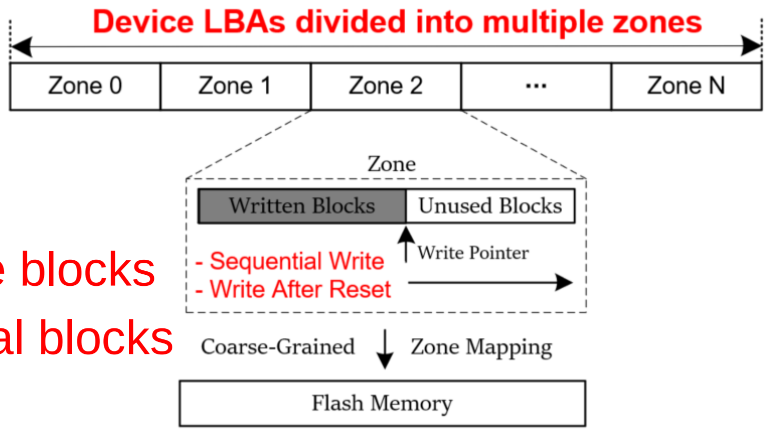
ZNS Interface (1/2)

- NVMe Zoned Namespaces (ZNS) Interface
  - ➢Divides SSDs’ storage space is into logical zones
  - ➢Efficiently manages data storage on flash SSDs

- Motivation of ZNS Interface
  - ➢Conventional block interface writes data in 4/8-KB blocks
  - ➢Flash erase block sizes are much larger (e.g., >1 MB)
    - ➢This discrepancy causes space fragmentation
      - →More garbage collection (GC) operations

ZNS Interface (2/2)

- Properties
  - Fixed-size zones
  - Sequential write & random read
  - Zone mapping
    - A ZNS zone is mapped to multiple erase blocks
    - An erase block contains multiple physical blocks



- Common scenarios
  - Log-structured merge-tree (LSM tree) based key-value stores
    - LevelDB, RocksDB, etc.
  - Log-structured file systems
    - F2FS, etc.

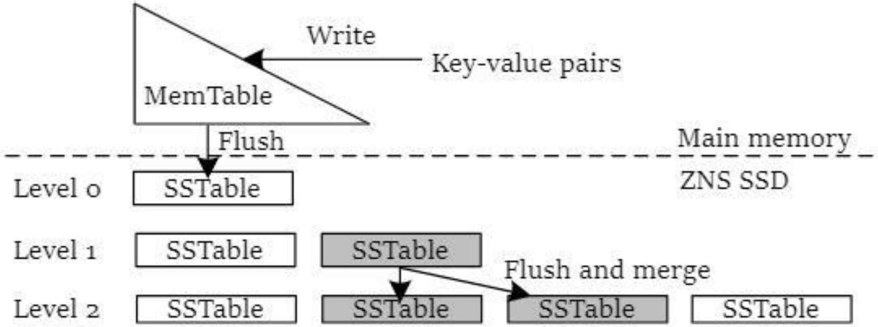
Comparing ZNS Interface to Block Counterpart

- Lowers DRAM space requirement
  - Larger granularity, smaller mapping table
- Eliminates GC resp. from device side
- Achieves 1.5x and 1.2x as fast on average in terms of write and read, respectively [1]
- For ZNS practitioners, it is crucial to design and implement host-side data-management algorithms

[1] Matias Bjørling , et al. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21), 2021

