Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes

You Zhou[1], Qiulin Wu[1], Fei Wu[1], Hong Jiang[2], Jian Zhou[1], and Changsheng Xie[1]
[1] Huazhong University of Science and Technology (HUST), China
[2] University of Texas at Arlington (UTA), USA

USENIX FAST 21

Duplicate Writes in SSDs

- Flash based SSD development trend:
  - Lower write speed & endurance
  - Lower cost & higher density
- Duplicate writes in SSD
  - Data duplication: 6% ~92% duplicate writes[SmartDedup, ATC'19]
  - Journaling structure: double-write in commit changes and flush
  - Data relocation: garbage collection, file defragmentation
- Eliminate duplicate writes on flash could improve performance and lifetime

Eliminate Duplicate Writes in SSDs

Copy data B from L1 to L2
or
Write duplicate data B to L2

L2P Mappings

LPN	PPN
L0	P3
L1	P1
L2	
L3	

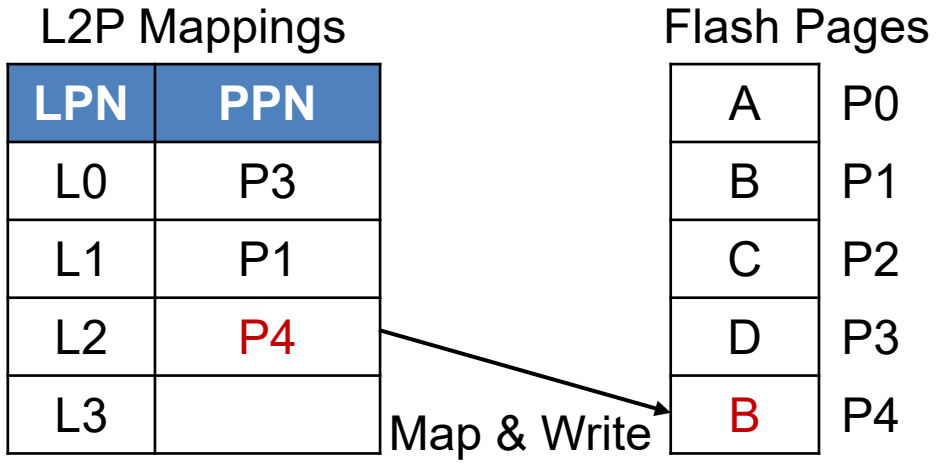
Flash Pages

A	P0
B	P1
C	P2
D	P3
	P4

LPN: host logical page number; PPN: flash physical page number; L2P: logical-to-physical

Eliminate Duplicate Writes in SSDs

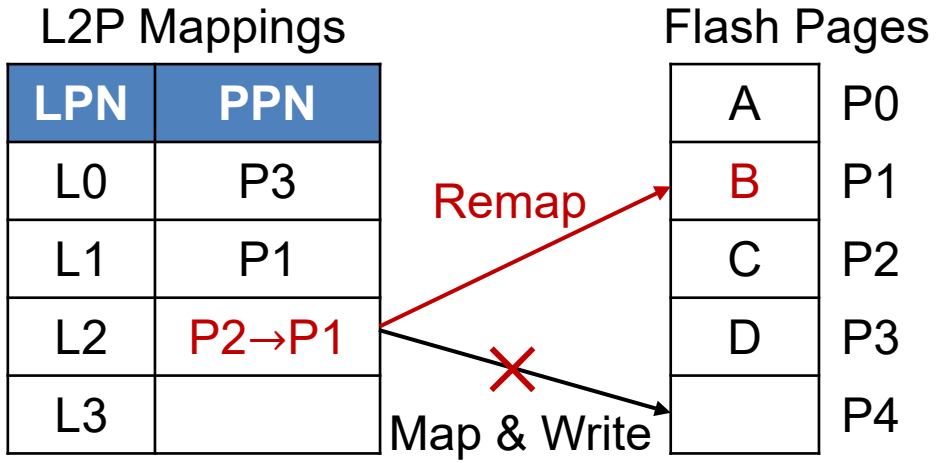
Copy data B from L1 to L2
or
Write duplicate data B to L2



LPN: host logical page number; PPN: flash physical page number; L2P: logical-to-physical

Eliminate Duplicate Writes in SSDs

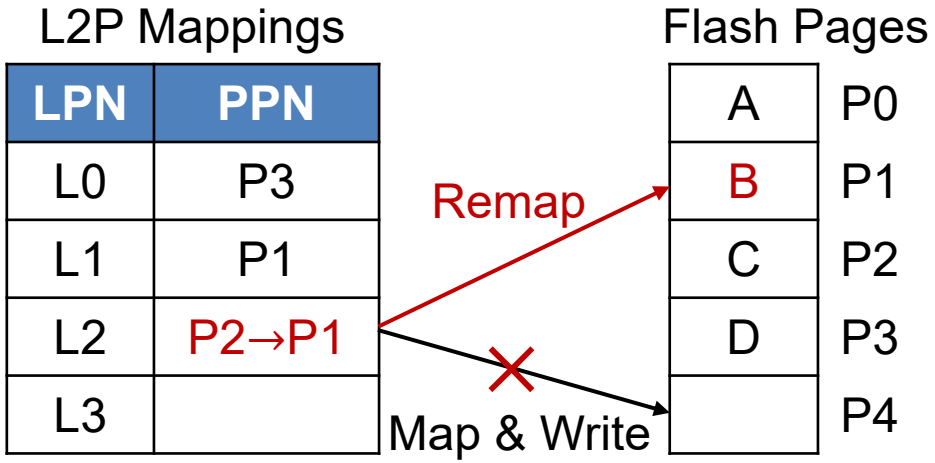
Copy data B from L1 to L2
or
Write duplicate data B to L2



