Designing a Flash Translation Layer

19 February, 2016

Niv Dayan

Introduction

- Niv Dayan (post-doc)
  - Data Systems Lab
  - IACS

- Today's talk:
  - Designing a Flash Translation Layer

- Hopeful learning outcomes
  - What flash looks like internally
  - See a cool application of LSM-trees

Introduction

- Structure
  - Background (how flash works)
  - Scalability Problem (for terabyte flash devices)
  - Solution (LSM-tree variant)
- Questions/discussions are welcome

Background

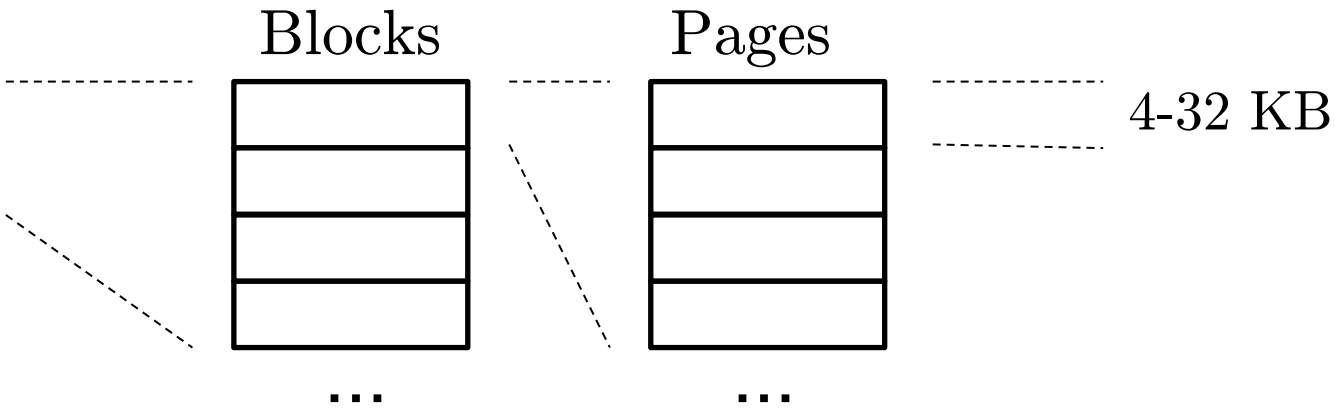
Background

- Flash devices increasingly used for many applications

- Advantages:
  - Good read/write performance
  - Low power consumption

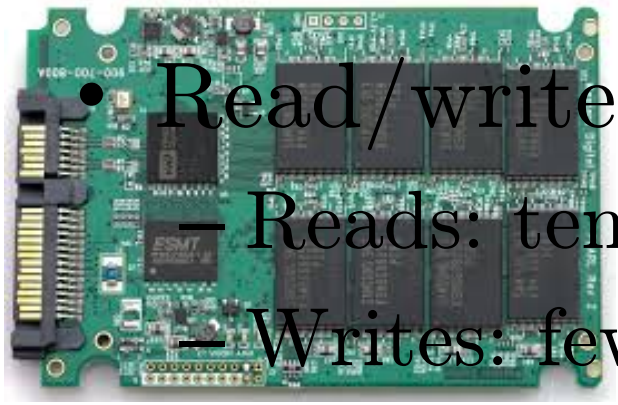
- Internally they are very complex

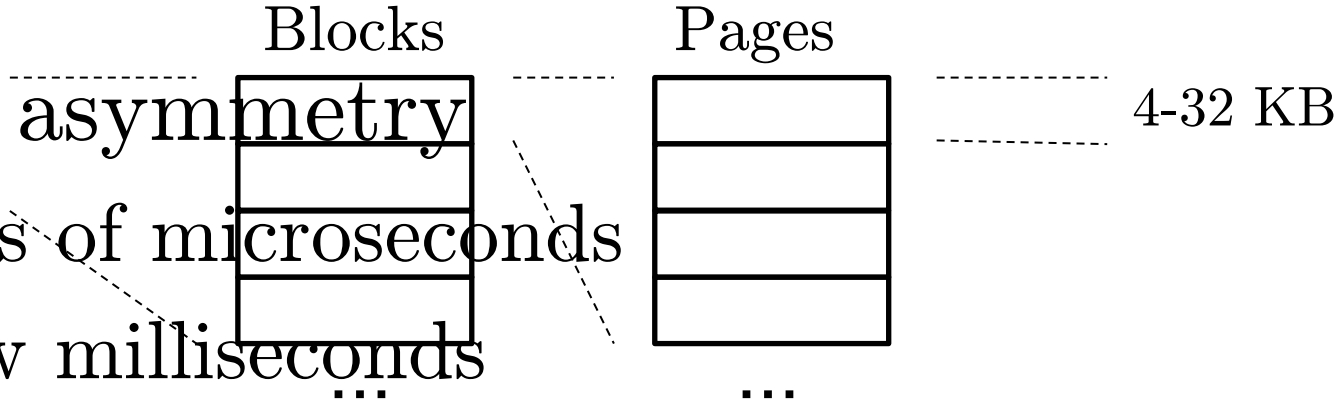
Background



- Ideosyncracies
  - Reads & writes at page granularity
  - Sequential writes within a block
  - Block-erase before update
  - Limited erases per block

Background

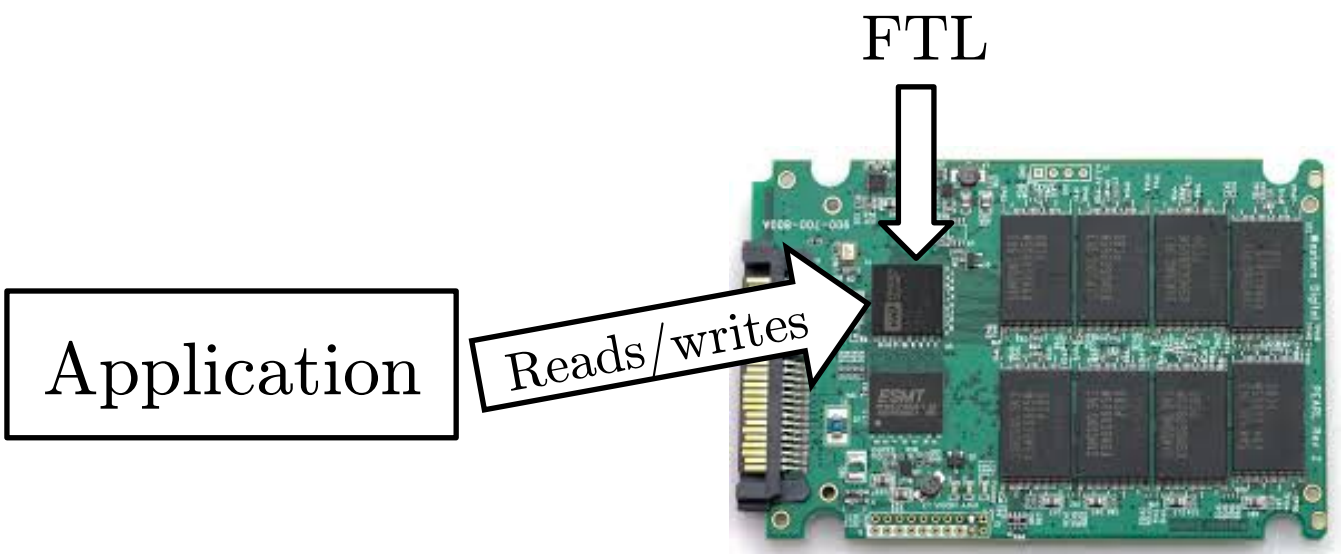




- Read/write asymmetry
  - Reads: tens of microseconds
  - Writes: few milliseconds

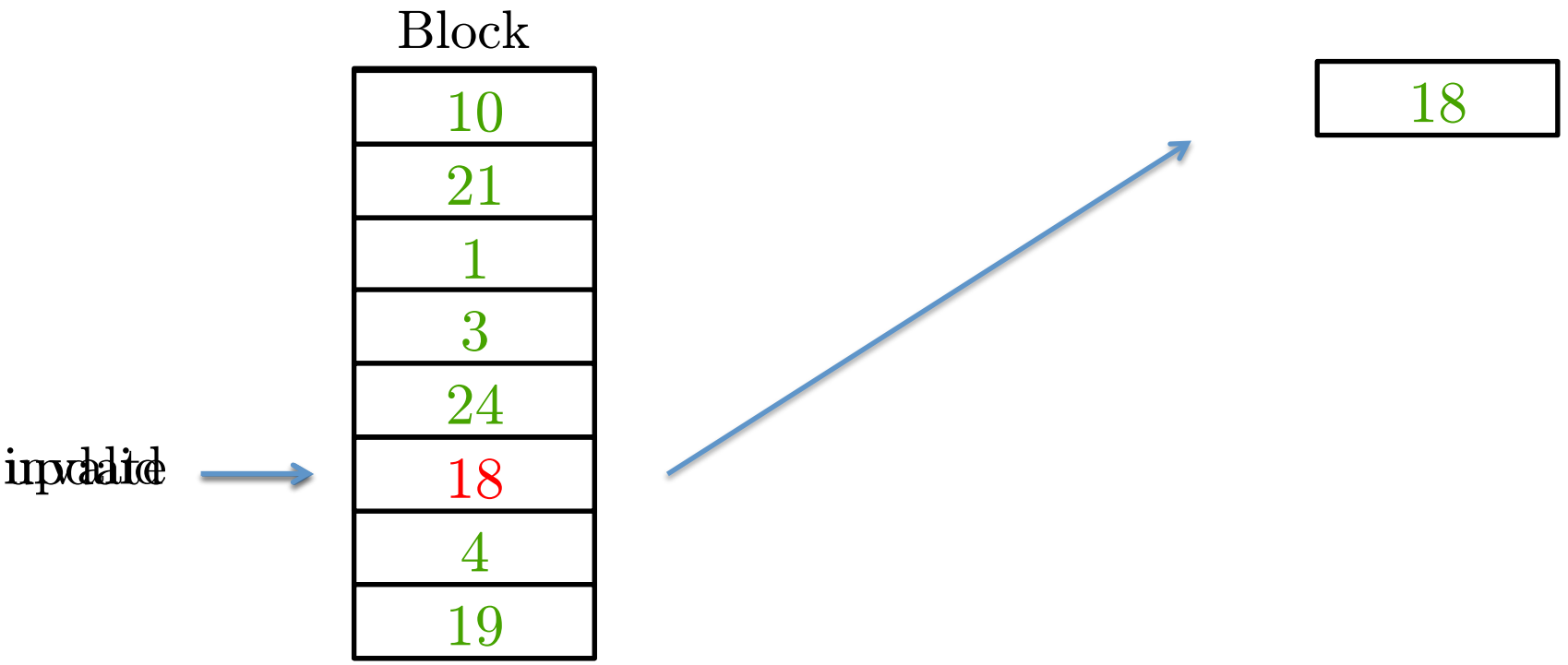
Background

- A flash translation layer (FTL) hides these constraints
- Exposes a simple block interface to Application



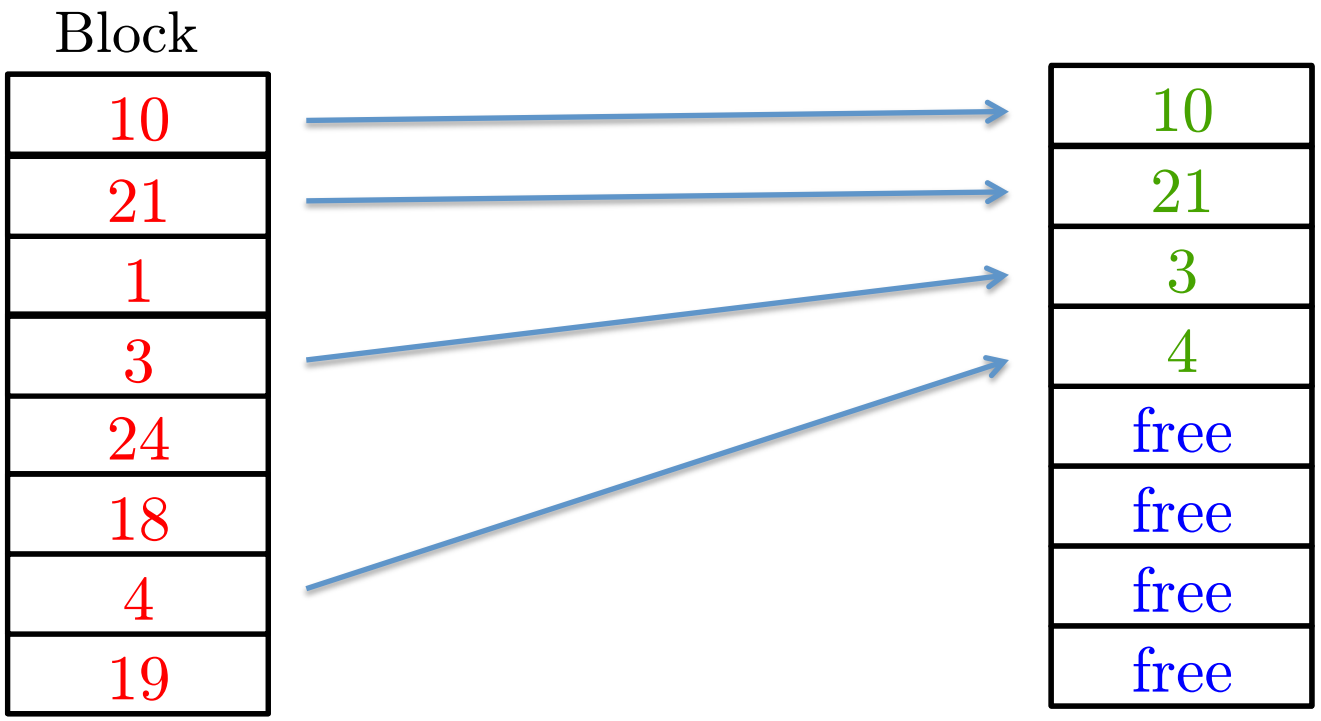
Background

- FTL writes pages sequentially within a block
- Recall erase-before-write rule
- Reduce cost through out-of-place updates



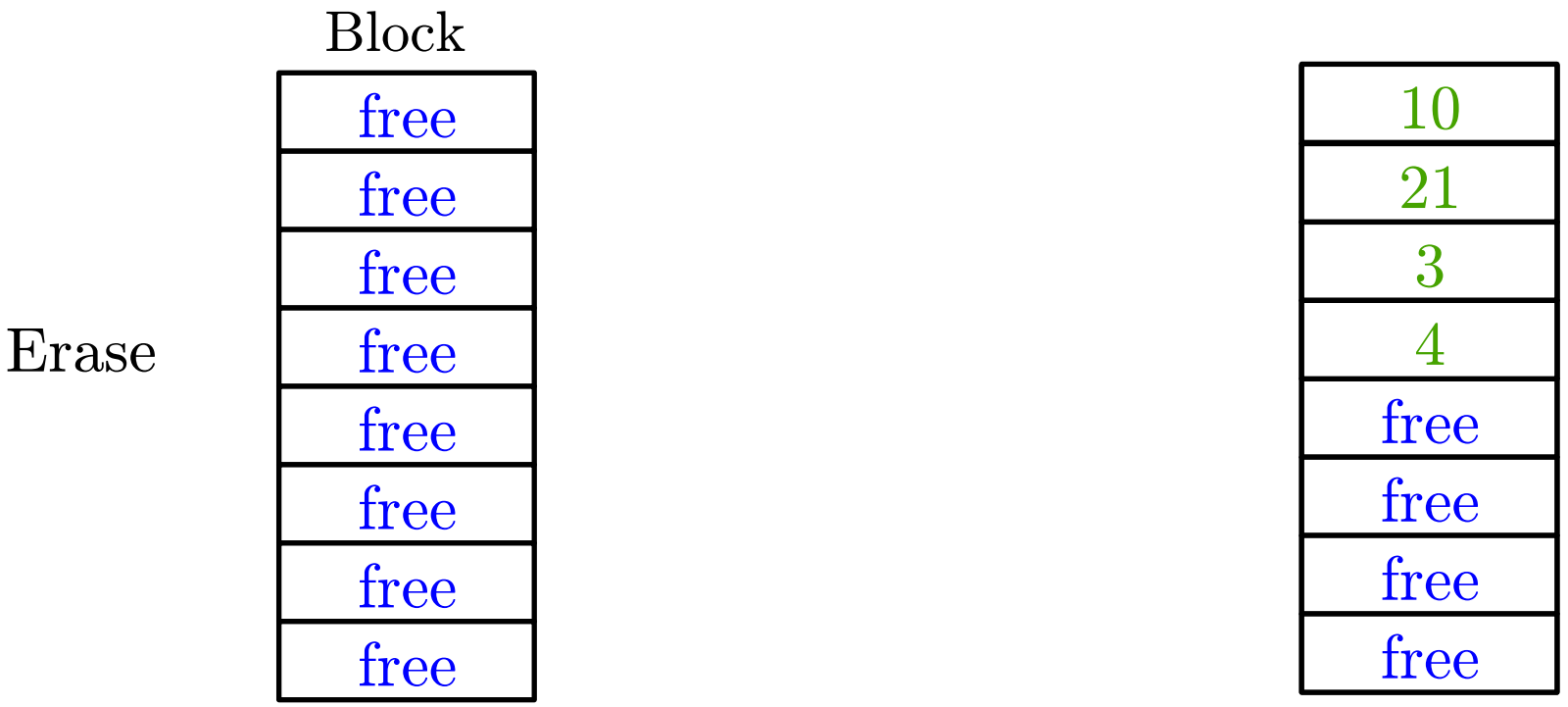
Background

- Eventually, invalid pages accumulate
- Perform garbage-collection to free space



Background

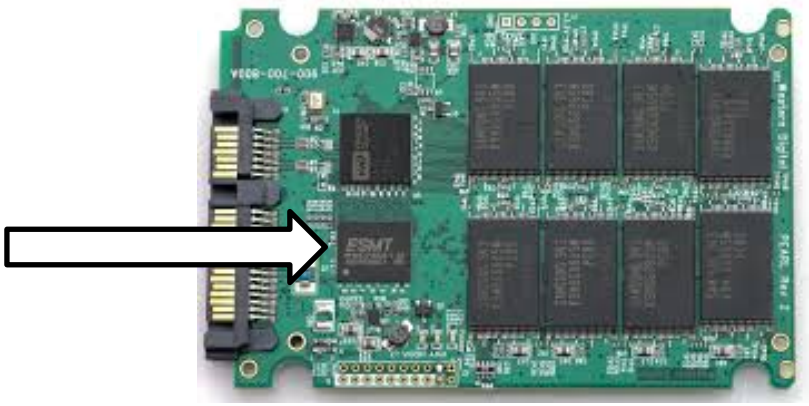
- Eventually, invalid pages accumulate
- Perform garbage-collection to free space