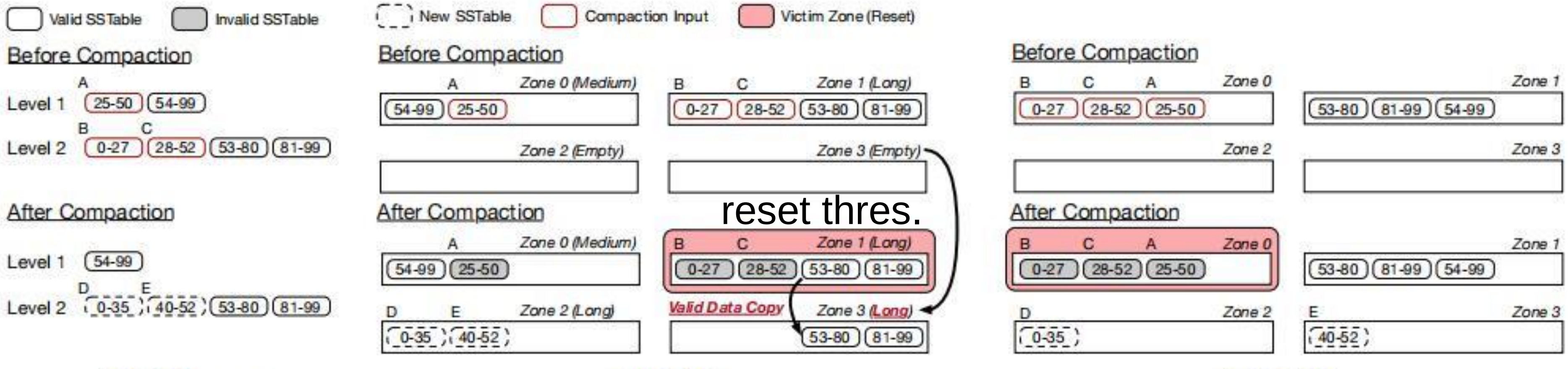
Log-Structures Merge Tree

- Most ZNS-relevant scenarios
  - Multiple levels and each level is double in size compared to its upper level
  - New data is first written to a log-like structure (Memtable)
  - Data is periodically merged/compacted into larger, sorted files (SSTables)
    - An SSTable is merged with files at lower tree level that hold keys overlapped with it
    - After merge/compaction, one or more new SSTables are created and the old SSTables are invalidated



Previous Work

- Conventional lifetime-based zone-allocation method (LIZA)
- Compact-aware zone-allocation method (CAZA) [2]



(a) LSM-tree (b) LIZA (c) CAZA

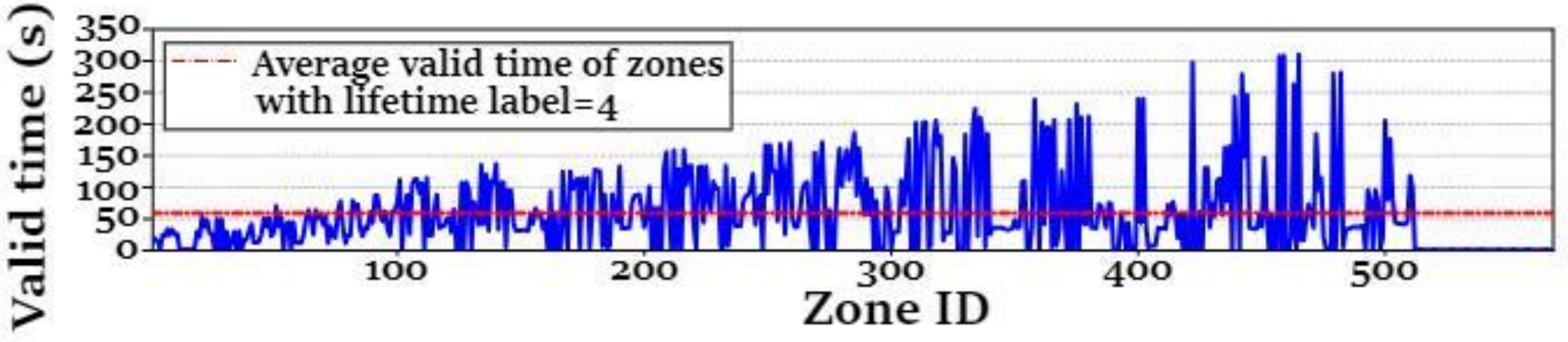
[2] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, "Compaction-aware Zone Allocation for LSM based Key-Value Store on ZNS SSDs," in Proc. of HotStorage'22, pp. 93–99, 2022.