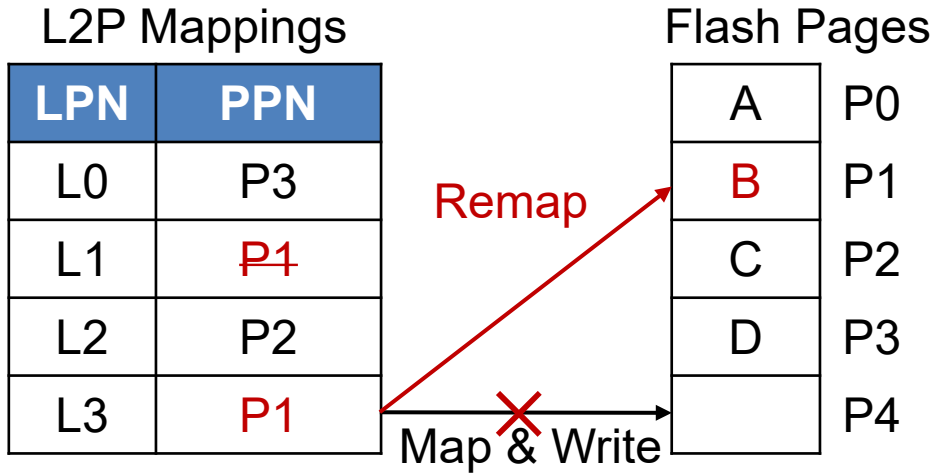
LPN: host logical page number; PPN: flash physical page number; L2P: logical-to-physical

Eliminate Duplicate Writes in SSDs

Copy data B from L1 to L2
or
Write duplicate data B to L2



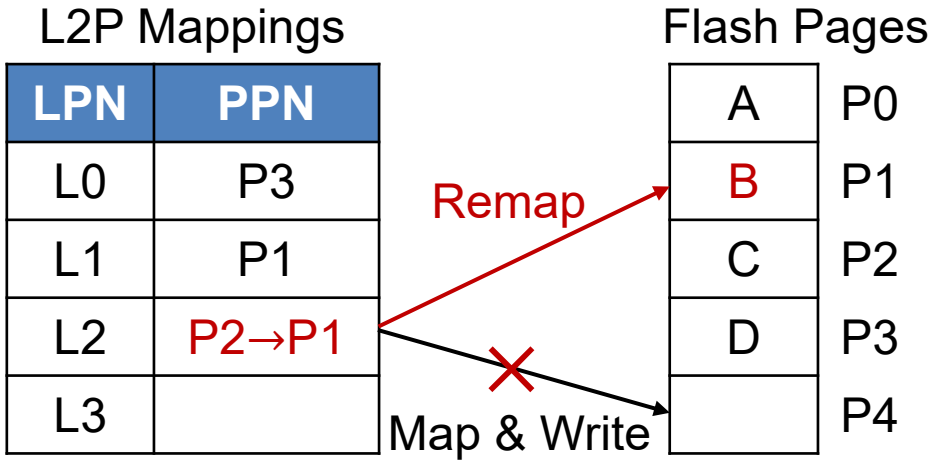
Move data B from L1 to L3



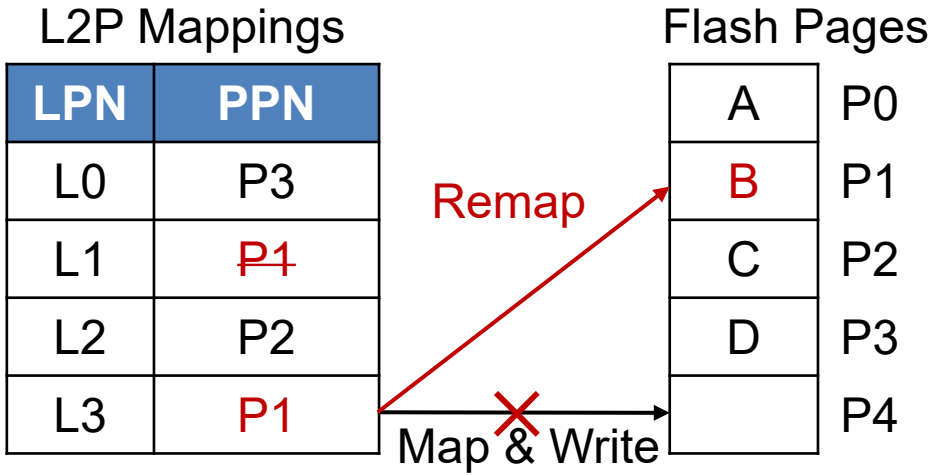
LPN: host logical page number; PPN: flash physical page number; L2P: logical-to-physical