Background

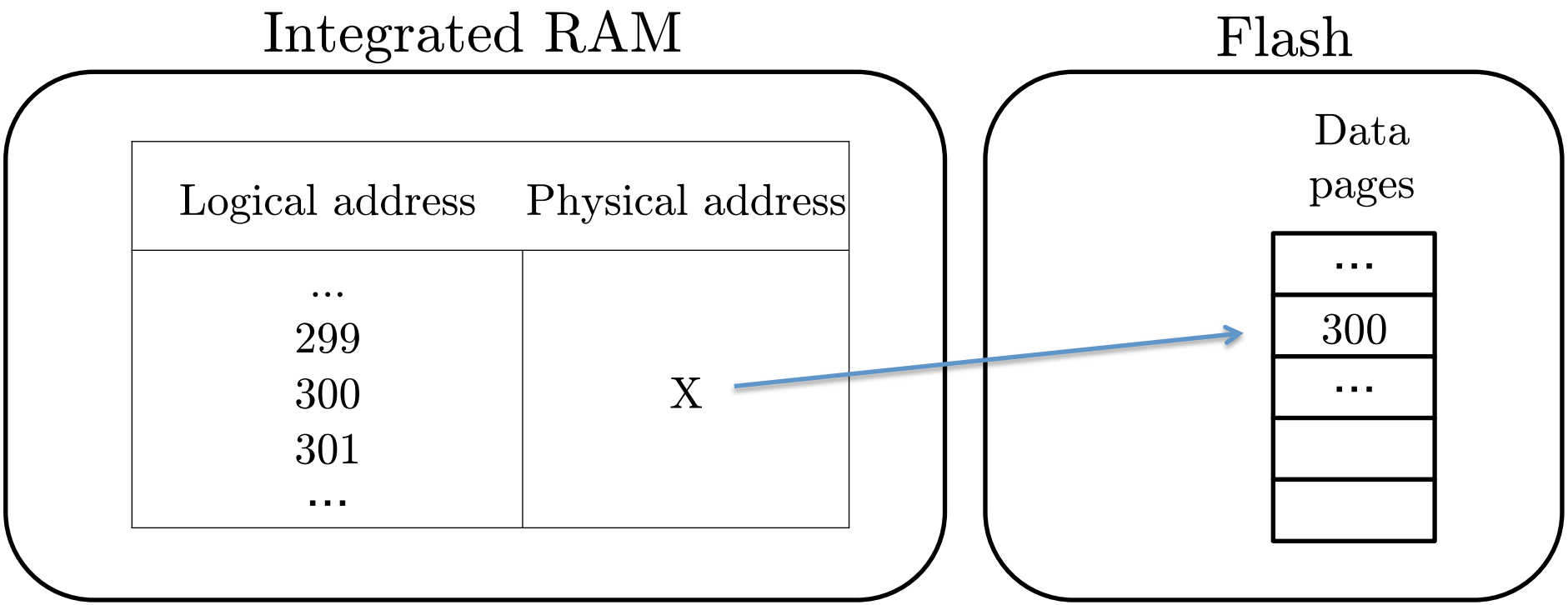
- Logical pages move around
- The FTL must maintain a mapping table from logical to physical addresses
- Device has RAM for storing metadata

Integrated RAM



Background

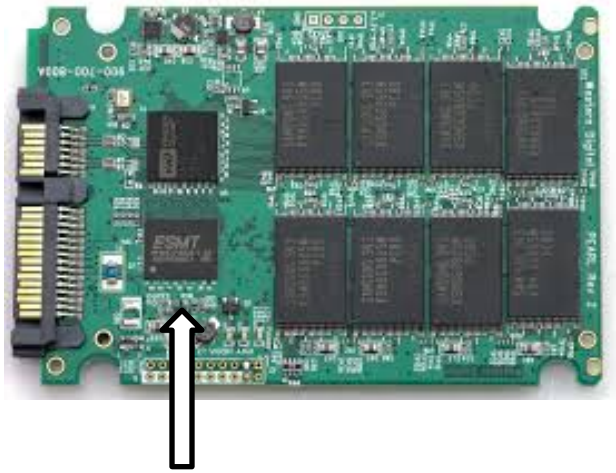
- **Example**
  - Application reads logical page 300



Background

- **Problem**
  - The mapping table is large
  - For a 2 TB flash device, 2 GB mapping table
  - Not enough integrated RAM

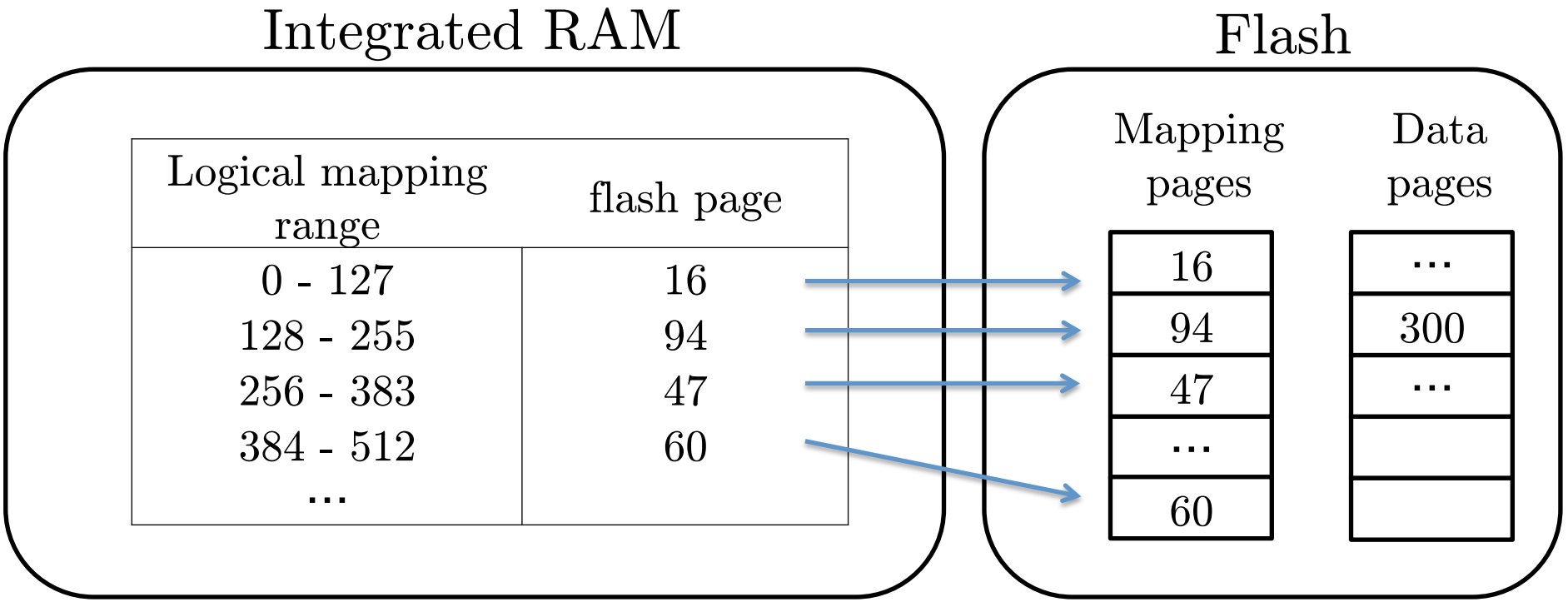
- Solution?
  - Store it in flash



Integrated RAM

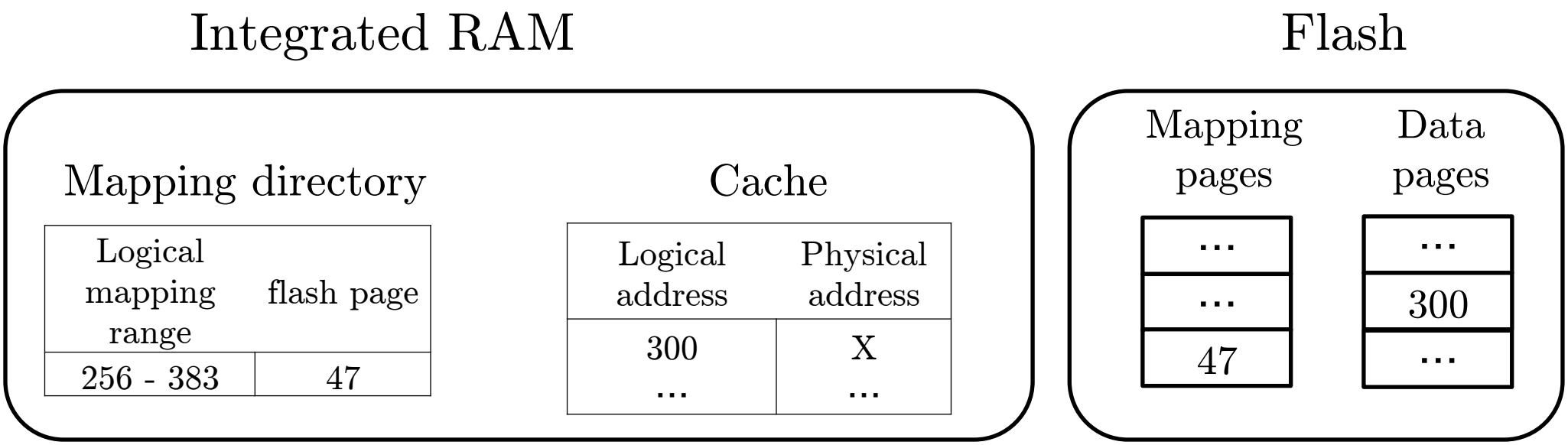
Background

- Mapping table is stored in flash
- E.g. Application reads page 300
- We pay in performance



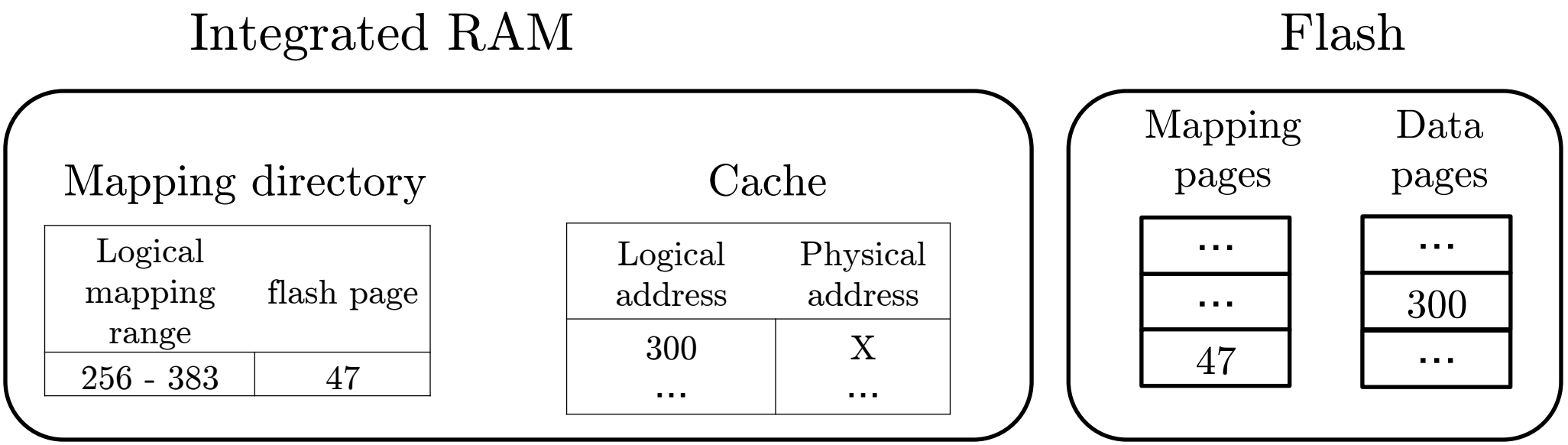
Background

- To reduce the overhead of mapping reads, we use a cache



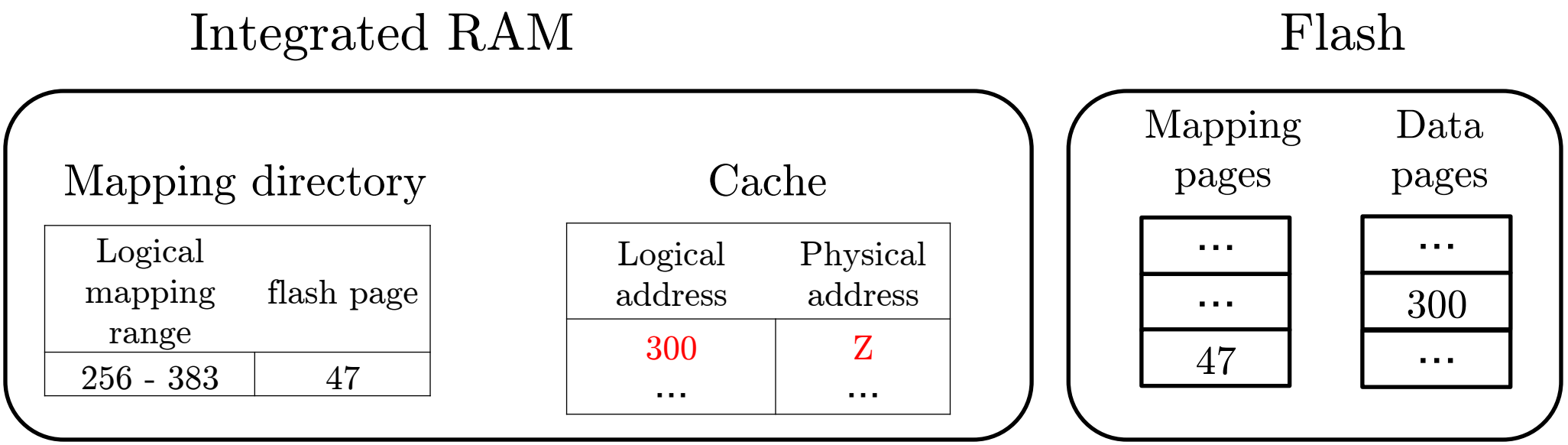
Background

- E.g. suppose application updates page 300
- We must update mapping page 47
- This can harm performance.
- Mark entry as “dirty”. Update lazily later.



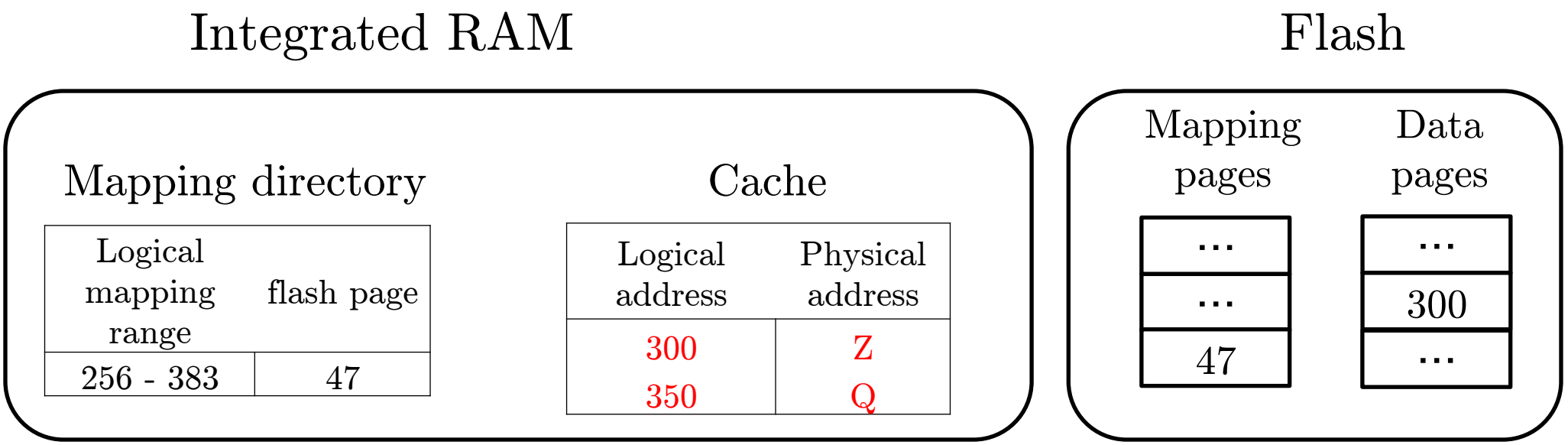
Background

- E.g. suppose application updates page 300
- We must update mapping page 47
- This can harm performance.
- Mark entry as “dirty”. Update lazily later.



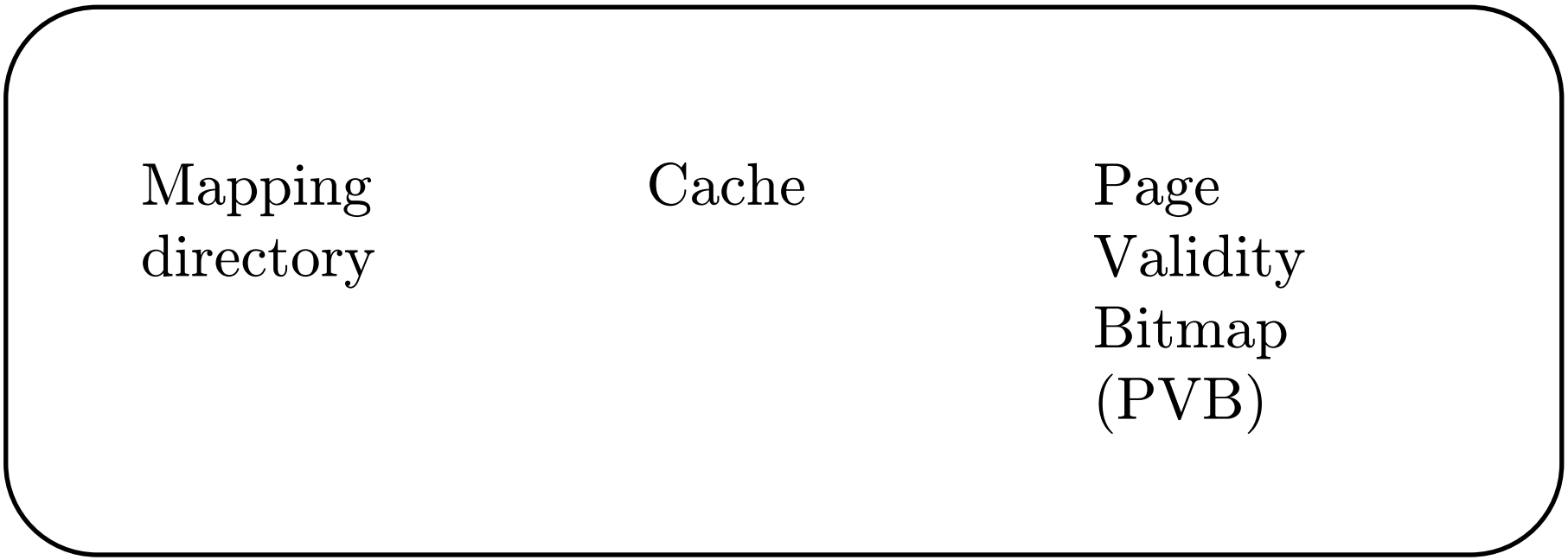
Background

- E.g. suppose application updates page 300
- We must update mapping page 47
- This can harm performance.
- Mark entry as “dirty”. Update lazily later.