Limits of Previous Work

- LIZA assigns SSTables at same tree levels to same zones, ignoring the fact that data lifetime within the same level notably varies
- CAZA assigns SSTables with overlapped key ranges to same zones, without considering the tree levels where these data is associated

Motivation

- Each tree level is associated with a lifetime label
- Such lifetime estimation is inaccurate, because SSTables at the same tree level may drastically vary in lifetime



Drastically varying valid times across zones with lifetime=4 (with a mean of 59.3 s and a standard deviation of 62.7 s)

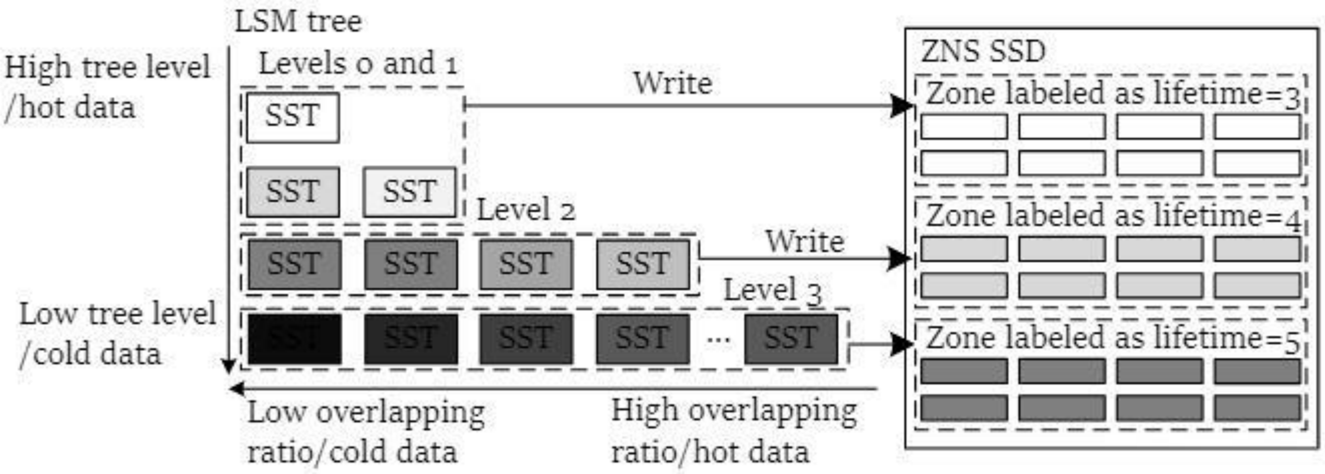
We must avoid using inaccurate lifetime estimation solely based on the LSM-tree levels

Design of OAZA

- **Vertical and Horizontal Lifetime Estimations**
  - Lifetime of each SSTable is estimated based on both the vertical and the horizontal factors
- **Overlapping Aware Zone Allocation**
  - After merge/compaction, if an SSTable at tree level *L* is created, the file is finally assigned to an appropriate zone by considering the overlapping ratio of the key range with data files at the lower level (*L+1*)

Lifetime Estimation

- Vertical Lifetime Estimation (based on tree level) prioritizes selecting from the zones containing level-*L* files as the destination to place the new file



- Horizontal Lifetime Estimation considers intra-level relative data hotness as the complementary factor

Overlapping Ratio

- Overlapping ratio indicates relative data hotness
  - Compares an SSTable to other SSTables within the same tree level
  - Each SSTable within a tree level have an overlapping ratio that shows how much the SSTable is overlapped with SSTables at the neighbor level in terms of key ranges
  - SSTables with similar overlapping ratios are preferred to be written to the same zone