Eliminate Duplicate Writes in SSDs

Copy data B from L1 to L2
or
Write duplicate data B to L2



Move data B from L1 to L3

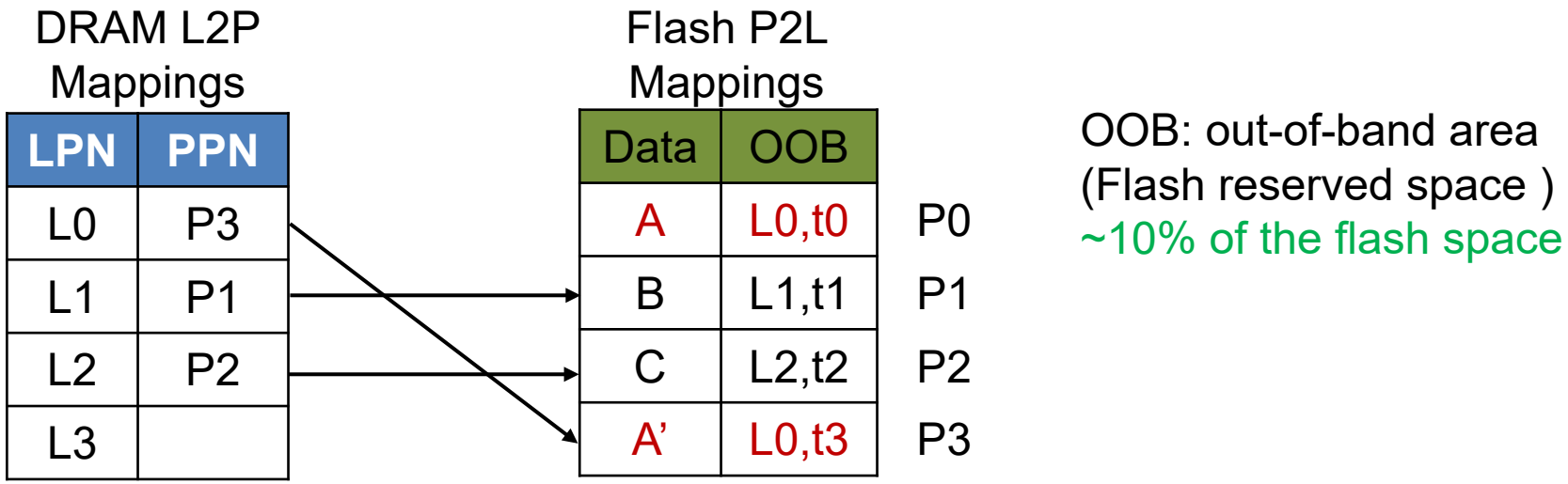


LPN: host logical page number; PPN: flash physical page number; L2P: logical-to-physical

- Use remap to reduce the number of writes

Mapping Inconsistency Problem with Remapping

- P2L mappings are necessary in modern SSD
  - Garbage collection (GC): locate and modify relevant L2P mappings
  - Power-off recovery (POR): restore L2P mappings

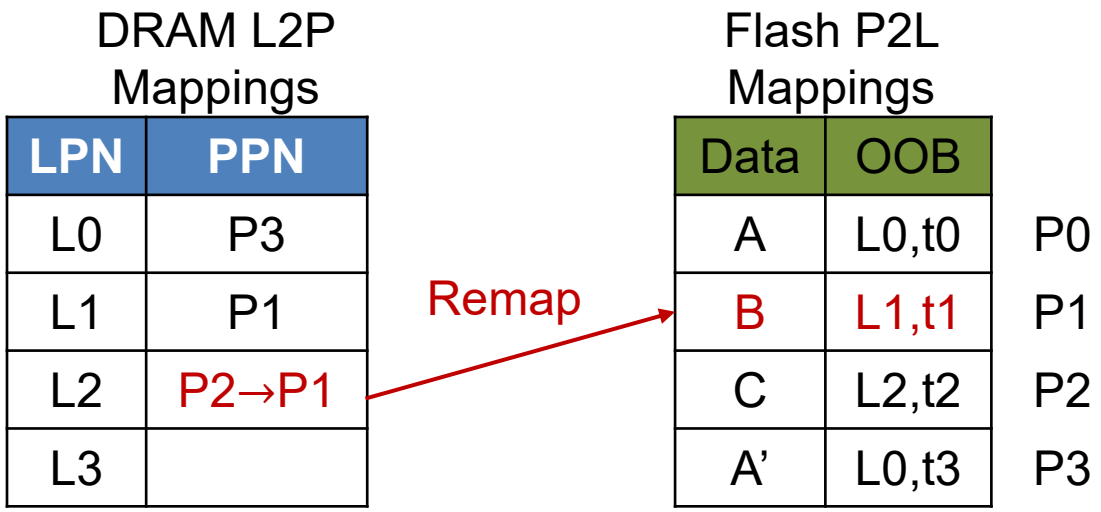


OOB: out-of-band area
(Flash reserved space)
~10% of the flash space

- But inconsistent L2P and P2L mappings lead to corrupted L2P mappings after GC/POR

Mapping Inconsistency Problem with Remapping

- Remap L2 to P1 for copying data B from L1 to L2
  - Lead to {L1, L2} → P1 in L2P mapping, but P1 → L1 in P2L mapping



Mapping Inconsistency Problem with Remapping

- Remap L2 to P1 for copying data B from L1 to L2
  - Lead to {L1, L2} → P1 in L2P mapping, but P1 → L1 in P2L mapping

DRAM L2P Mappings

LPN	PPN
L0	P3
L1	P1
L2	P2→P1
L3	

Remap

Flash P2L Mappings

Data	OOB	
A	L0,t0	P0
B	L1,t1	P1
C	L2,t2	P2
A’	L0,t3	P3

After …

GC migrates data B to P1’