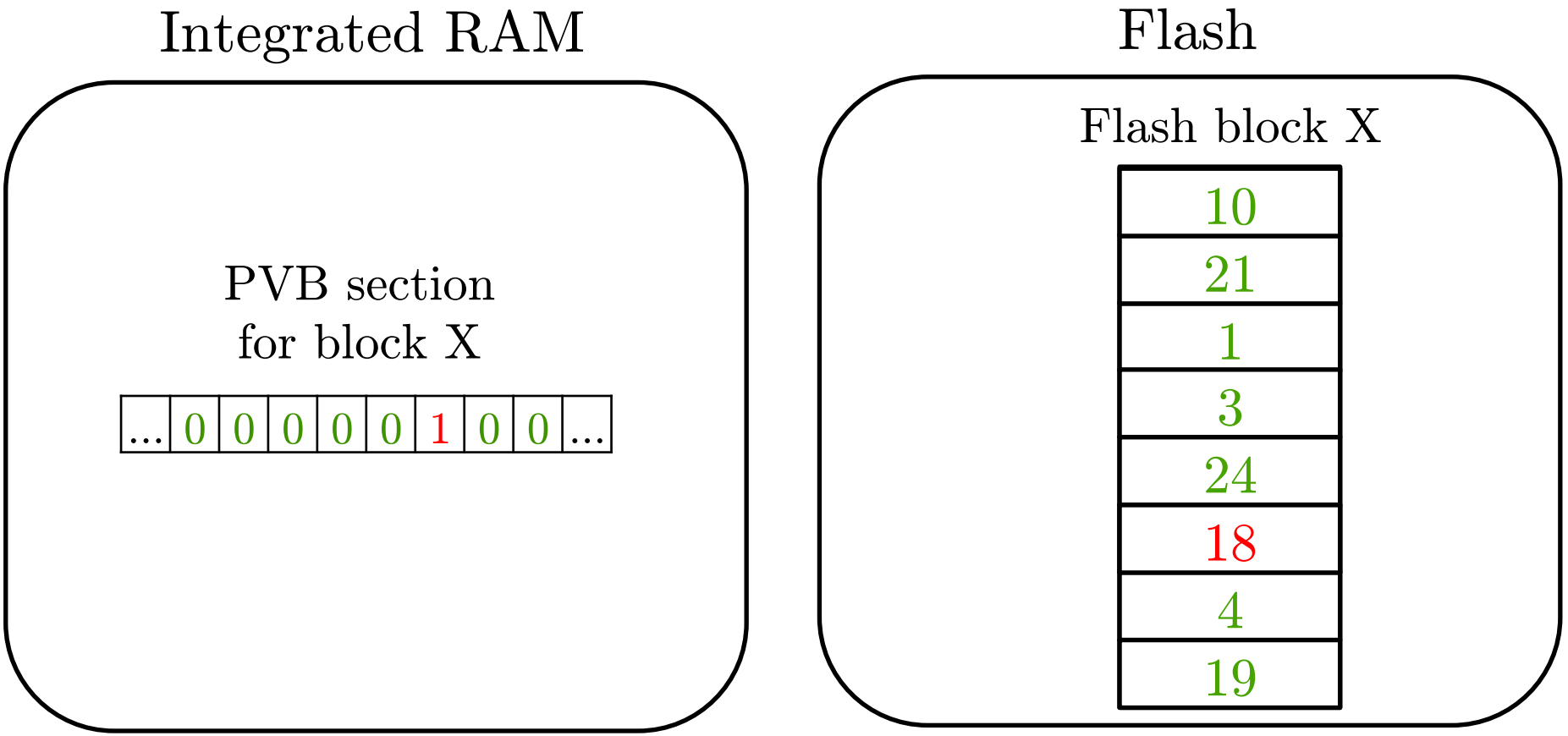
Background

- The FTL also maintains a bitmap
- Keeps track of which pages are invalid
- Needed for garbage-collection



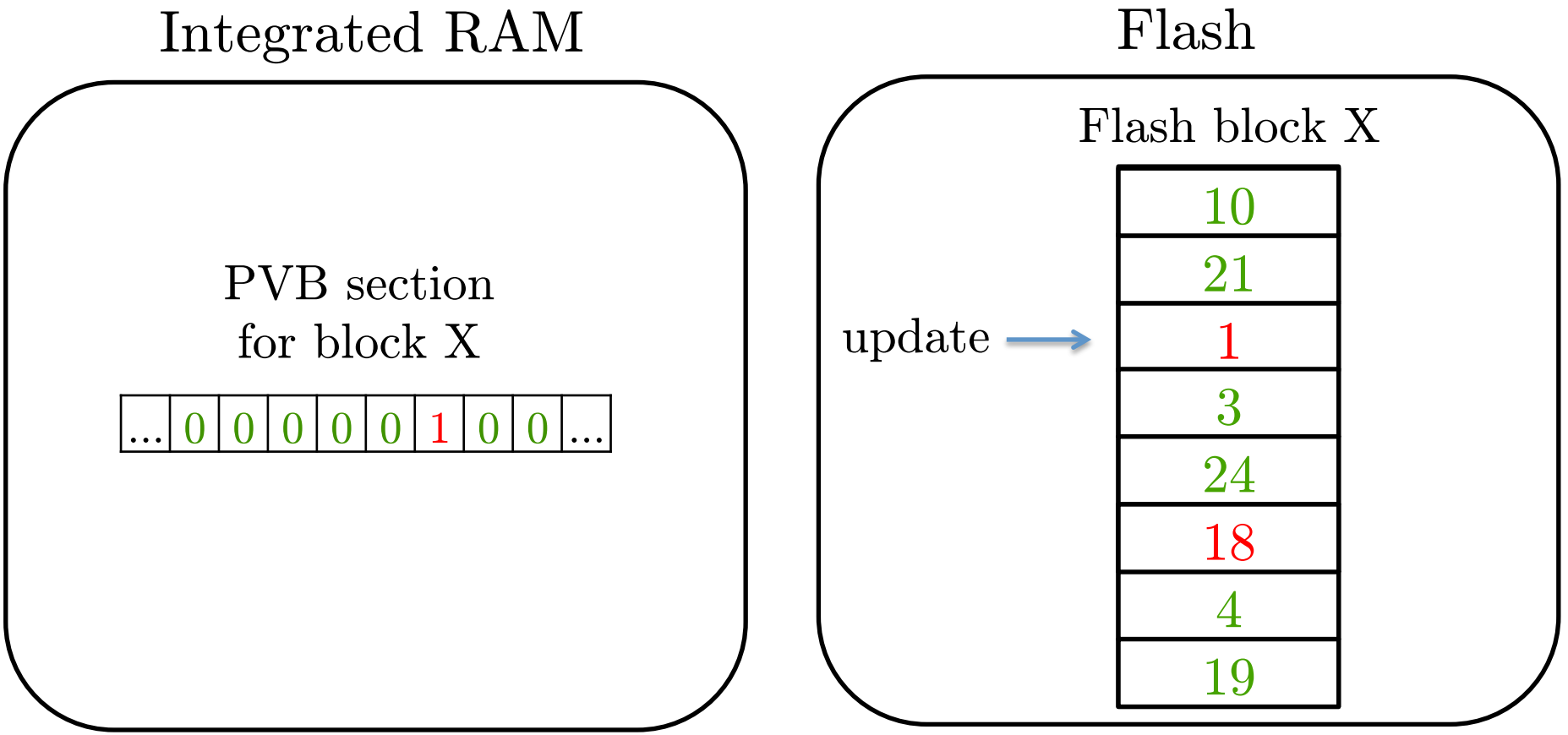
Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



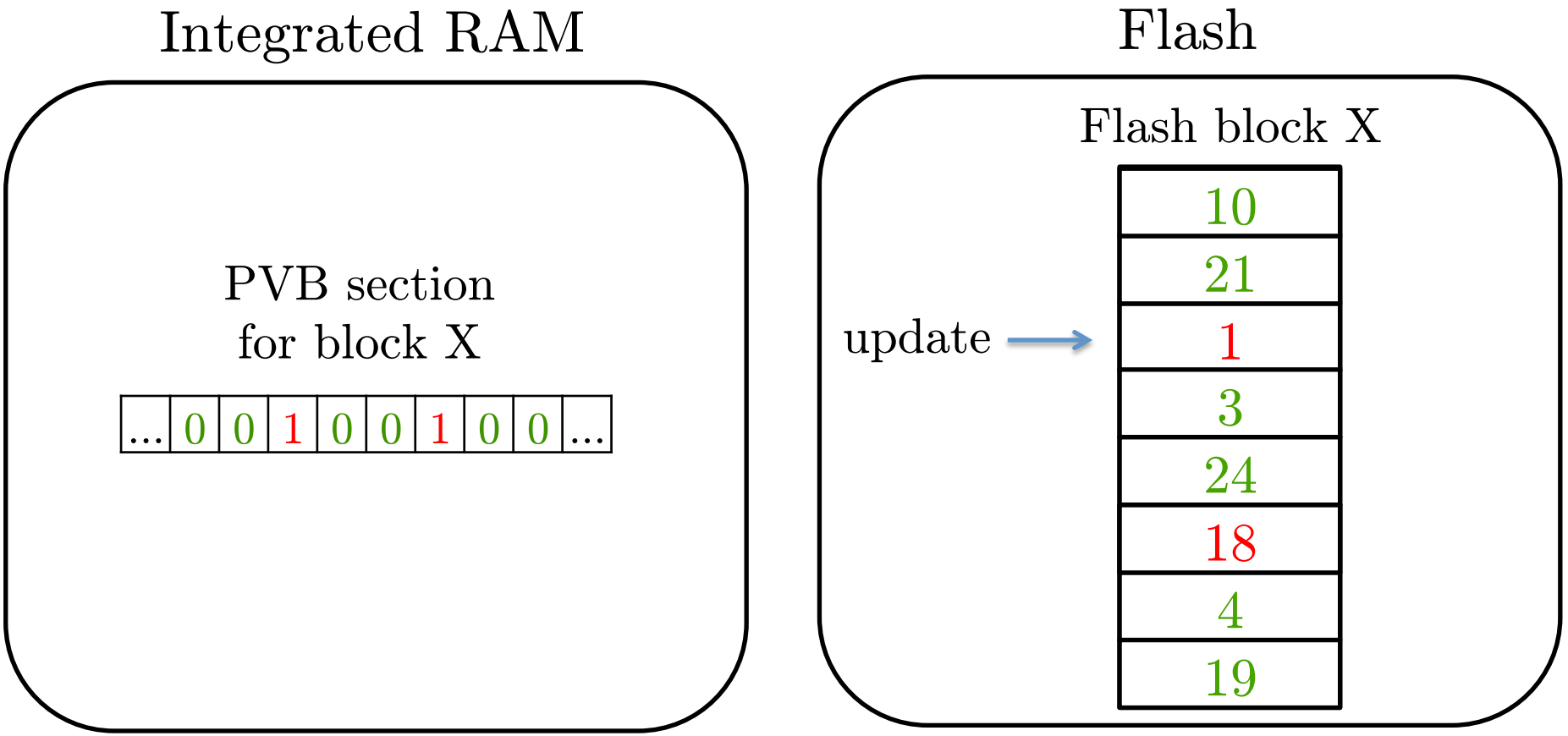
Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



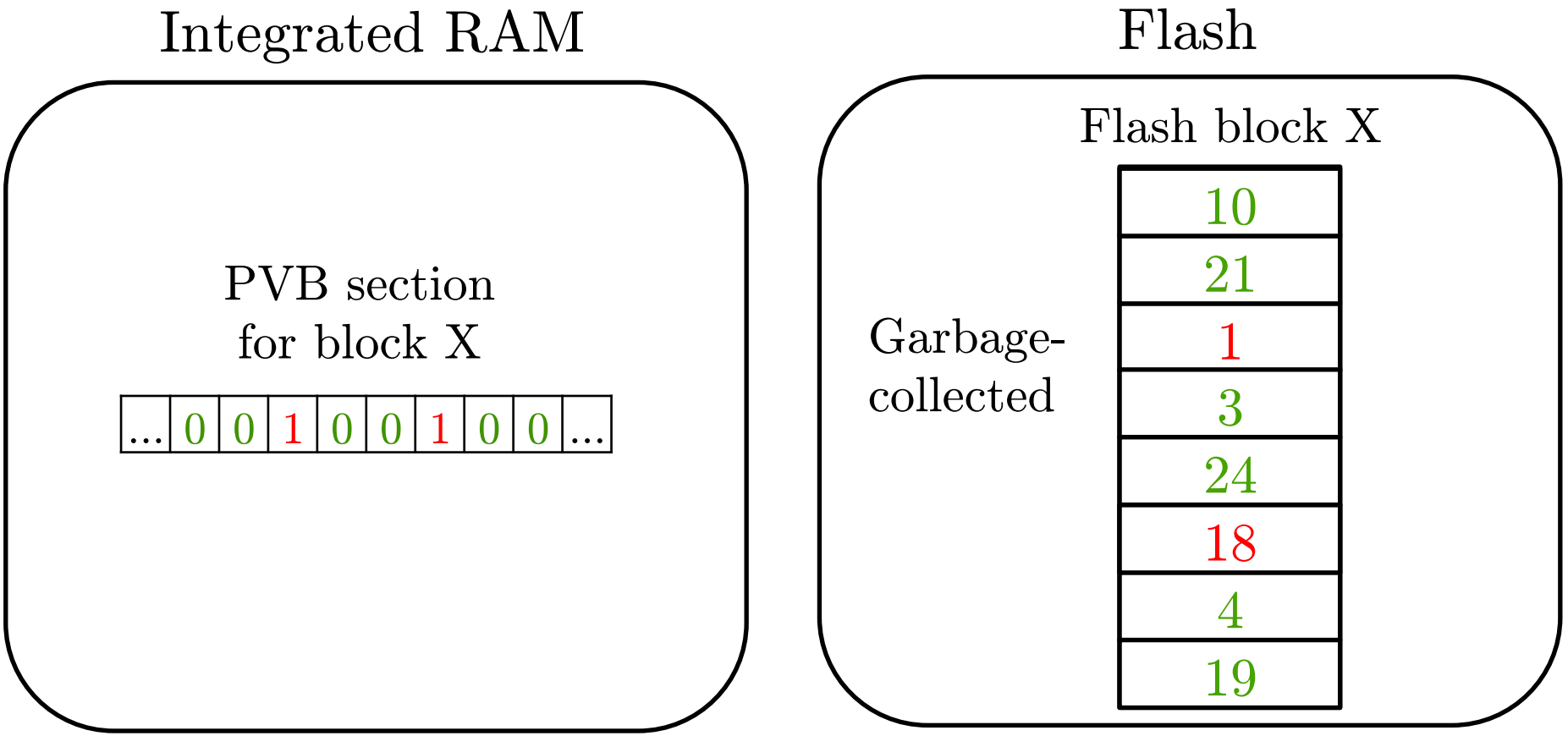
Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



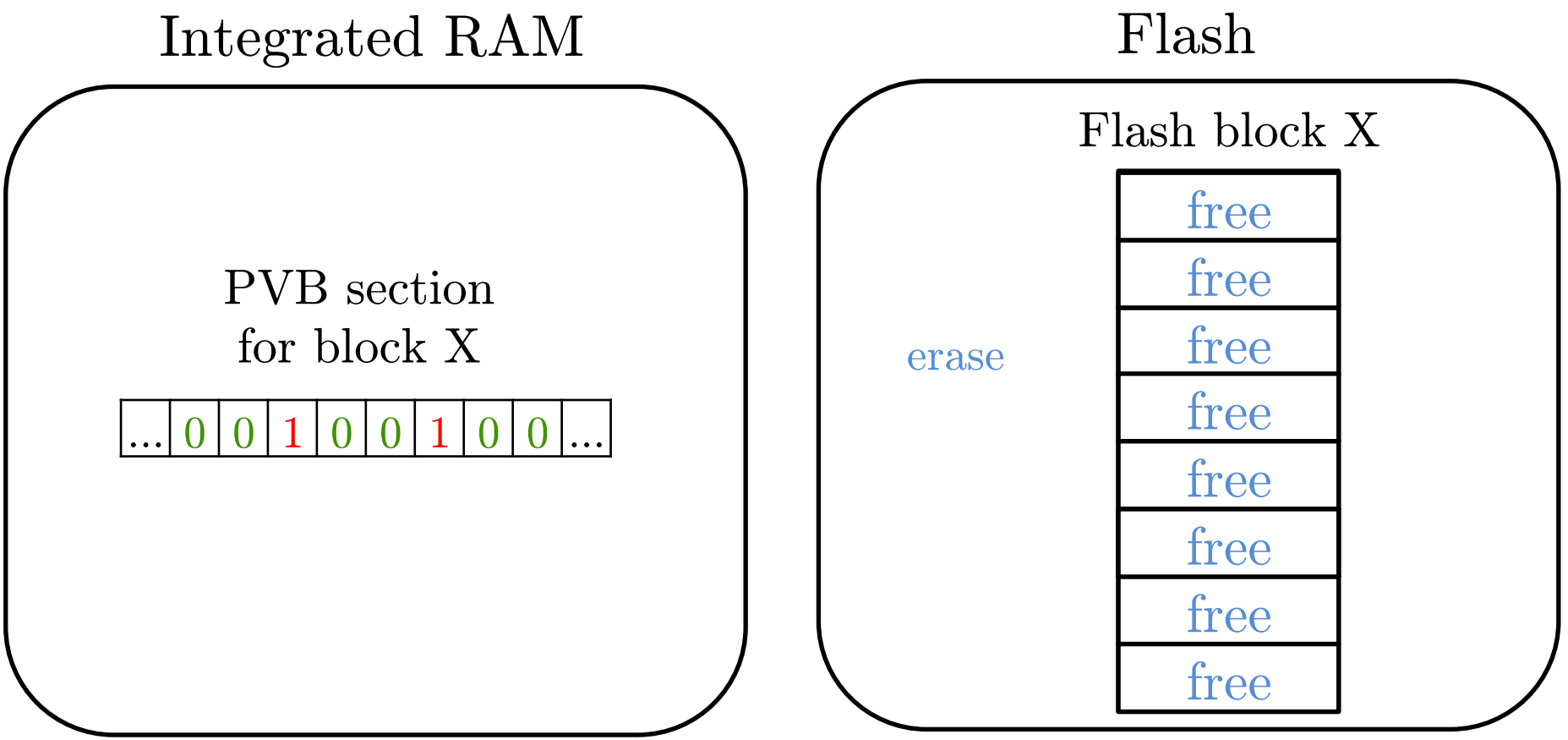
Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



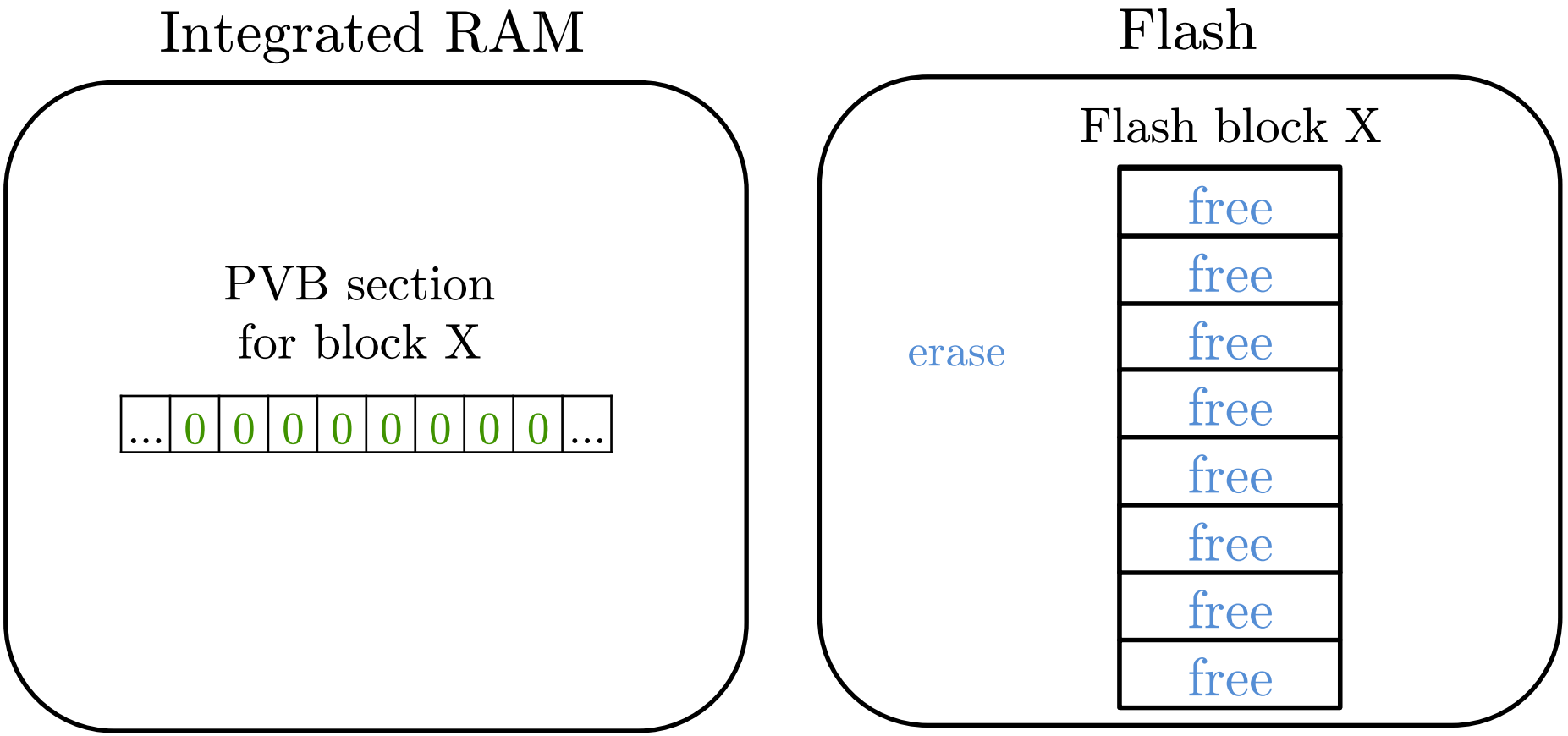
Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