Calculation of Overlapping Ratio

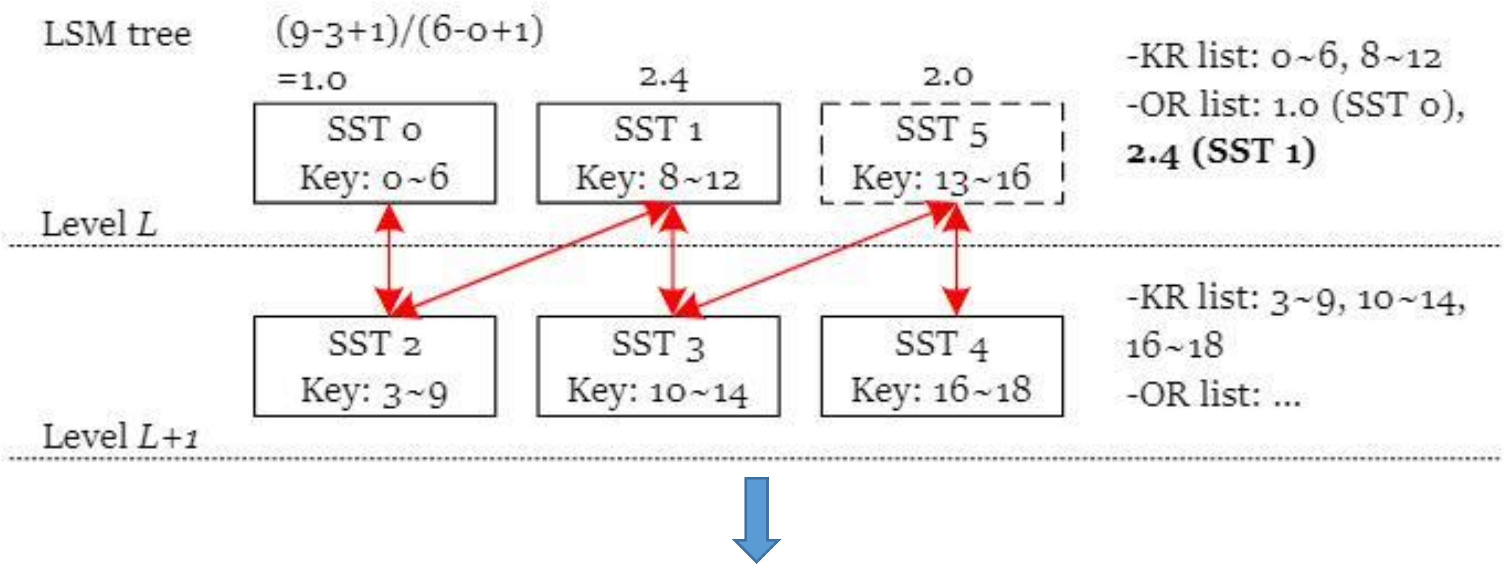
- Assume an SSTable S_{new} is created that is associated with tree level *L*
  - ➢ K_{new}: Keys of S_{new}
- SS_{L+1} is the set of all SSTables associated with level *L+1*
- O_{new}: Overlapping ratio of S_{new}
  - ➢ $O_{new} = \frac{\sum |Ki|}{|K_{new}|}$, where K_i is the range of keys of any $S_i \in SS_{L+1}$ if S_i share keys with S_{new}

Overlapping Aware Zone Allocation

- Two sorted lists implemented for each tree level
  - Overlapping-ratio list (OR) manages the overlapping ratio of every SSTable at the tree level
    - → SSTables are sorted by descending ratios
  - Key-range list (KR) manages the key range of every SSTable at the tree level
- When a new SSTable (S_{new}) is created at tree level *L*
  - The overlapping ratio of S_{new} is calculated by comparing the its key range with the KR of *L+1* (*L-1* if *L* is the bottom level of LSM tree)
  - The overlapping ratio of S_{new} is compared with the OR of *L* to find an SSTable S_{target} with the closest overlapping ratio
  - S_{new} is written to the zone where S_{target} resides

Illustrative Example

- Zone Allocation searches in the overlapping-ratio list of *L* to identify an S_{target} with the overlapping ratio closest to that of SST 5



OAZA writes SST 5 to the zone that contains SST 1.