LPN	PPN
L1	P1→P1’
L2	P1(erased)

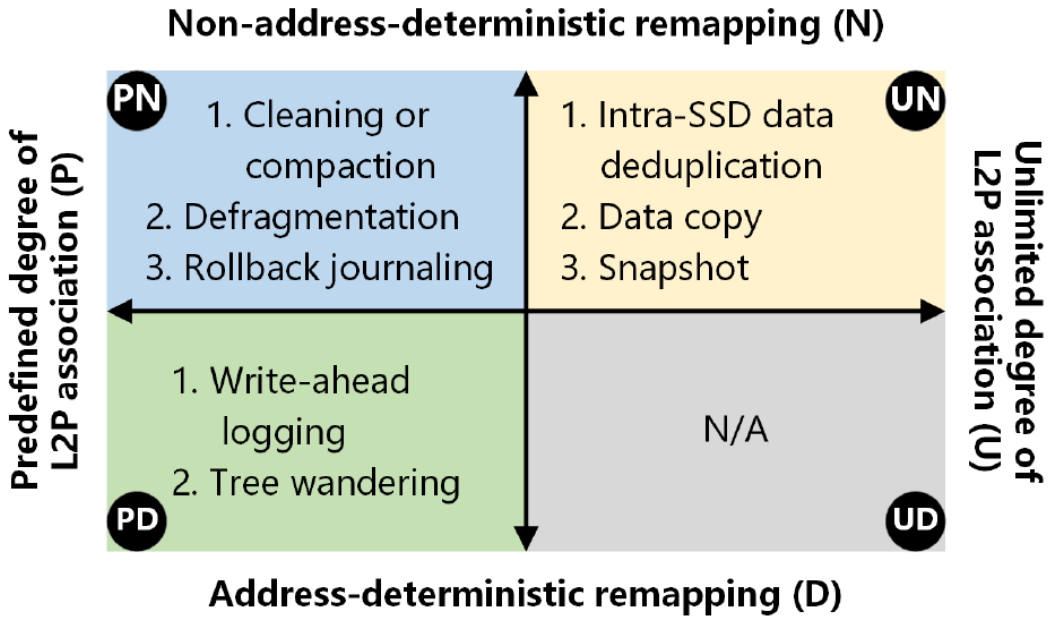
Recover L2P via P2L

LPN	PPN
L1	P1
L2	P1(invalid)

Inconsistent L2P and P2L mappings lead to corrupted L2P mappings after GC/POR

Existing Works on Remapping

- Two dimensions of remapping
  - M-to-1 L2P mappings:
    - M is limited or unlimited
  - Target addresses of remappings are predetermined or uncertain
    - Write-ahead logging: M = 2, remap data from log to predetermined home locations
    - Data deduplication: M and addresses of remappings depend on workloads



- How do they solve the inconsistency problem?

Approaches to Solve Inconsistency Problem

- Persist a device-wide log

P,L,t	P,L,t	P,L,t	P,L,t	P,L,t	P,L,t	P,L,t

  - High lookup overheads & poor scalability
    - Log size grows: ~1GB for 300GB remap data
    - Log scanning in every GC takes seconds
  - Or limit the log size by disabling remapping

- NVRAM OOB
  - Only fit in PN & PD-type remapping scenarios

Data	NVRAM OOB
B	(L1,t1), (L2, t4)

- Prewrite L2 to P1 OOB
  - Only fit in PD-type scenarios

L2P Mappings

LPN	PPN
L1	P1
L2	P2→P1

Remap

P2L Mappings

Data	OOB	
B	L1,t1,L2	P1
C	L2,t2	P2

The Problem & Main Idea

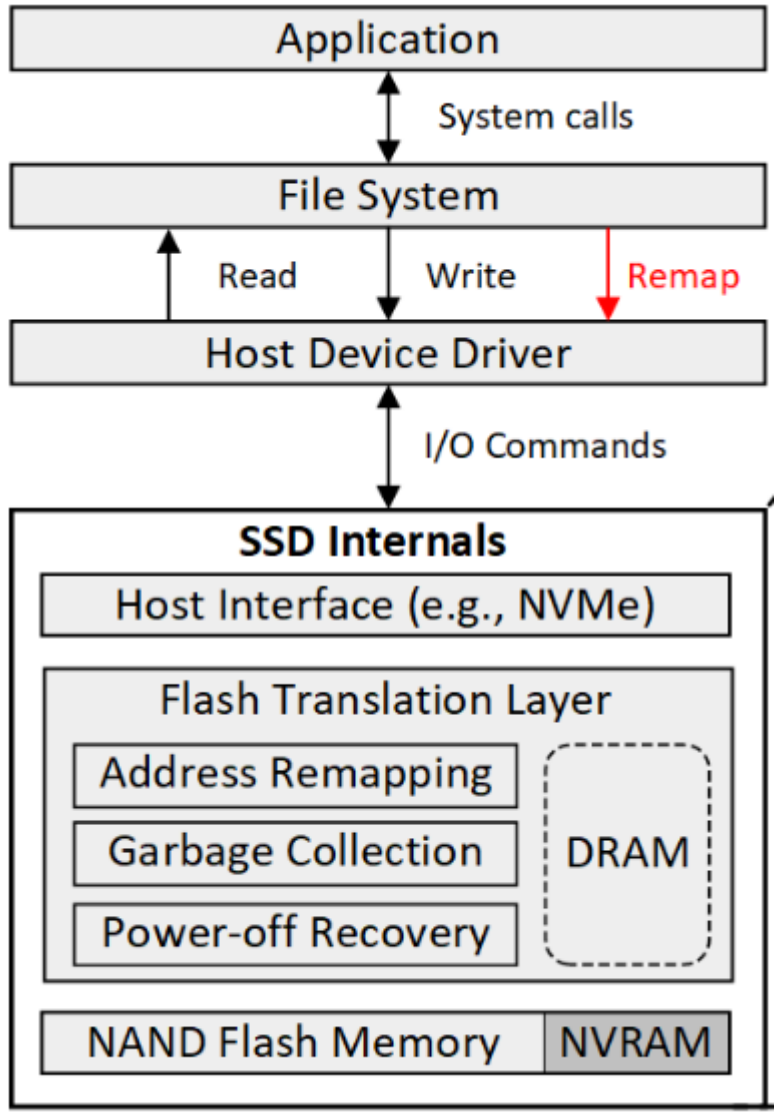
- Mapping inconsistency problem cannot be addressed properly, existing schemes severely limit the usage of SSD address remapping
- On demand, small local logs in segmented NVRAM for remap metadata management
  - Fast lookups and persistence of P2L mappings
  - Small local logs in segmented NVRAM for flash GC units