Background

- PVB enables garbage-collection victim-selection
- Which pages to migrate



Background

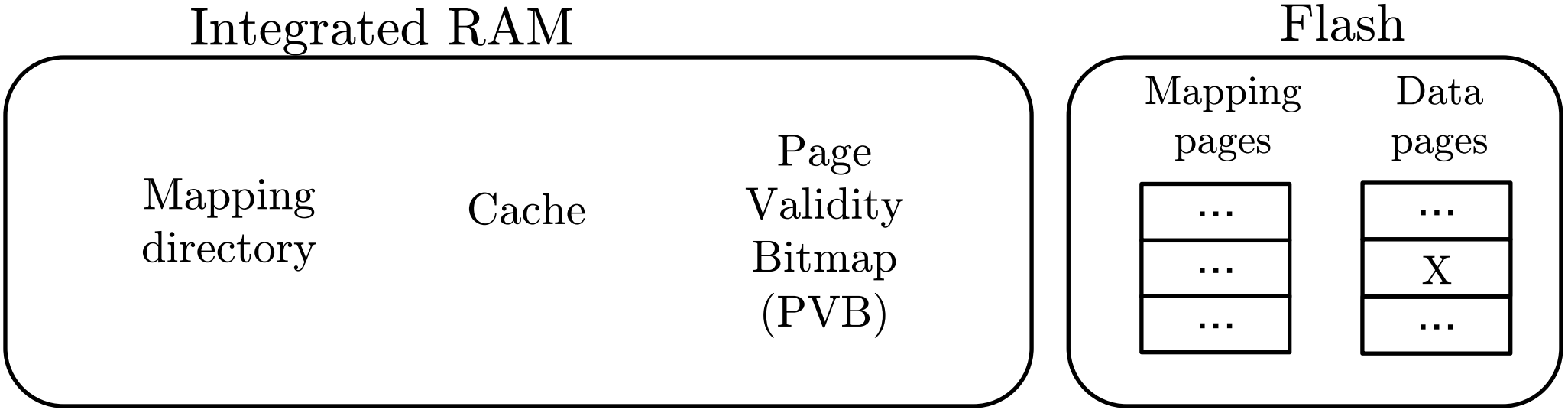
- Summary
  - Logical pages are updated out-of-place
  - Garbage-collection takes place to free space

  - A translation table keeps track of location of data
  - The translation table is stored in flash

  - PVB is used to identify live pages in GC
  - PVB is stored in integrated RAM

Background

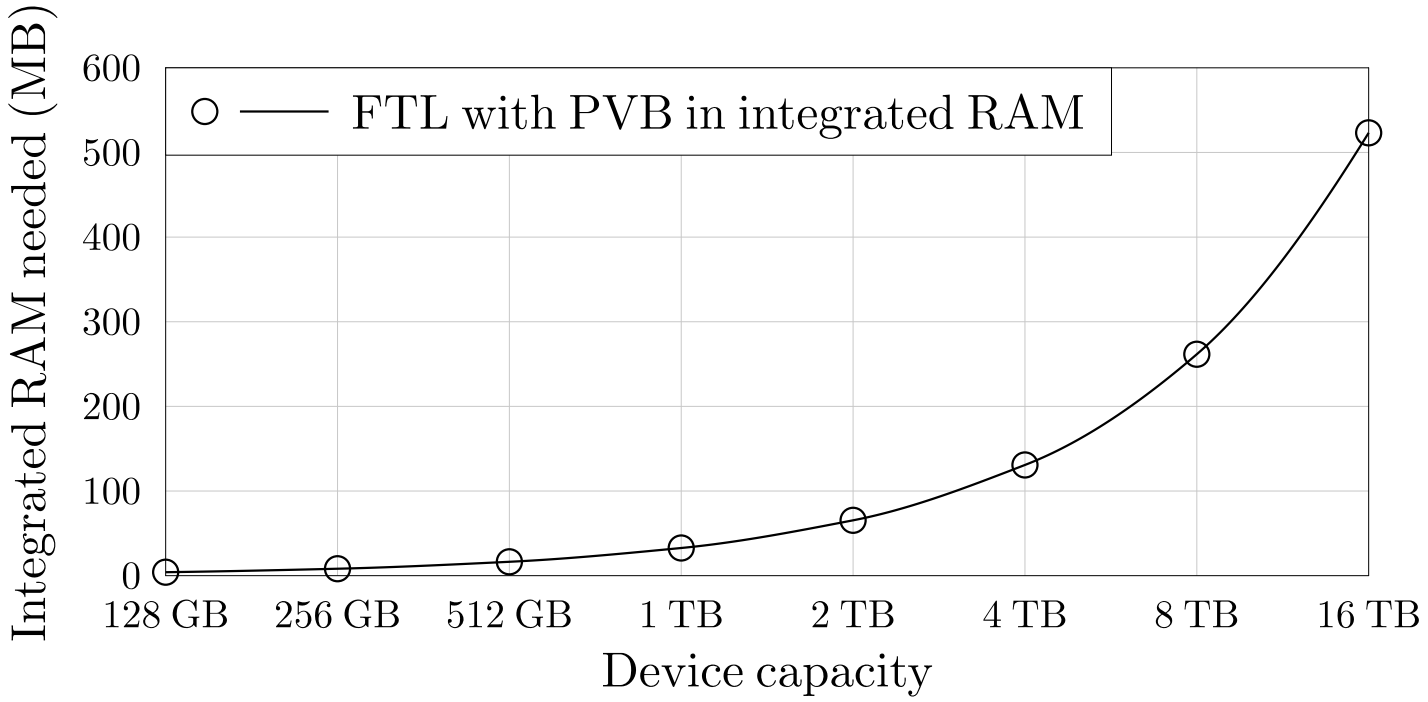
- Integrated RAM is volatile
- When power fails, we lose all RAM-resident metadata
- How can we recover PVB
- Answer: scan all mapping pages



Problem

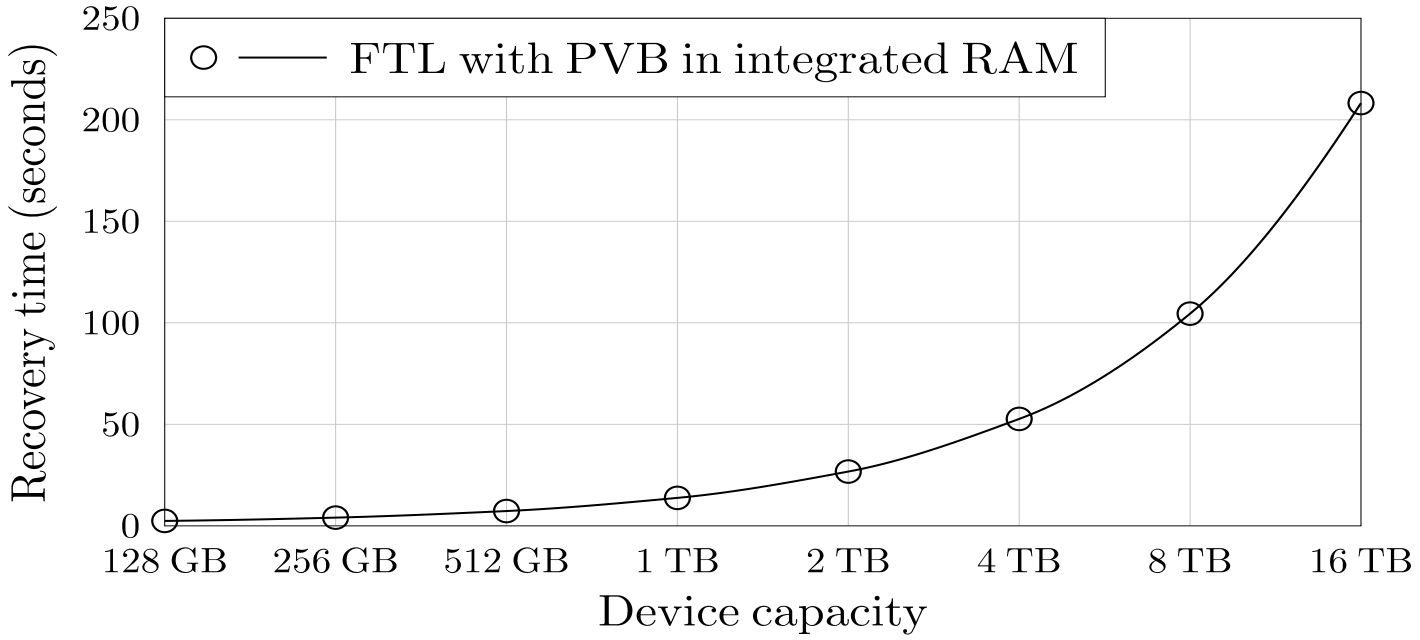
Problem

- Size of PVB is proportional to capacity
- As flash devices scale to terabytes, size of PVB is becoming impractical



Problem

- Integrated RAM is volatile
- When power fails, we lose PVB.
- Recovering PVB is too long



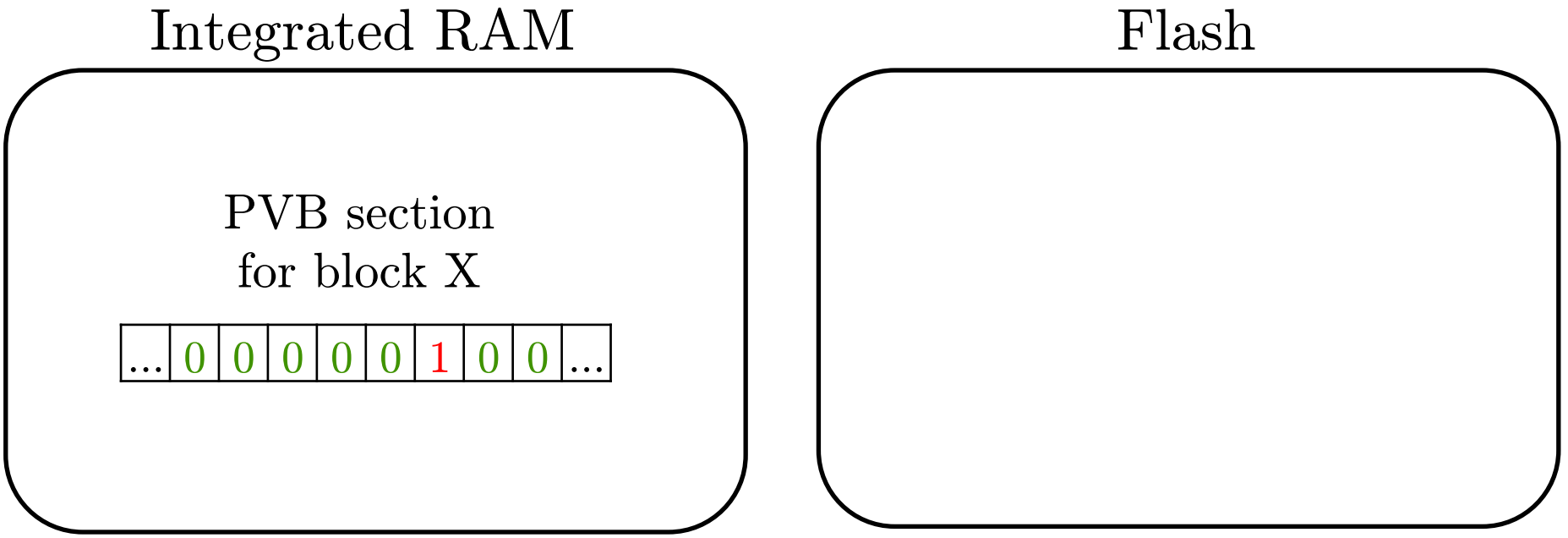
Problem

- **Observations**:
  - PVB consumes 95% of integrated RAM
  - Recovering PVB takes 51% of recovery time

- **Simple solution**
  - Store PVB in flash
  - Problem: bad for performance

Problem

- **Why?**
  - Every time a logical page is updated, we must also update PVB.
  - This doubles the number of writes



Problem

- Recall that writes are 10-100 times more expensive than reads on flash
- Writes also wear out the device

- Thus, storing PVB naively in flash:
  - halves throughput
  - halves device lifetime

Problem

- **Definition**: Write-amplification is the number of physical writes taking place internally for each logical write.

- **Goal**:
  - Store PVB in flash
  - Keep write-amplification low
  - Fast lookup time

Solution

Solution

- Insight:
  - PVB is updated once per application write
  - PVB is accessed once per garbage-collection operation
  - A garbage-collection operation happens once for every ≈100 application writes

Solution

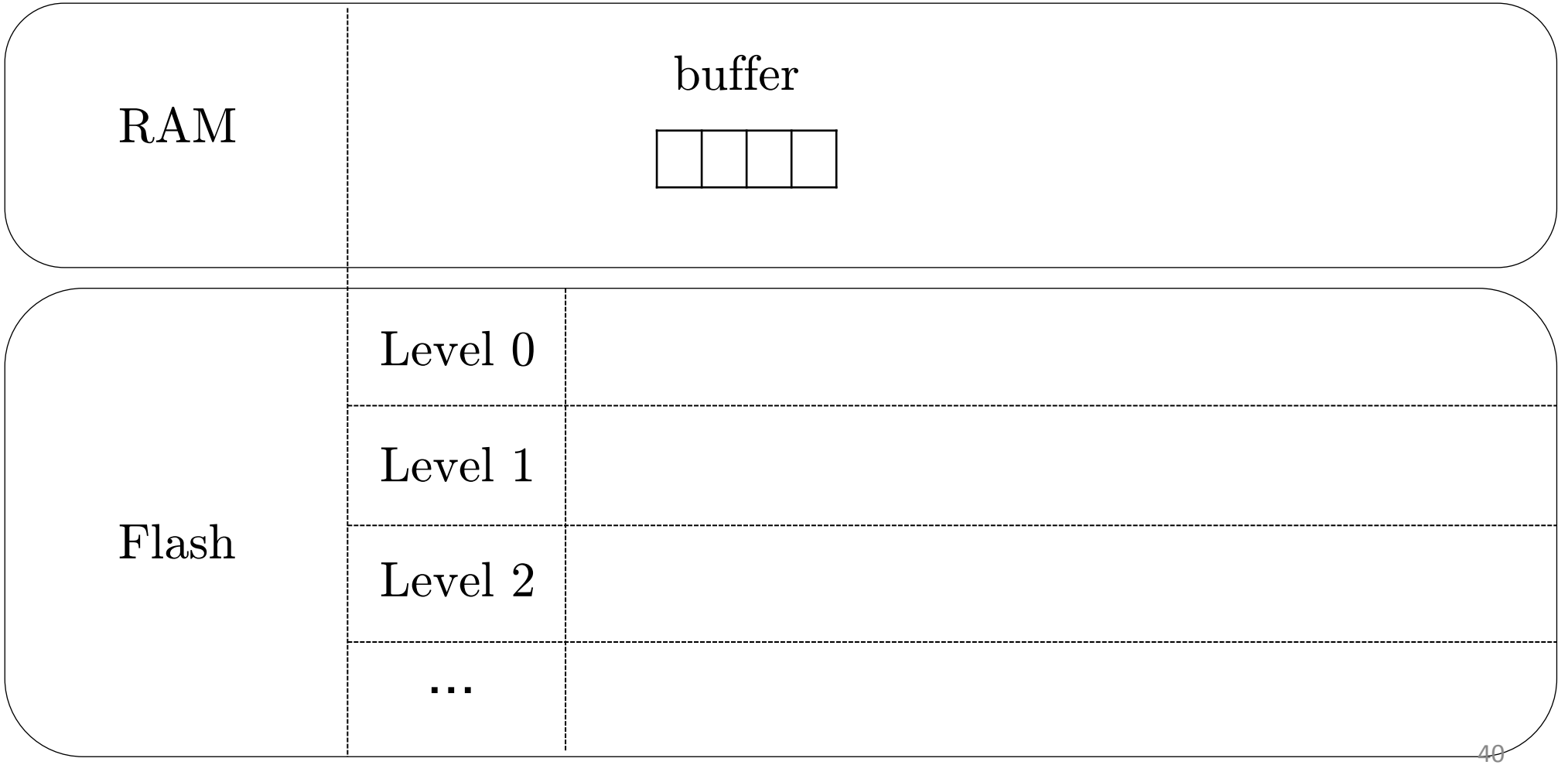
- Since PVB is mostly updated, we propose a write-optimized data structure

- We use an LSM-tree with some modifications

Solution

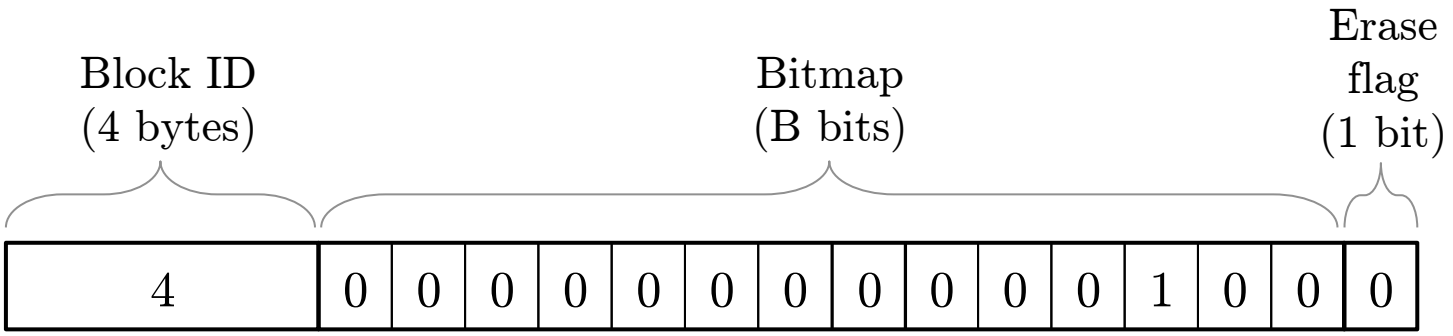
- Key ideas: buffer updates about page validity in integrated RAM
- When the buffer fills, flush to flash
- Reorganize this data in flash periodically to keep access time fast