Overhead

- OAZA only introduces a small time overhead
  - the sorted lists are implemented with the C++ standard library (stl::set)
  - For an LSM tree with *N* SSTables, looking and inserting into the two lists are of an O(log *N*) time complexity

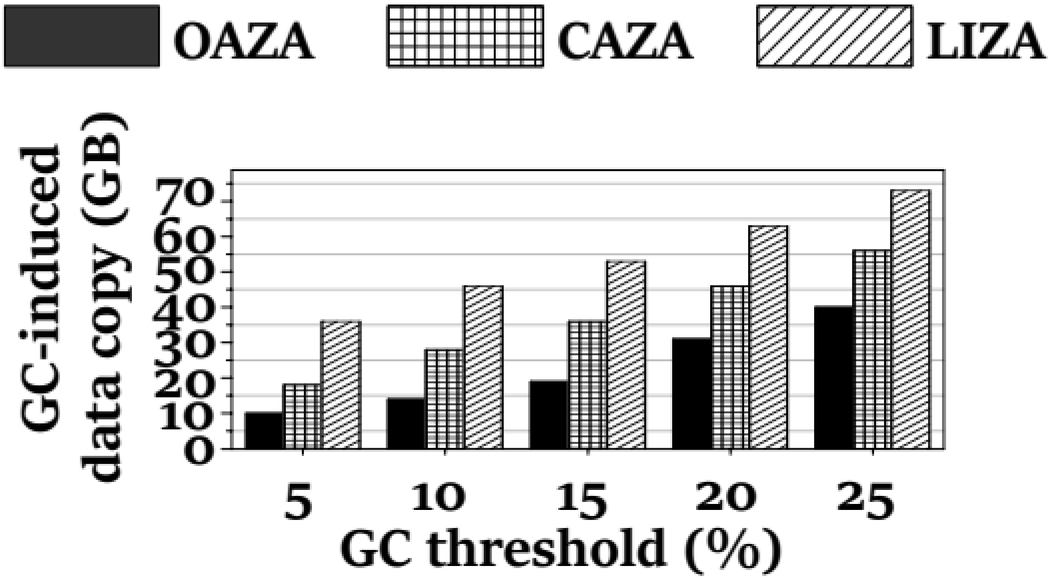
Experiments Setup

- Detailed Configurations

FEMU	FEMU version: 7.0.0, Linux kernel: 5.13
ZNS	Zone size: 128 MB; No. of Zones: 256; Zone Parallelism: 32
Simulated SSD	Channels: 8, Chips/Channel: 2, Dies/Chip: 2, Planes/Die: 4, Blocks/Plane: 256, Pages/Block: 256, Page capacity: 4 KB
db_bench (RocksDB)	Key Size: 16 B; Value Size: 8 KB; Max SST File Size: 32 MB; I/O Mode: Direct I/O
Workloads	Random writes (6 million KV pairs)

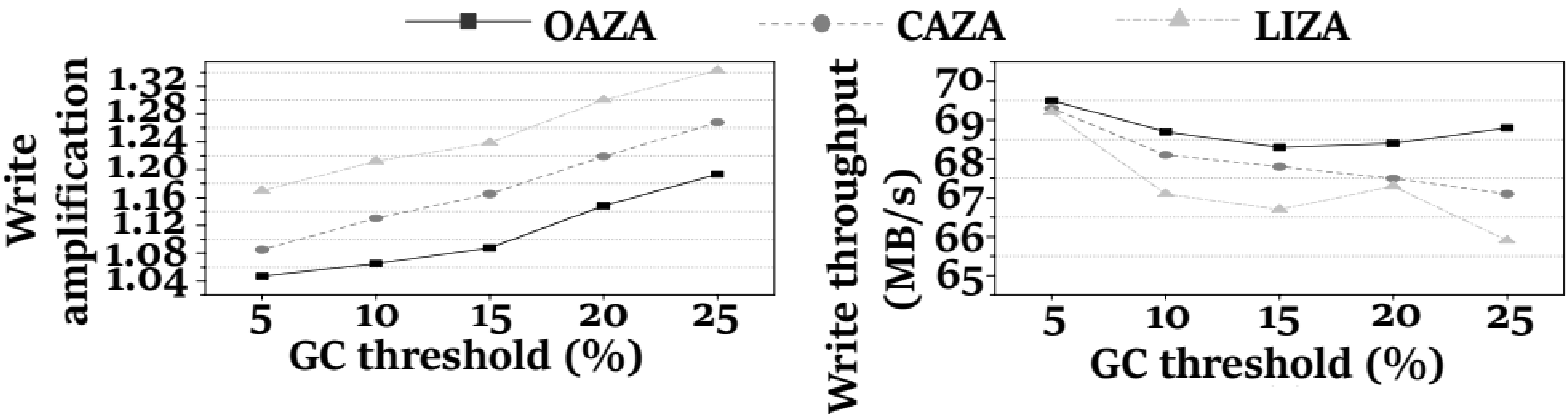
- OAZA is compared to
  - ➢ LIZA
  - ➢ CAZA
- GC Threshold
  - ➢ real-time space utilization