Overview

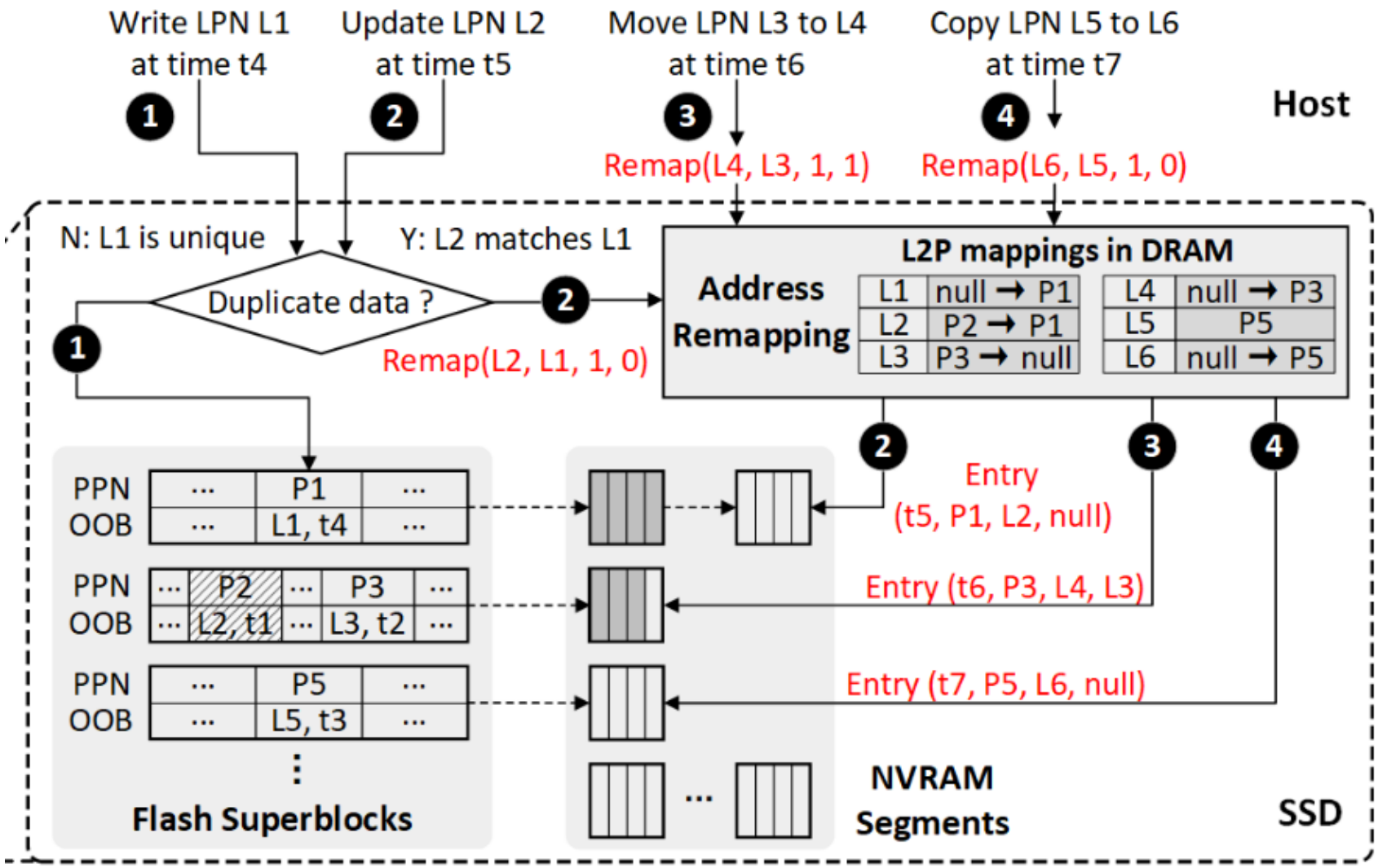
- Goal: full potentials of remapping
  - Use NVRAM for remapping metadata storage
  - Support GC and POR

Remap command complete:

- **L2P mappings modified in DRAM**
- **Relevant RMM entries persisted on NVRAM.**