Solution



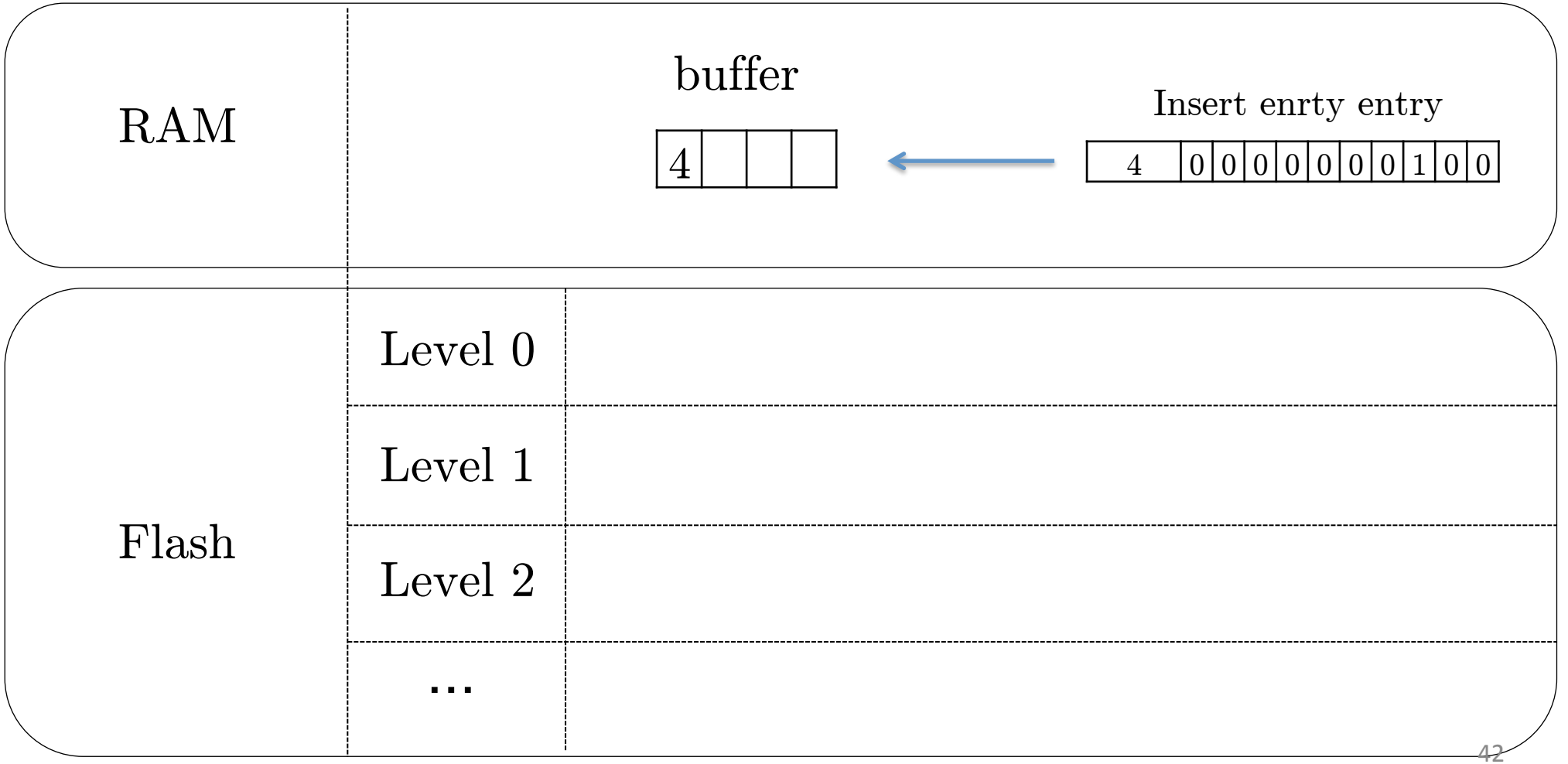
Solution

- When a page is invalidated, create entry:
  - Block ID that invalidated page is in
  - Bitmap with B bits set to 0
  - Mark invalidated page as 1

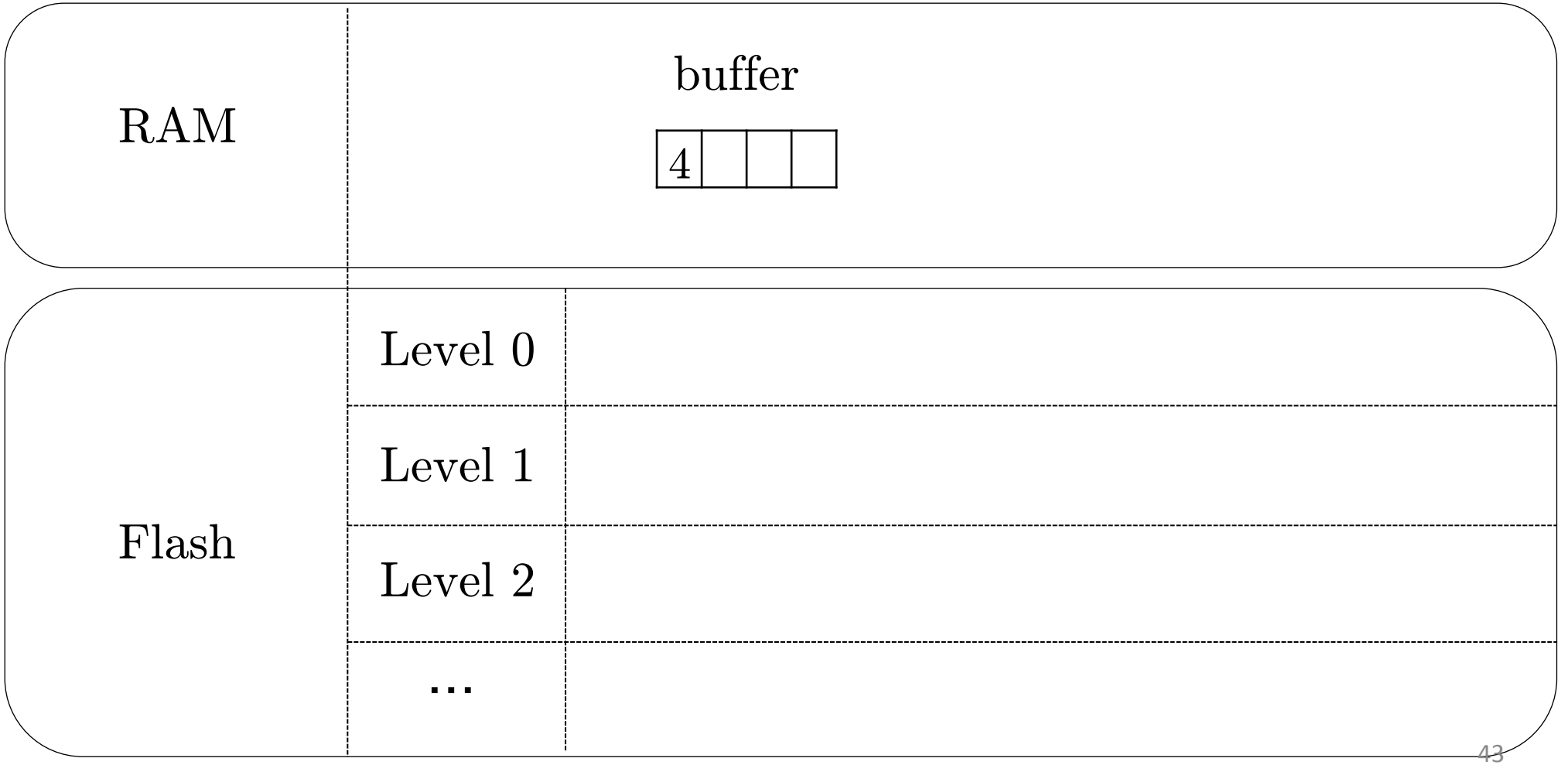


  - Entry is inserted into the buffer

Solution

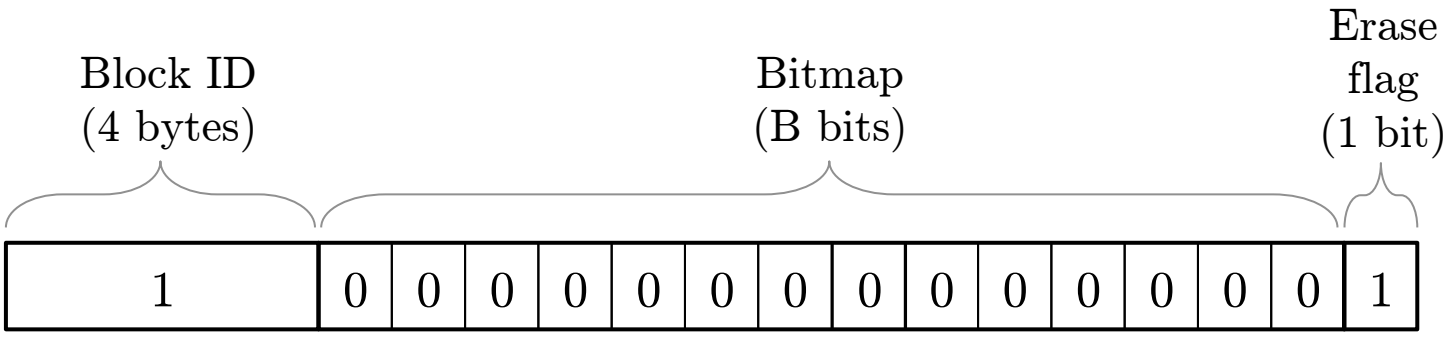

RAM
buffer
4
Insert enrty entry
4 0 0 0 0 0 0 0 1 0 0
Flash
Level 0
Level 1
Level 2
...

Solution



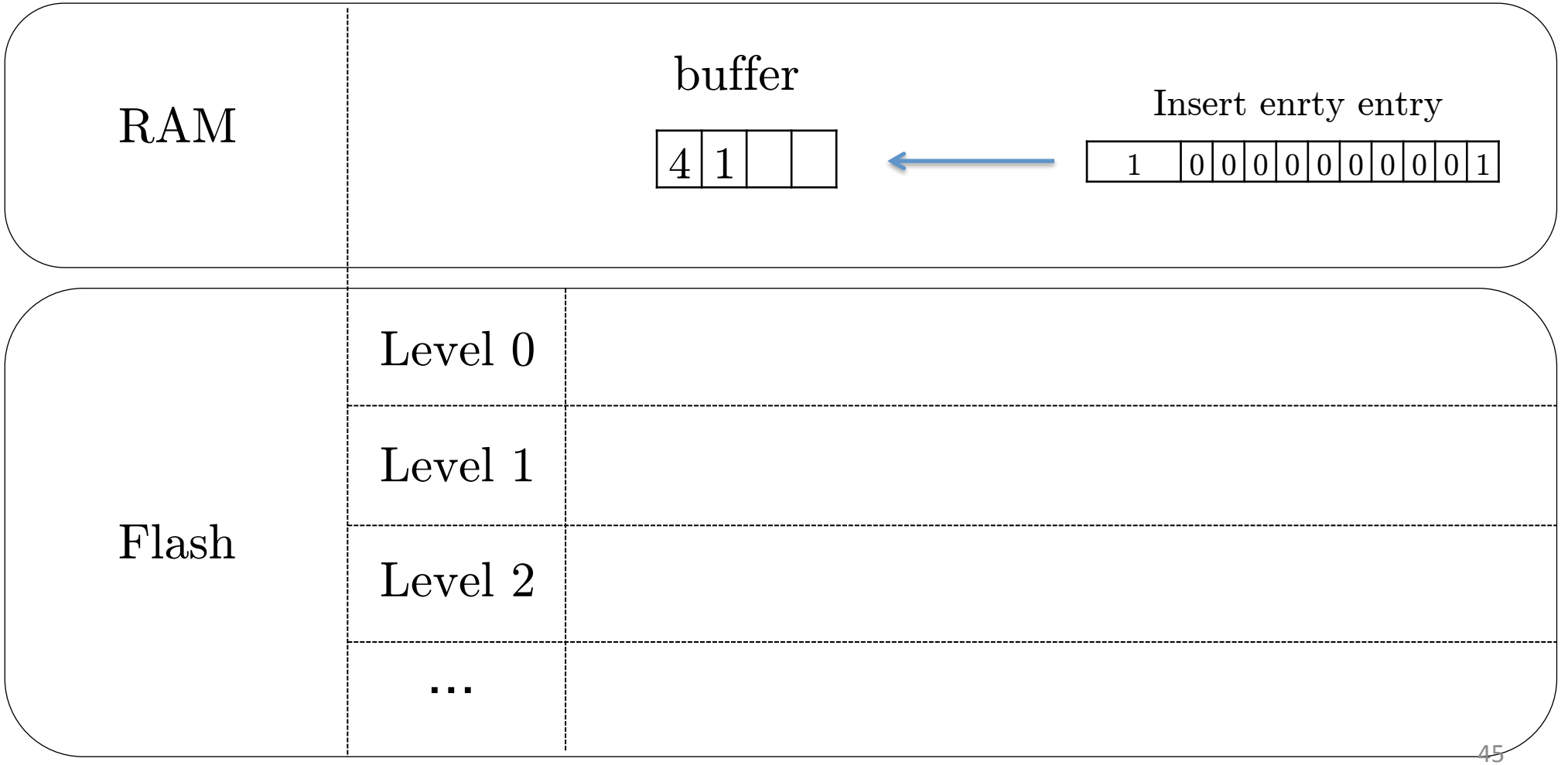
Solution

- When a block is erased, create an entry
  - Block ID of erased block
  - All bits set to 0
  - Erase flag set to 1

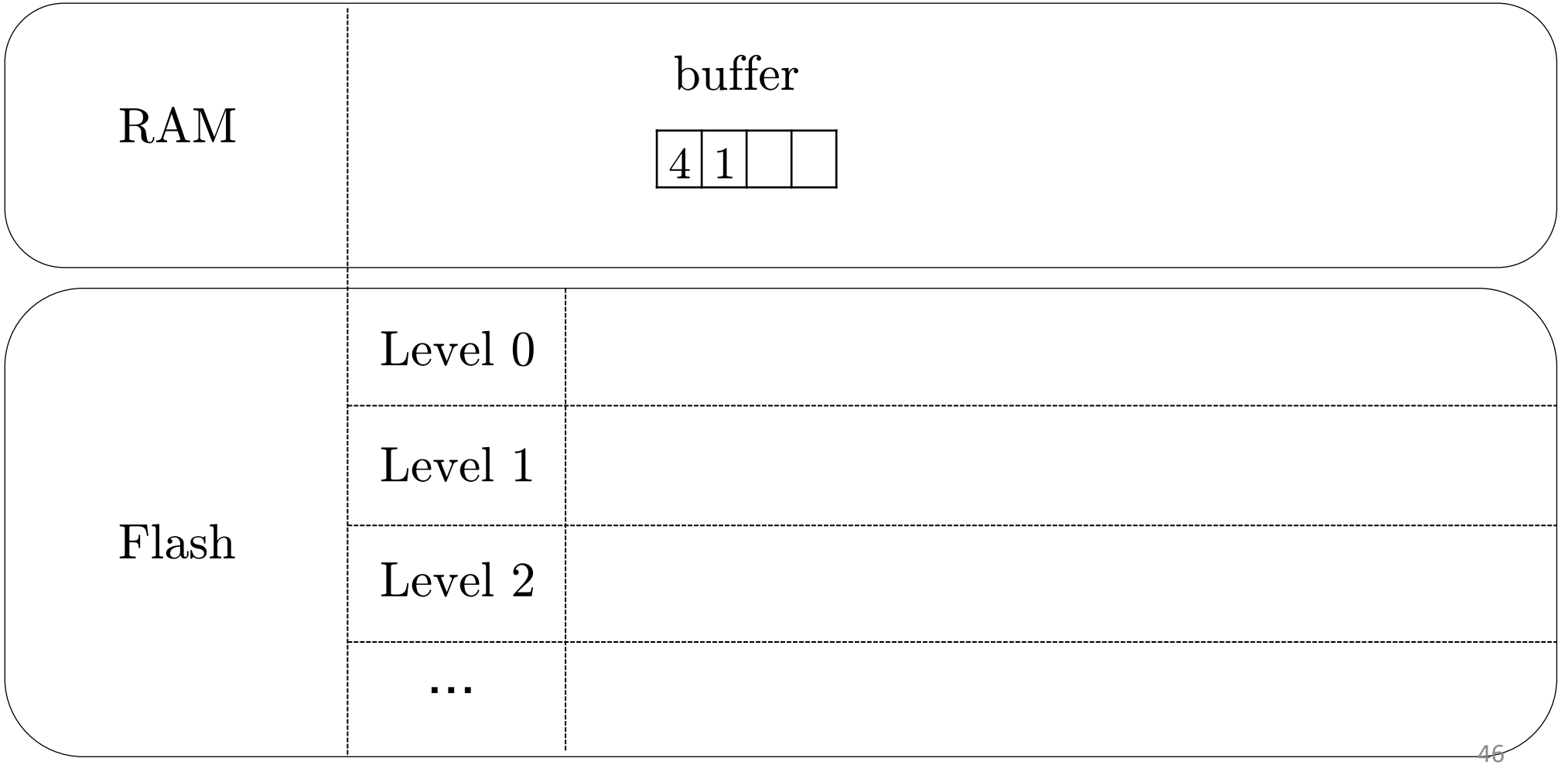


  - Entry is inserted into the buffer

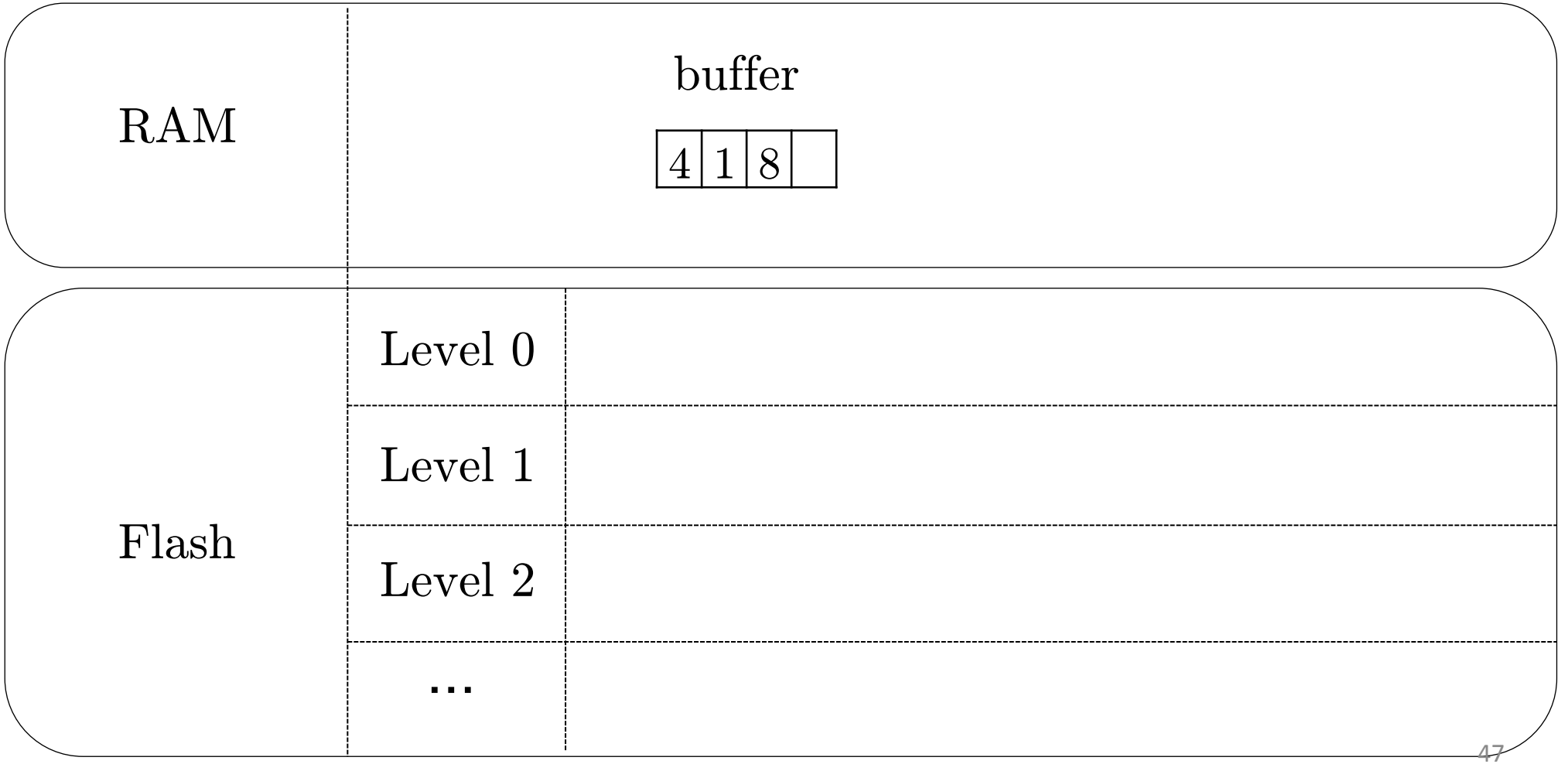
Solution


RAM
buffer
4 1
Insert enrty entry
1 0 0 0 0 0 0 0 0 0 1
Flash
Level 0
Level 1
Level 2
...