Performance: GC-Induced Data Copy



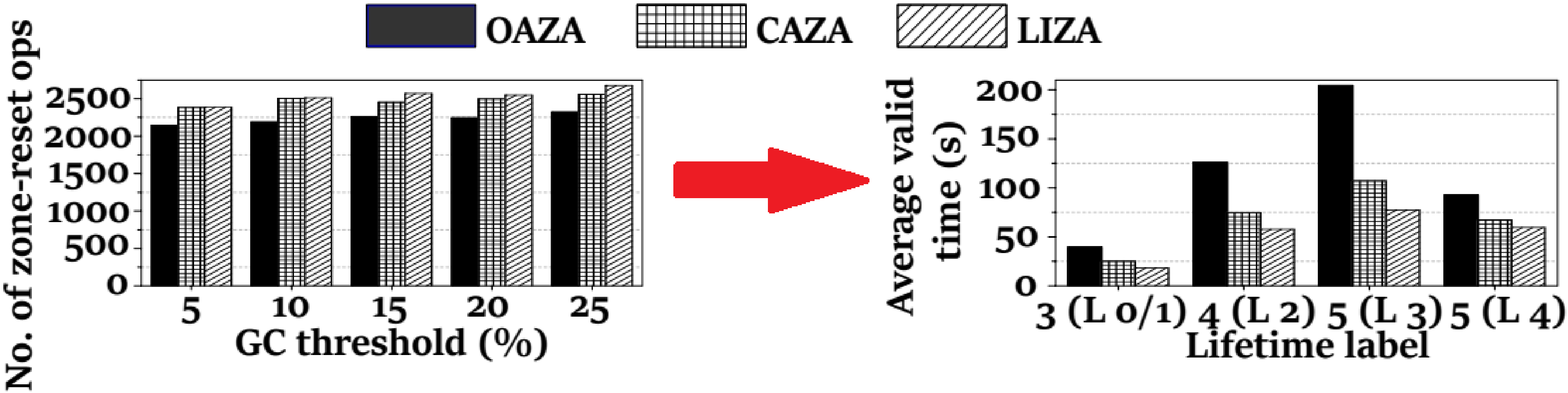
- Most straightforwardly, OAZA notably reduces the amount of data copy which is incurred by GC
  - ➢Reduces by 1.8x-3.6x (avg. 2.7x) compared to LIZA
  - ➢Reduces GC-induced data copy by 1.4x-2x (avg. 1.7x) compared to CAZA

Performance: Write Amp. & Throughput



- OAZA also exhibits
  - ➢Lower write amplification factor (1.1x) than LIZA (1.3x) and CAZA (1.2x) do
  - ➢Higher write throughput (69 MB/s) than LIZA (67 MB/s) and CAZA (68 MB/s) do

Why does OAZA perform better?



- Thanks to more sophisticated data placement, OAZA
  - ➢ Reduces the number of zone-reset operations by 10%, and thus
  - ➢ Prolongs the zone lifetime by 2.2x and 1.7x on average compared to LIZA and CAZA, respectively

Conclusion

- Previous work suffers from
  - ➢Inaccurate lifetime estimation solely based on the LSM-tree levels
  - ➢High write-amplification factors & unfavorable space utilization

- OAZA assign an appropriate zone to a new SSTable
  - ➢Based on vertical and horizontal lifetime estimations

- Compared to LIZA and CAZA, OAZA
  - Reduces the amount of GC-induced data copy by average factors of 2.7× and 1.7×, respectively
  - Achieves a low write-amplification factor of 1.1× (whereas LIZA=1.3× and CAZA=1.2×)

Future Work

- We plan to
  - Apply OAZA to more workloads
  - Implement OAZA in hardware platforms

Thanks for your attention!