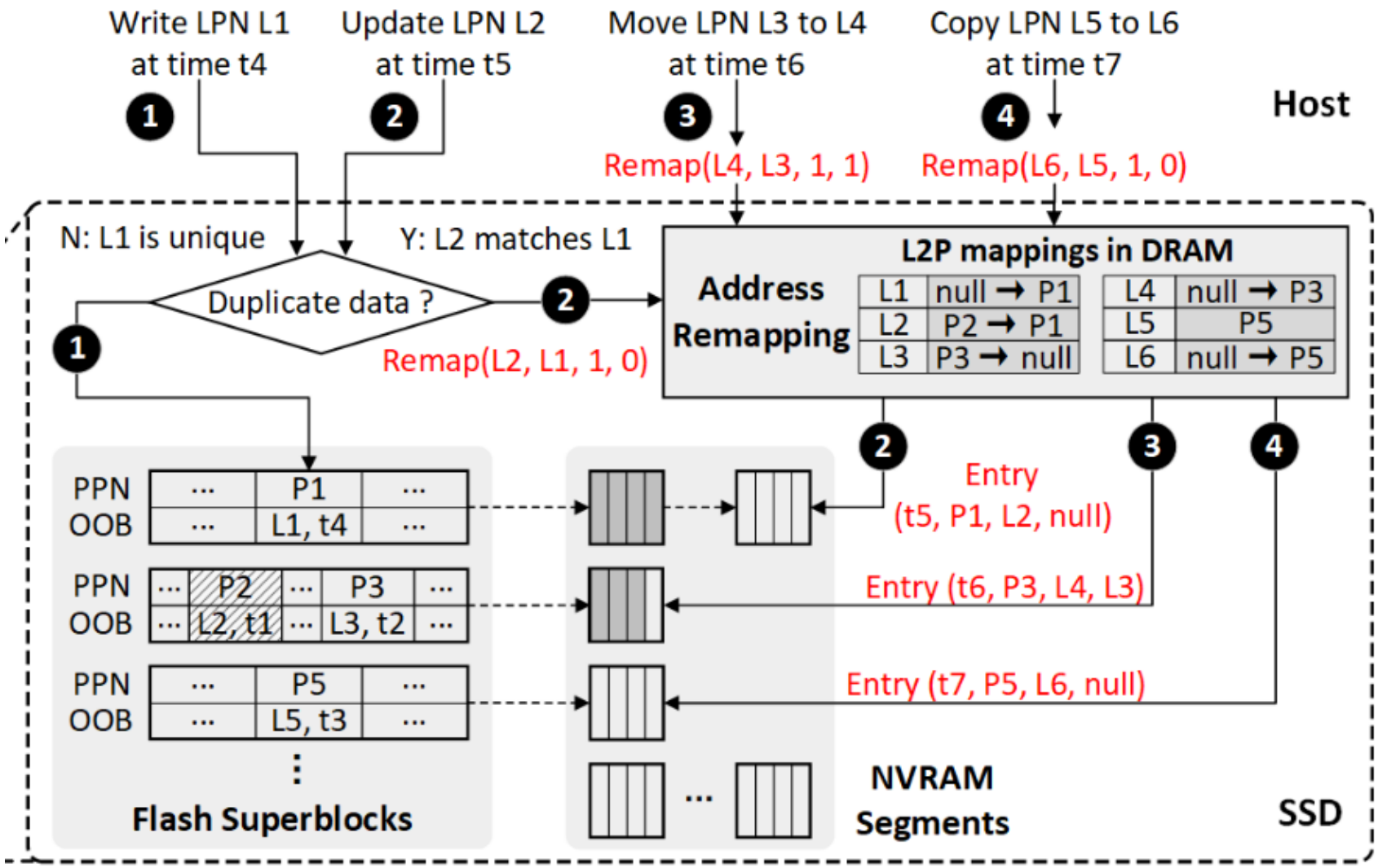
Structure



- **3, 4** are invoked by host software
- **2** is invoked by FTL internally (e.g., by an intra-SSD deduplication engine)
- remapFlag = 1: relocation
- remapFlag = 0: data copy

➢ remap(tgtLPN, srcLPN, length, remapFlag)

Structure



- The P2L mapping is kept on SSD OOB
- Additionally store entries on NVRAM segments for each flash superblock

- Entry(timeStamp, target PPN, LPN, alterable field)

Structure

- NVRAM support 8-byte atomic writes
  - Remapping metadata is 2 x 8-bytes
  - Completely written entries will have all “1” torn bits

First 8 Bytes	
0	Torn bit
1-21	Flash page offset in superblock
22-63	Write/Remap sequence number

Second 8 Bytes	
64	Torn bit
65-95	Target LPN
96	Remapping flag
97-129	Null or source LPN