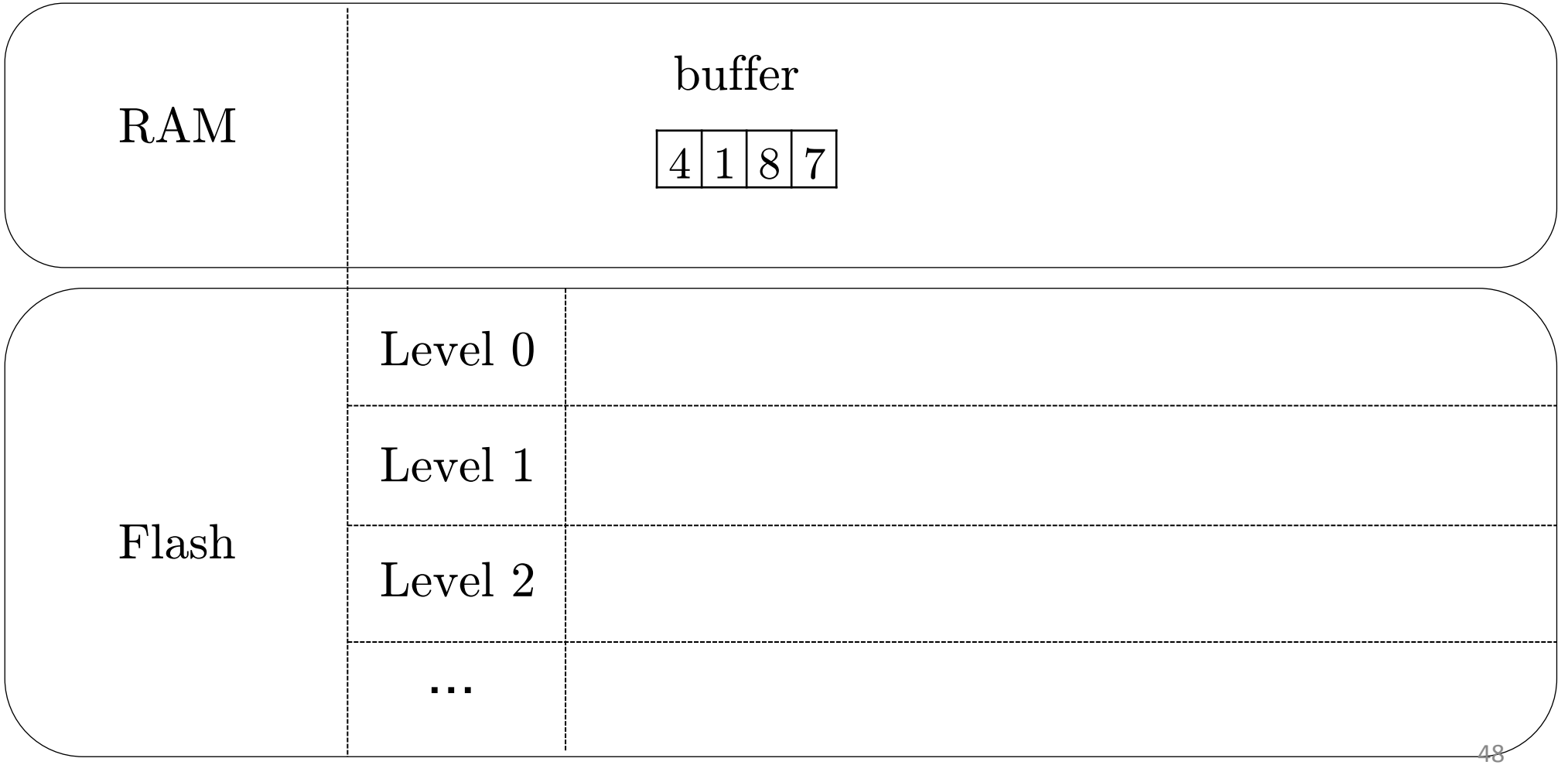
Solution



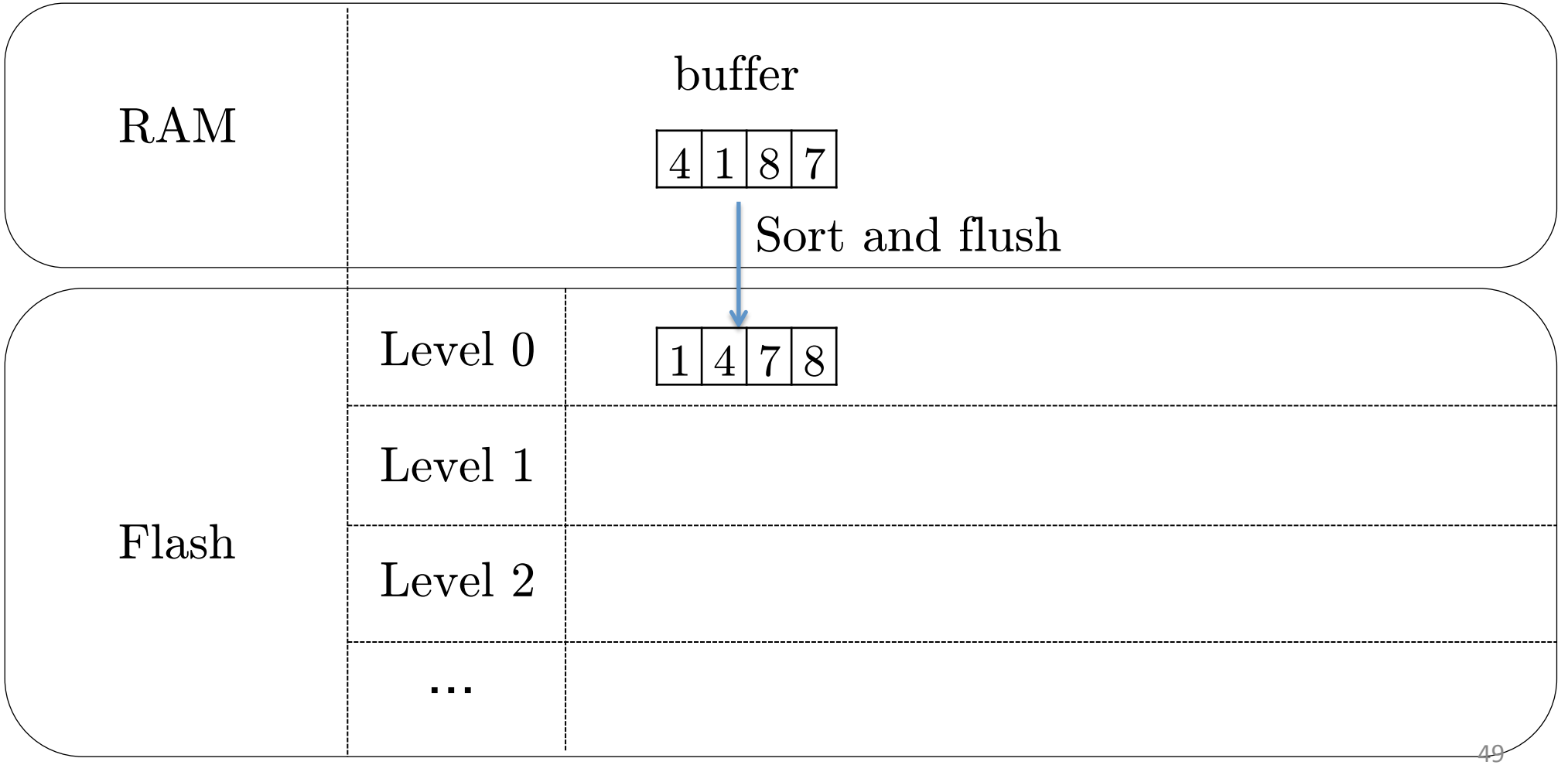
Solution



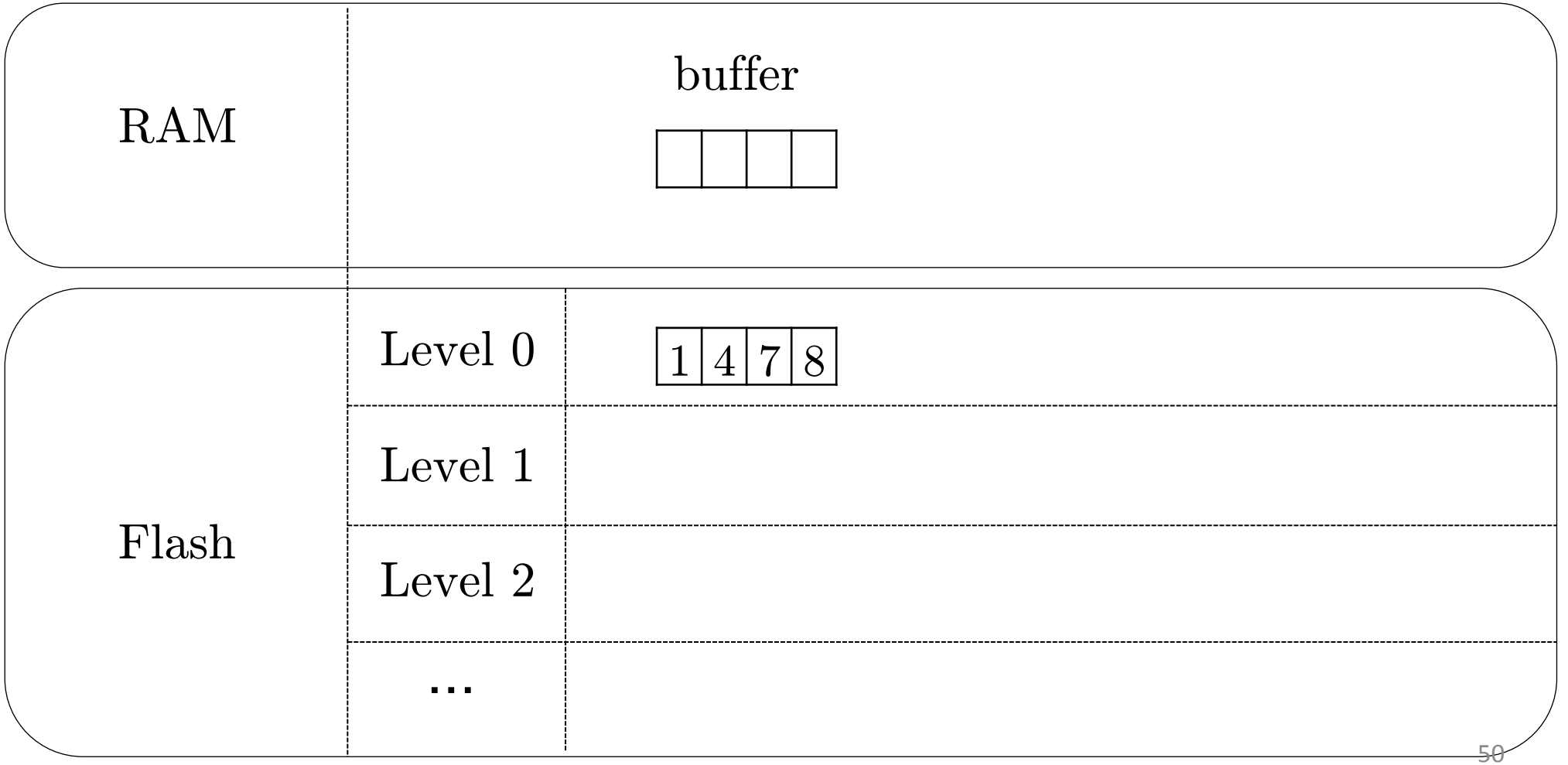
Solution



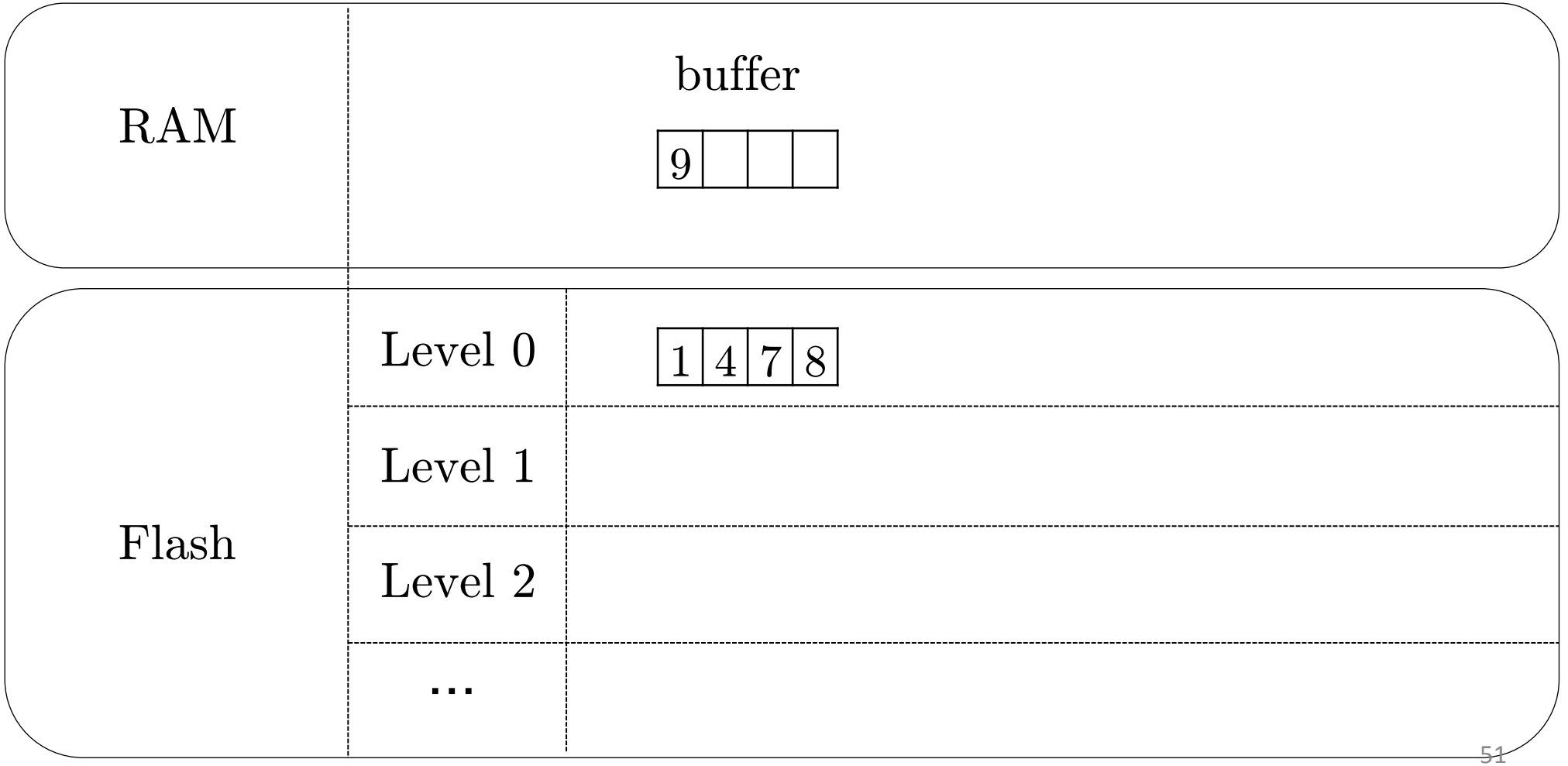
Solution


RAM
buffer
4 1 8 7
Sort and flush
Flash
Level 0
1 4 7 8
Level 1
Level 2
...

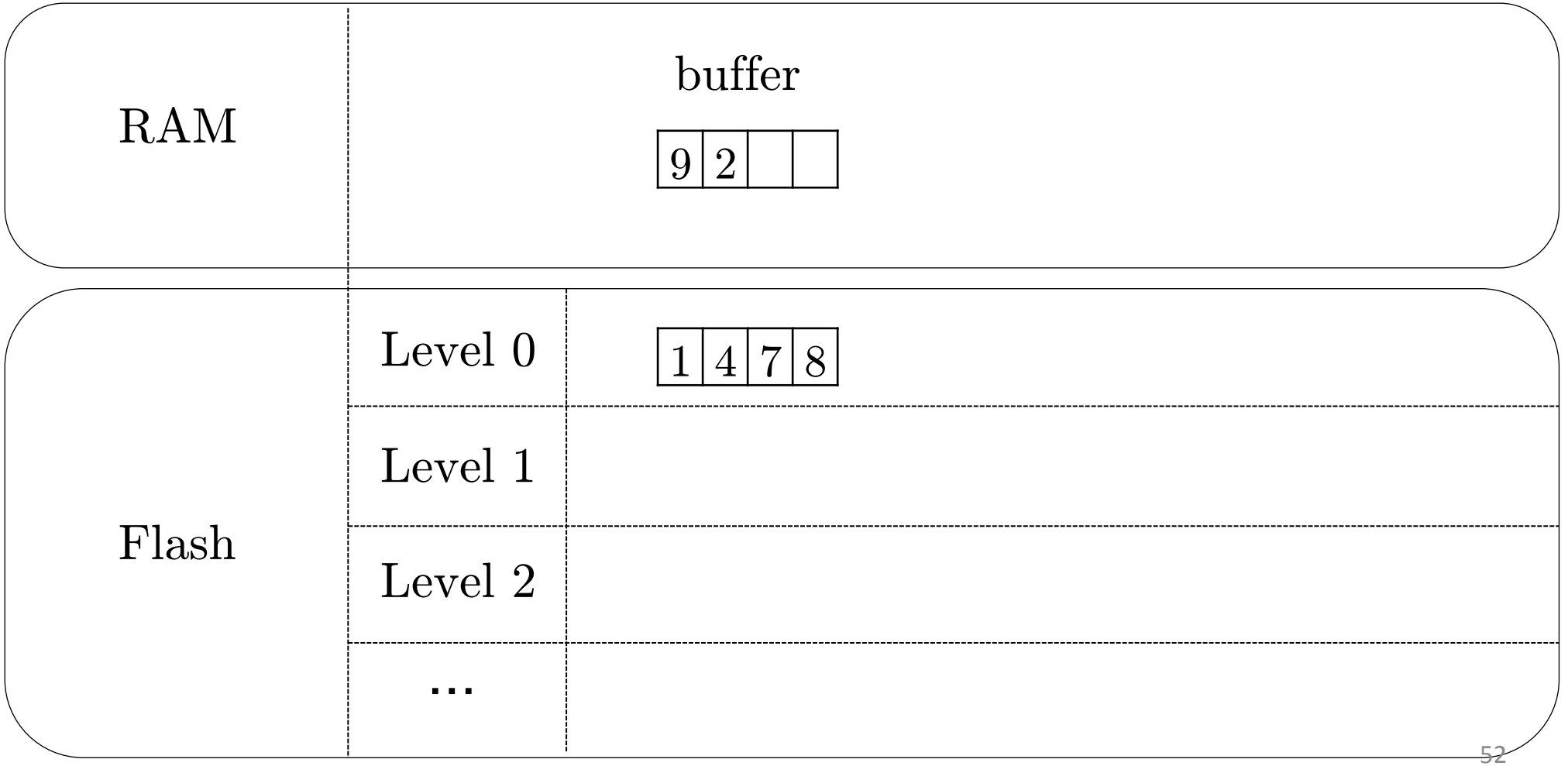
Solution



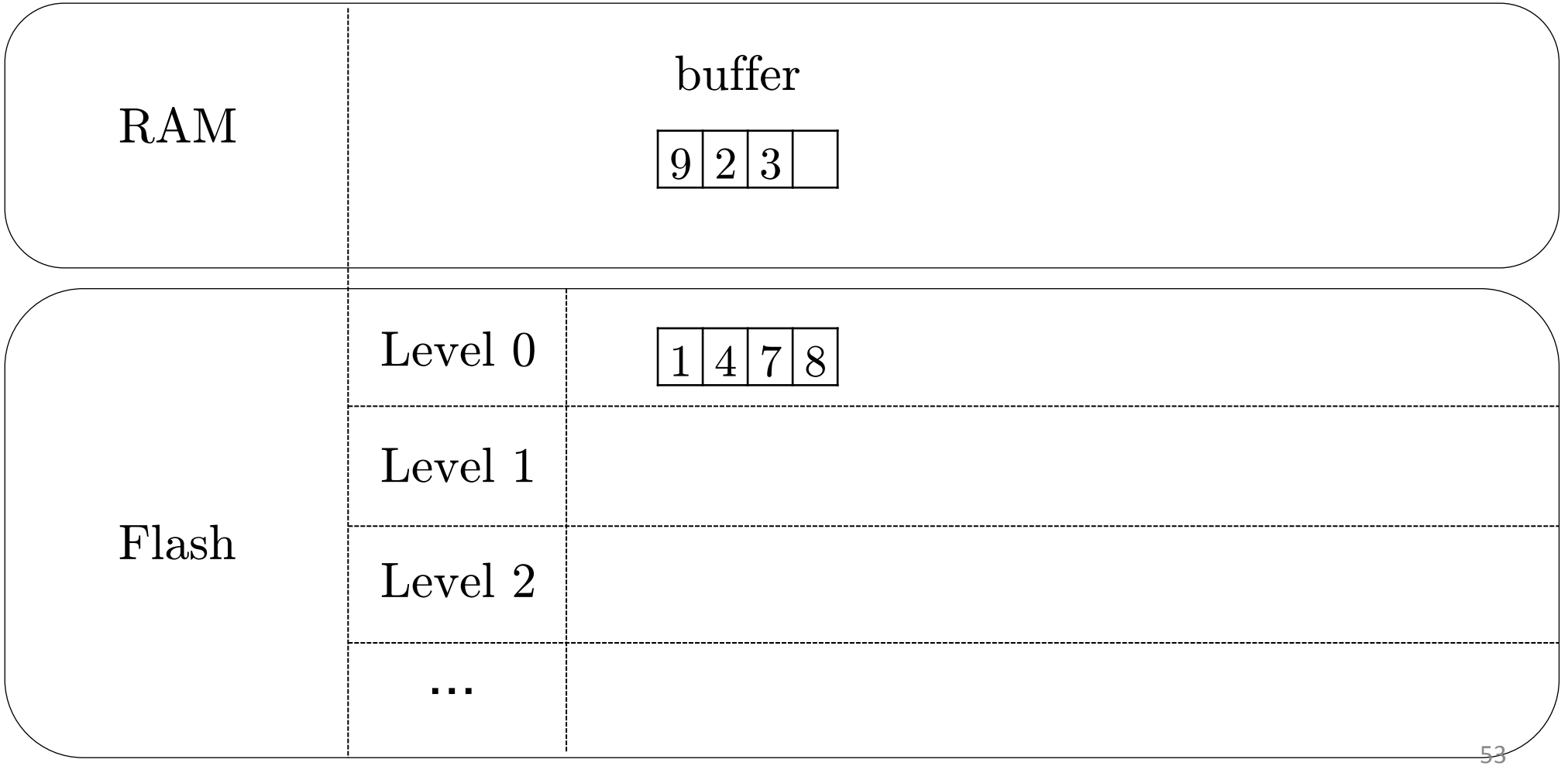
Solution



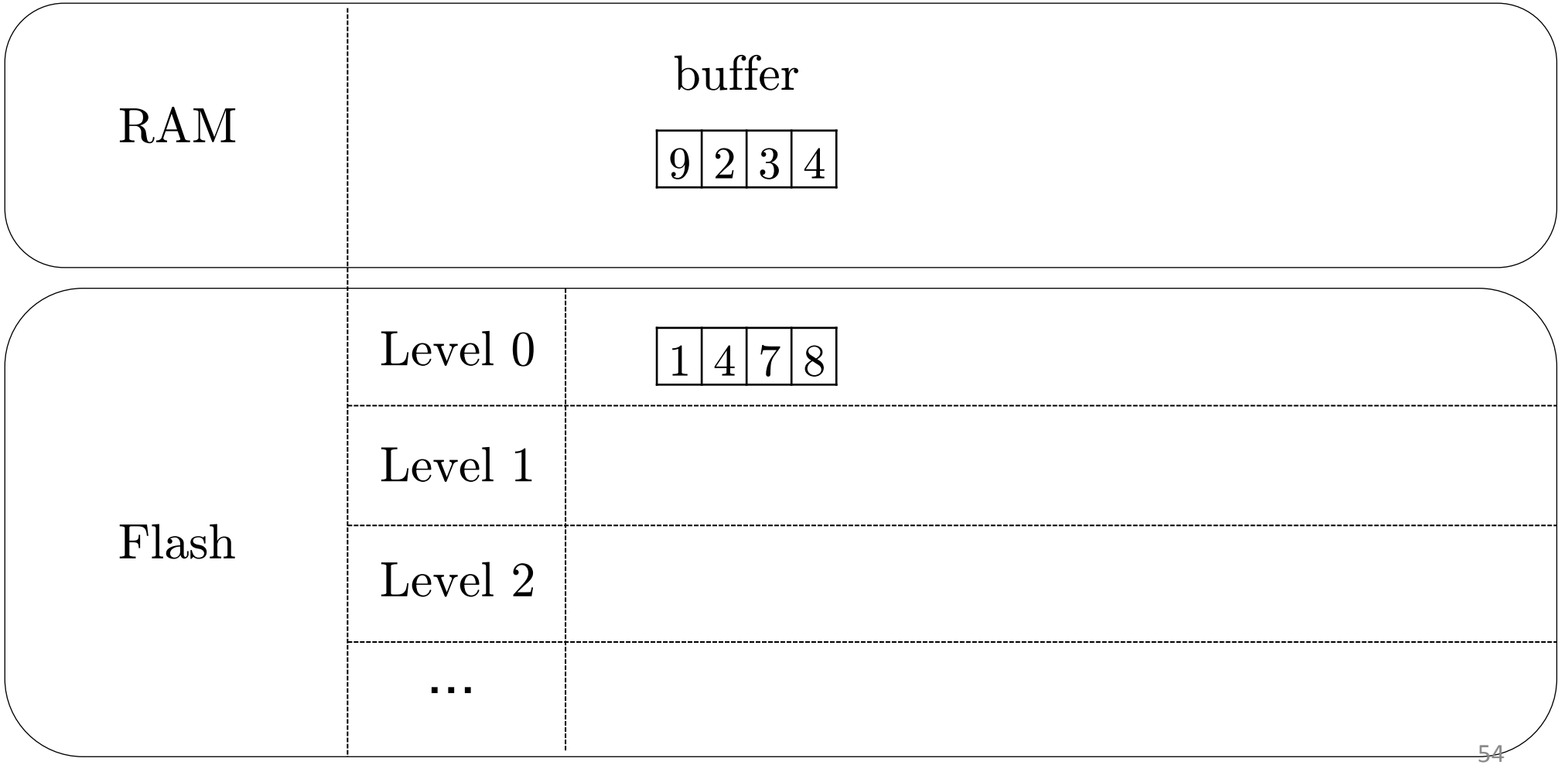
Solution



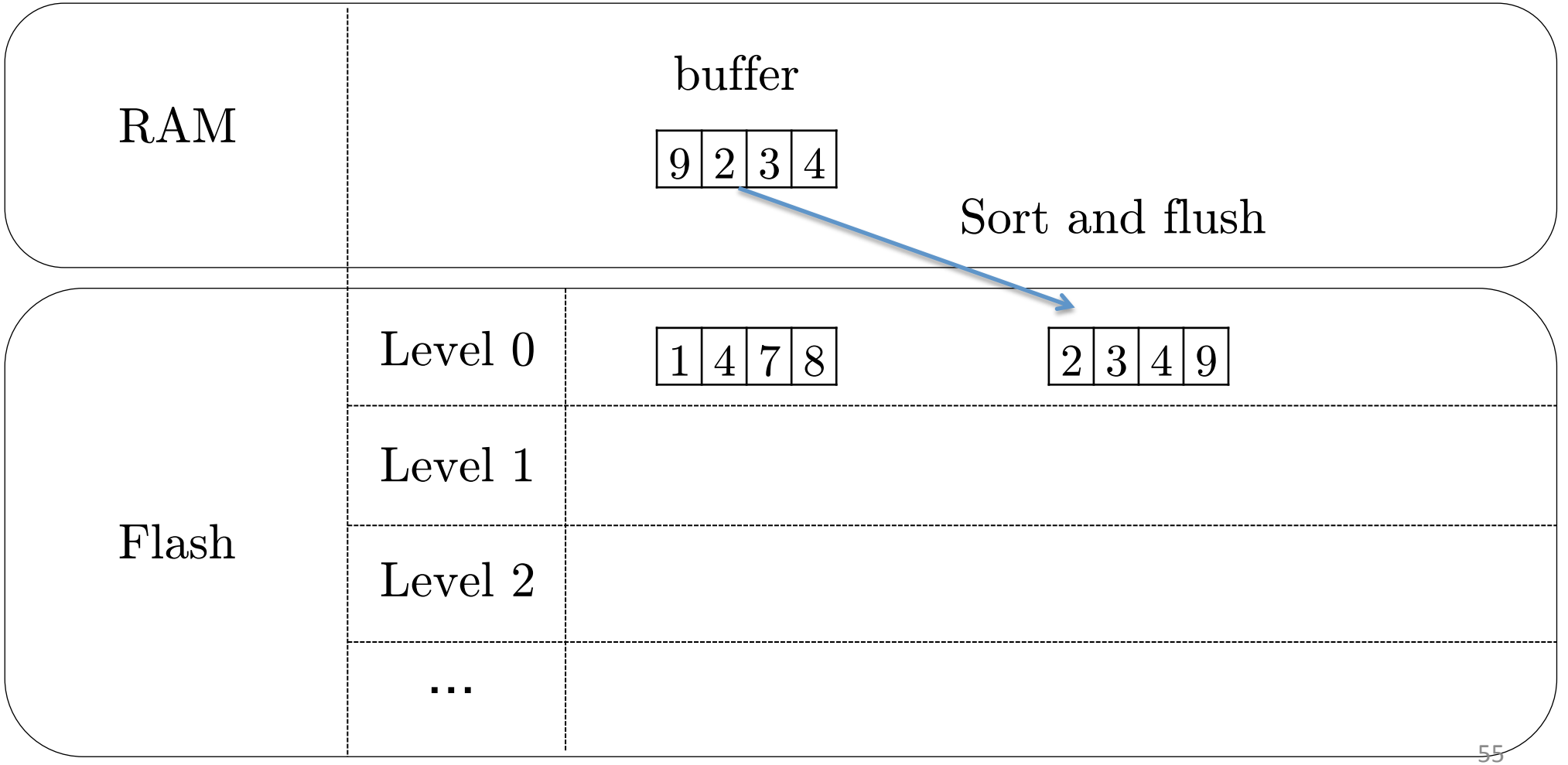
Solution



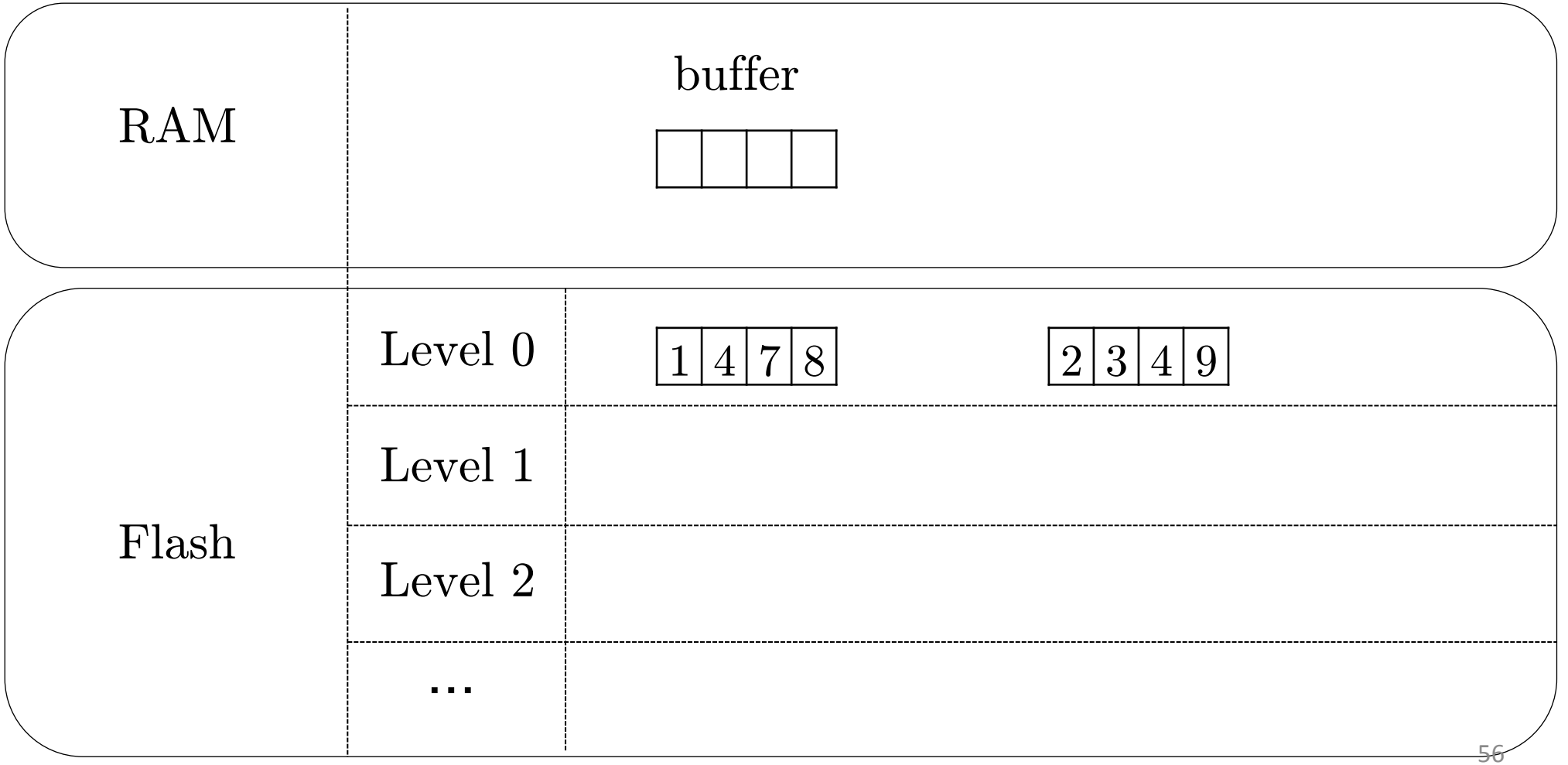
Solution



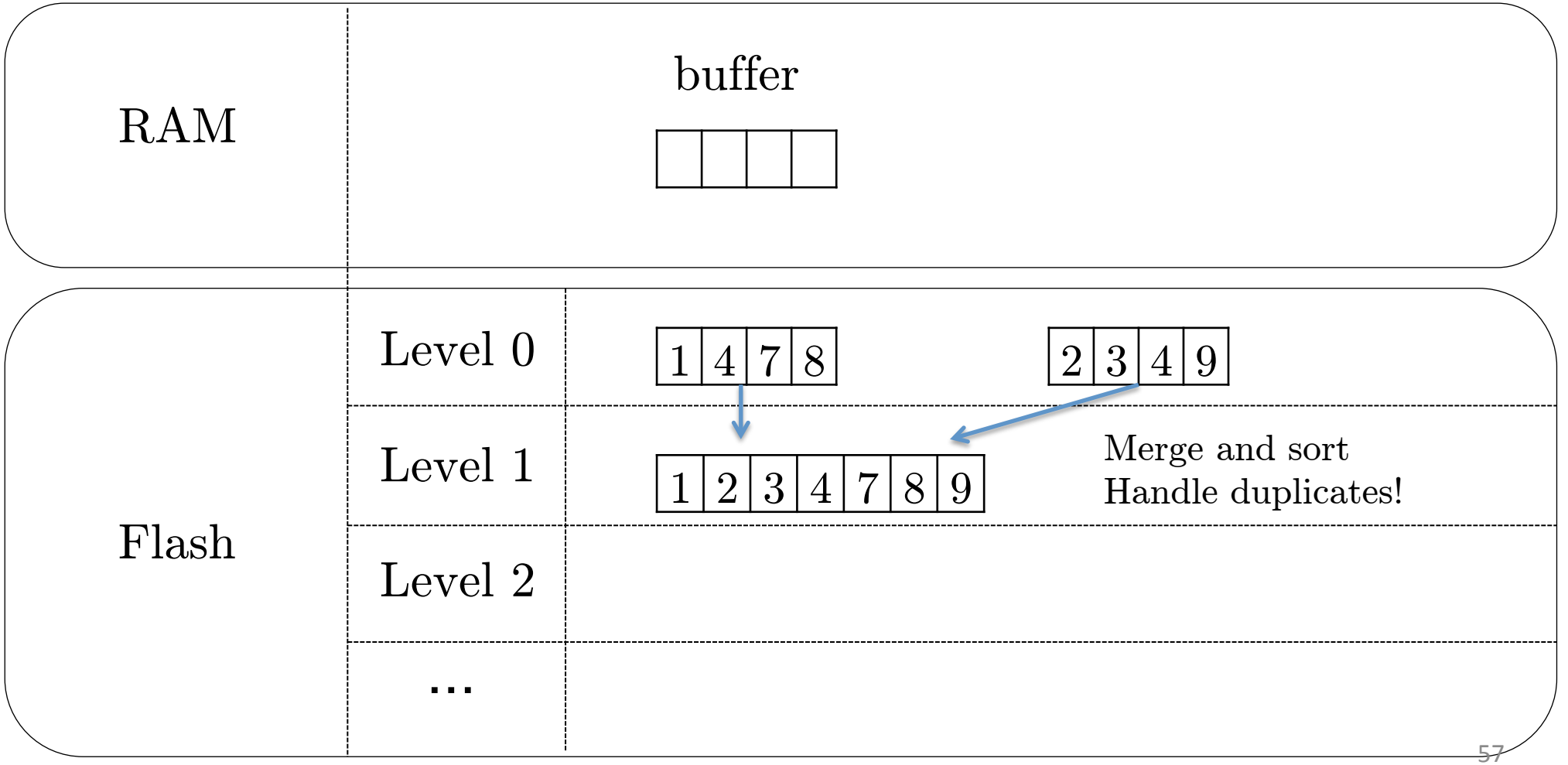
Solution


RAM
buffer
9 2 3 4
Sort and flush
Flash
Level 0
1 4 7 8
2 3 4 9
Level 1
Level 2
...