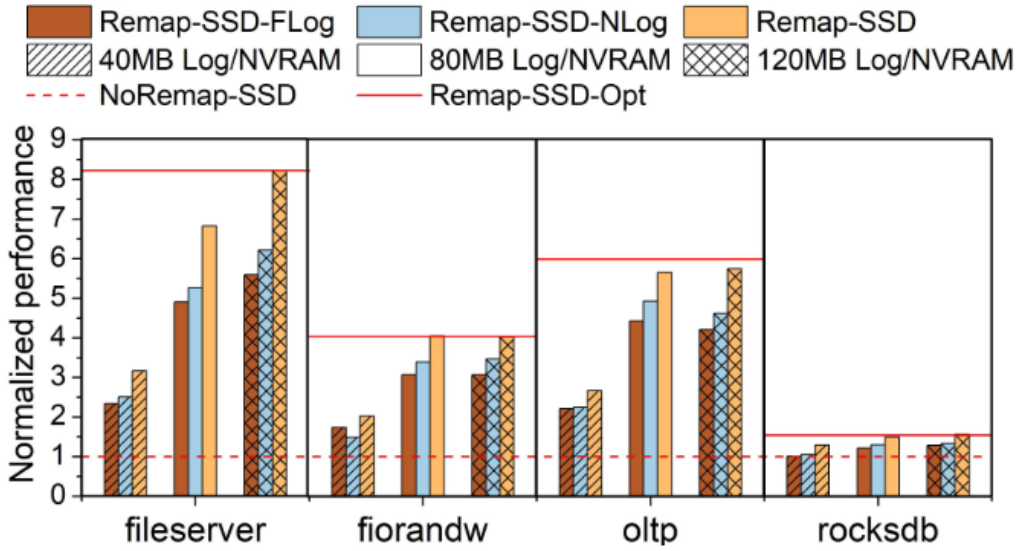
- Discard entries with “0” torn bit for consistency

Evaluation

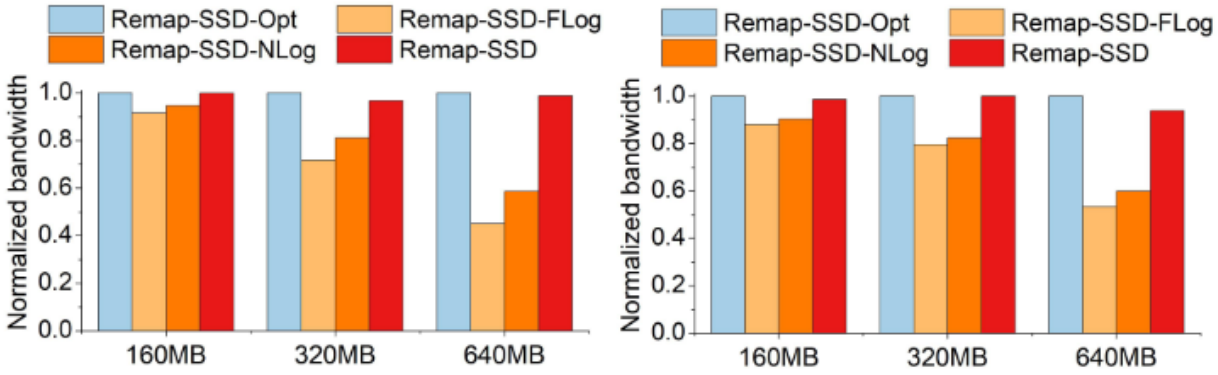
- Platforms
  - FEMU SSD emulator (32 GB): Filebench, FIO, YCSB on RocksDB/MongoDB, db_bench
  - SSDsim simulator (256 GB): FIU dedup traces
- Comparison
  - NoRemap-SSD: baseline
  - Remap-SSD-Flog: existing scheme with device-wide log on flash memory
  - Remap-SSD-Nlog: store device-wide log on NVRAM
  - Remap-SSD-Opt: no limits remapping and no P2L lookup overheads
- Case studies
  - Data deduplication; Write-ahead logging in SQLite; GC in F2FS

Data deduplication

- Performance results with different log/NVRAM sizes
  - Small log/NVRAM limit the usage of remapping
  - Large log/NVRAM cause high P2L lookup overheads



32GB SSD, 40MB-80MB-120MB log/NVRAM
(simulated content locality, 30% duplicate data)

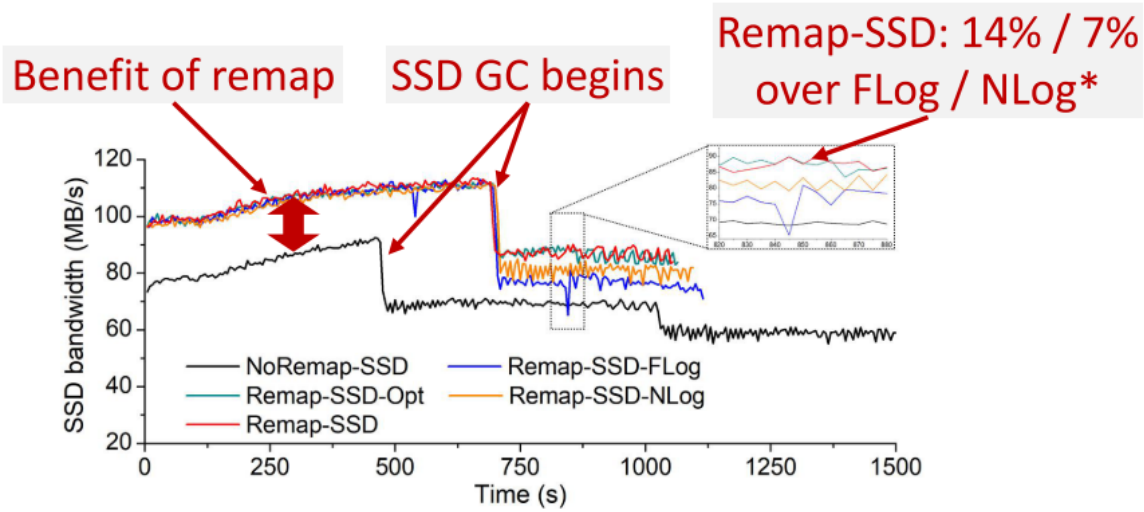


256GB SSD, 160MB-320MB-640MB log/NVRAM
(real-world dedup traces *homes* and *mail*)

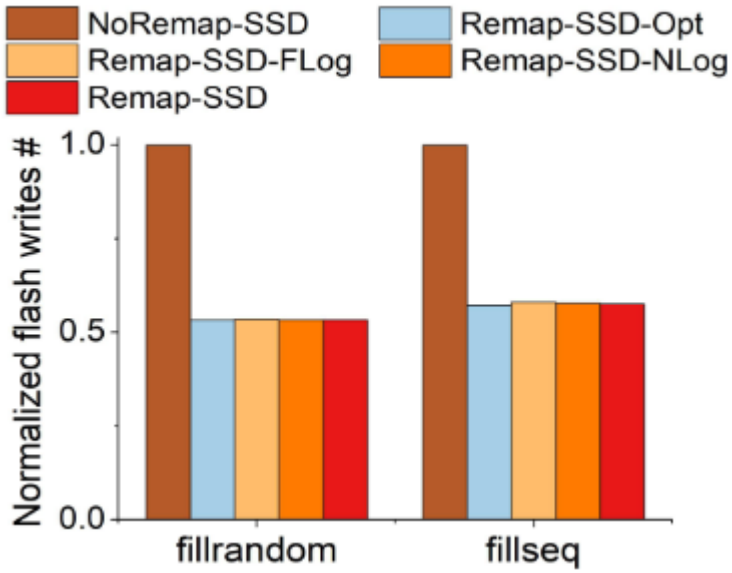
Remap-SSD: near-optimal performance & scalability

SQLite WAL

- Remapping enables single-write write-ahead-logging (WAL)



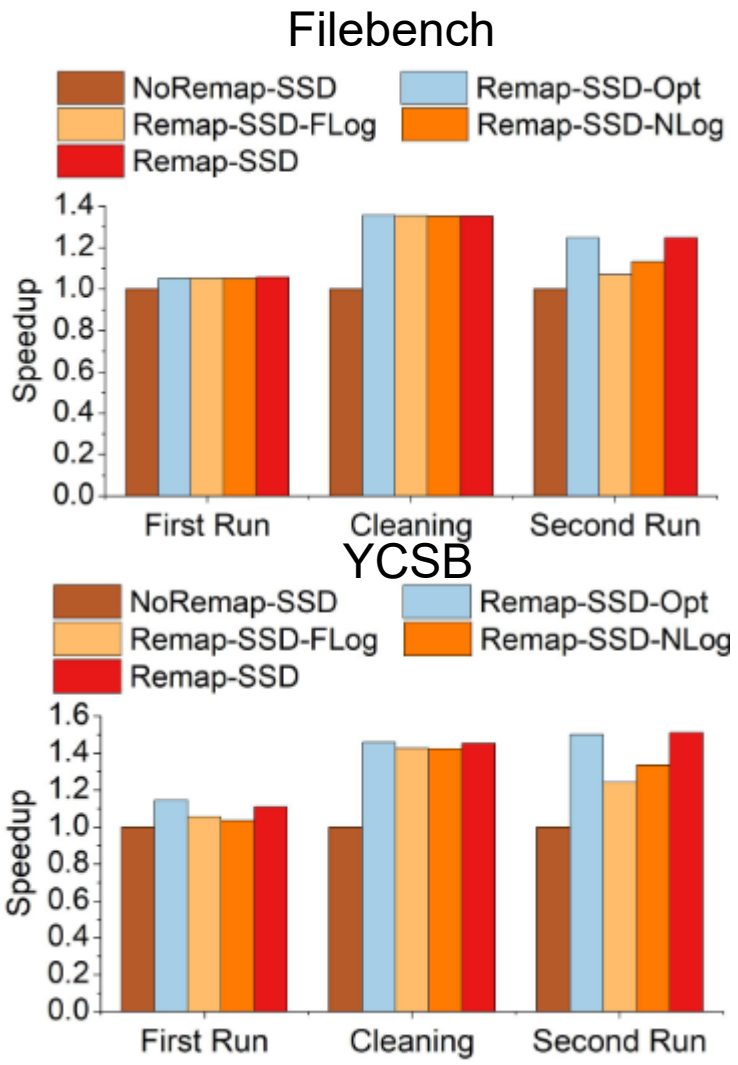
SSD bandwidth in fill-random (db_bench)



45% fewer flash writes

F2FS Cleaning

- First run of workload to age F2FS
  - Similar performance: few clean/remap operations
- Cleaning F2FS until all invalid data are reclaimed
  - Accumulate remapping metadata entries
  - **Remapping accelerates cleaning by 28%**
- Second run of workload to show performance
  - **Remap-SSD improves performance by 19% over FLog and 12% over NLog.**



Conclusion

- Remapping can eliminate duplicate writes but its usage is limited due to the L2P and P2L mapping inconsistency problem
  - A device-wide log for P2L mappings: high lookup overheads
  - Other solutions: only specific remapping scenarios
- Remap-SSD
  - Remap primitive: logical writes of duplicate data
  - Remap metadata: mapping consistency, remapping atomicity
  - Metadata management: fast lookups and persistence of P2L mappings
    - Maintain small local logs in segmented NVRAM for flash GC units on demand