Solution

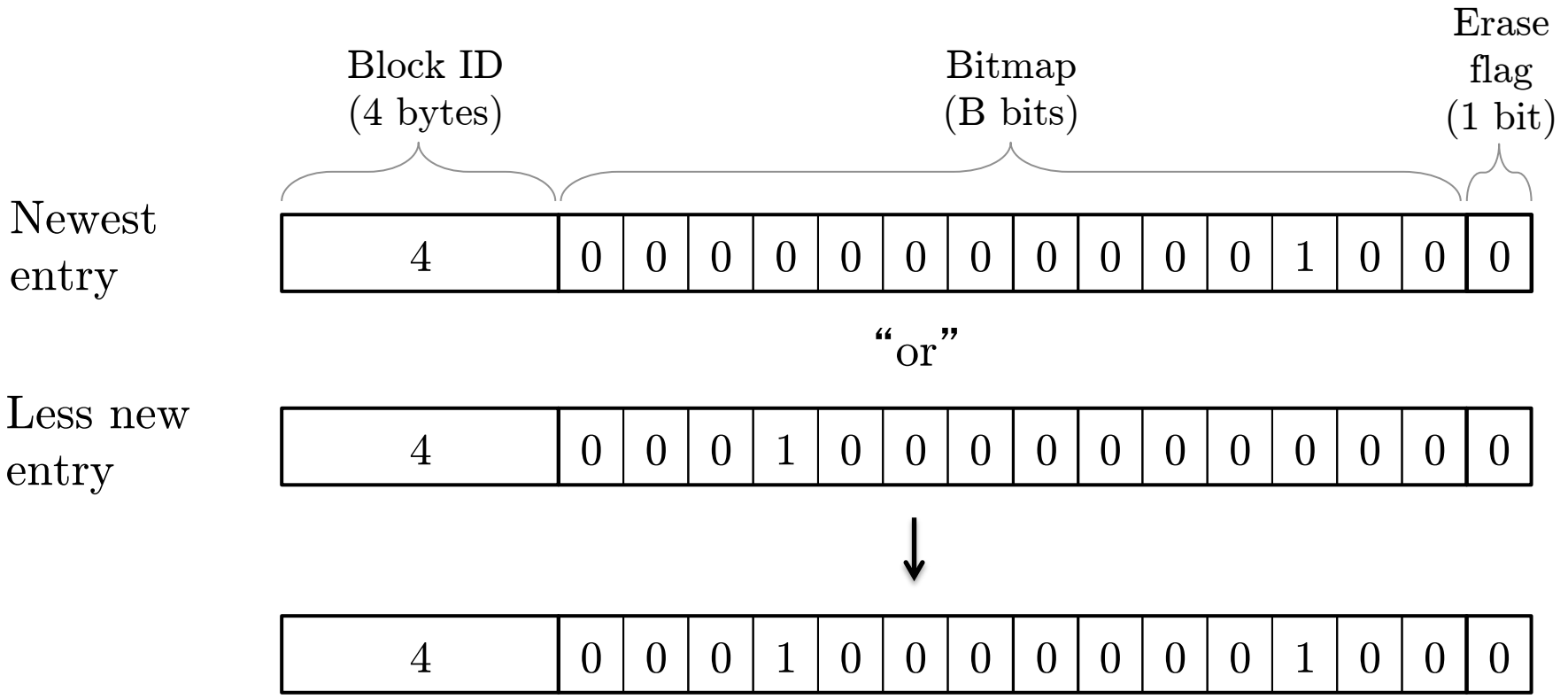


Solution


RAM
buffer
Flash
Level 0
1 4 7 8
2 3 4 9
Level 1
1 2 3 4 7 8 9
Merge and sort
Handle duplicates!
Level 2
...

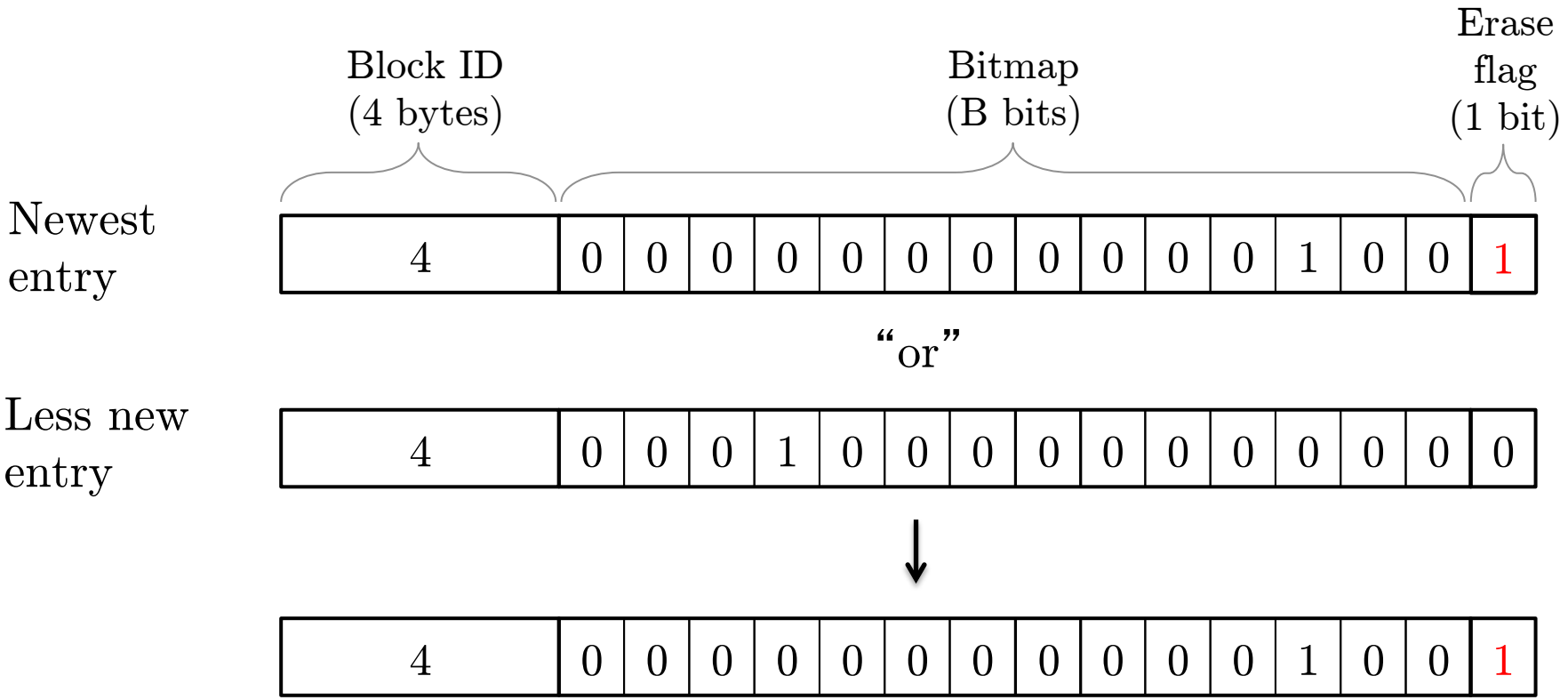
Solution

- Merge bitmaps using bitwise “or” operation

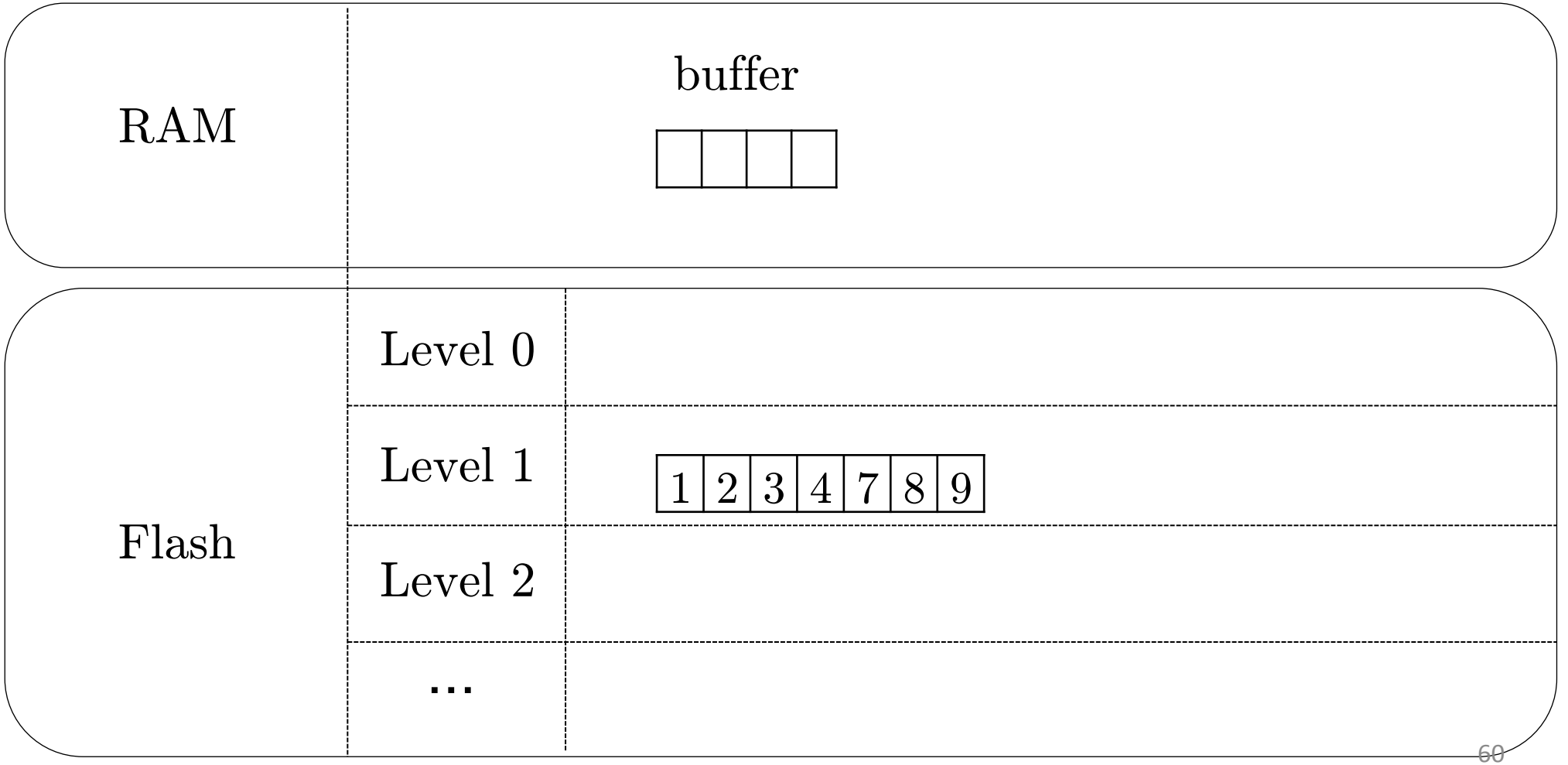


Solution

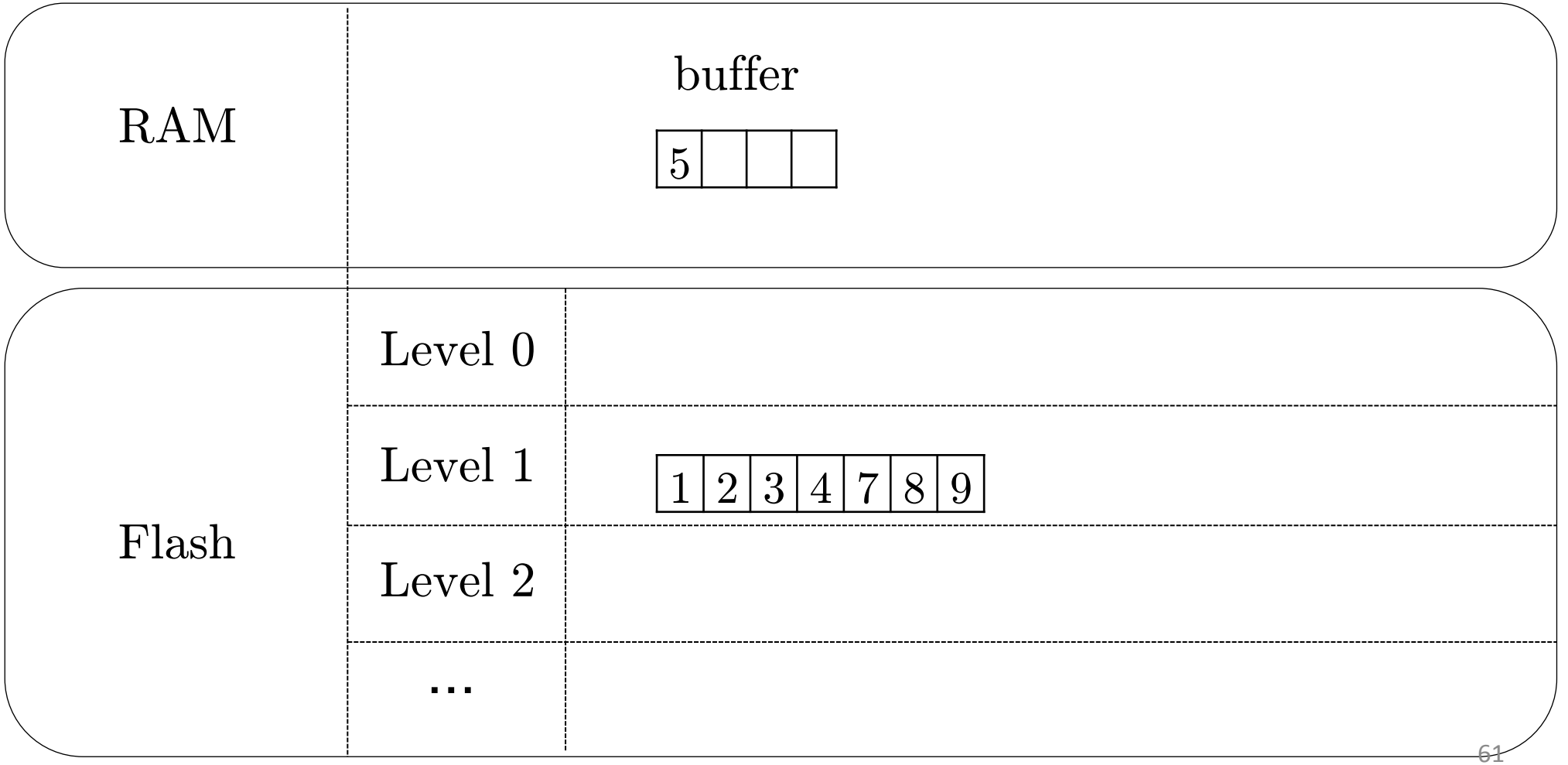
- However, if erase flag of more recent entry is 1, ignore less recent entry



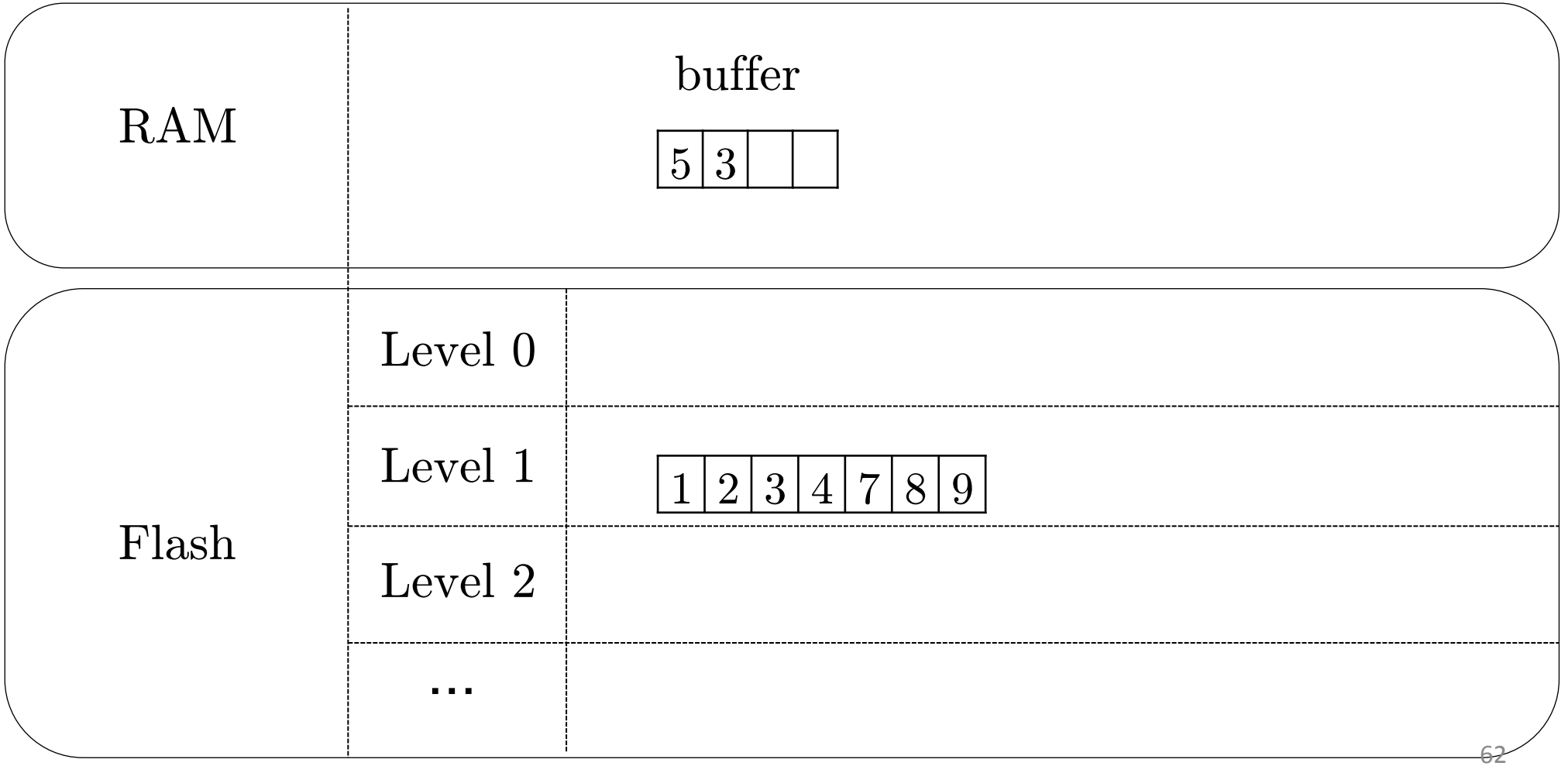
Solution



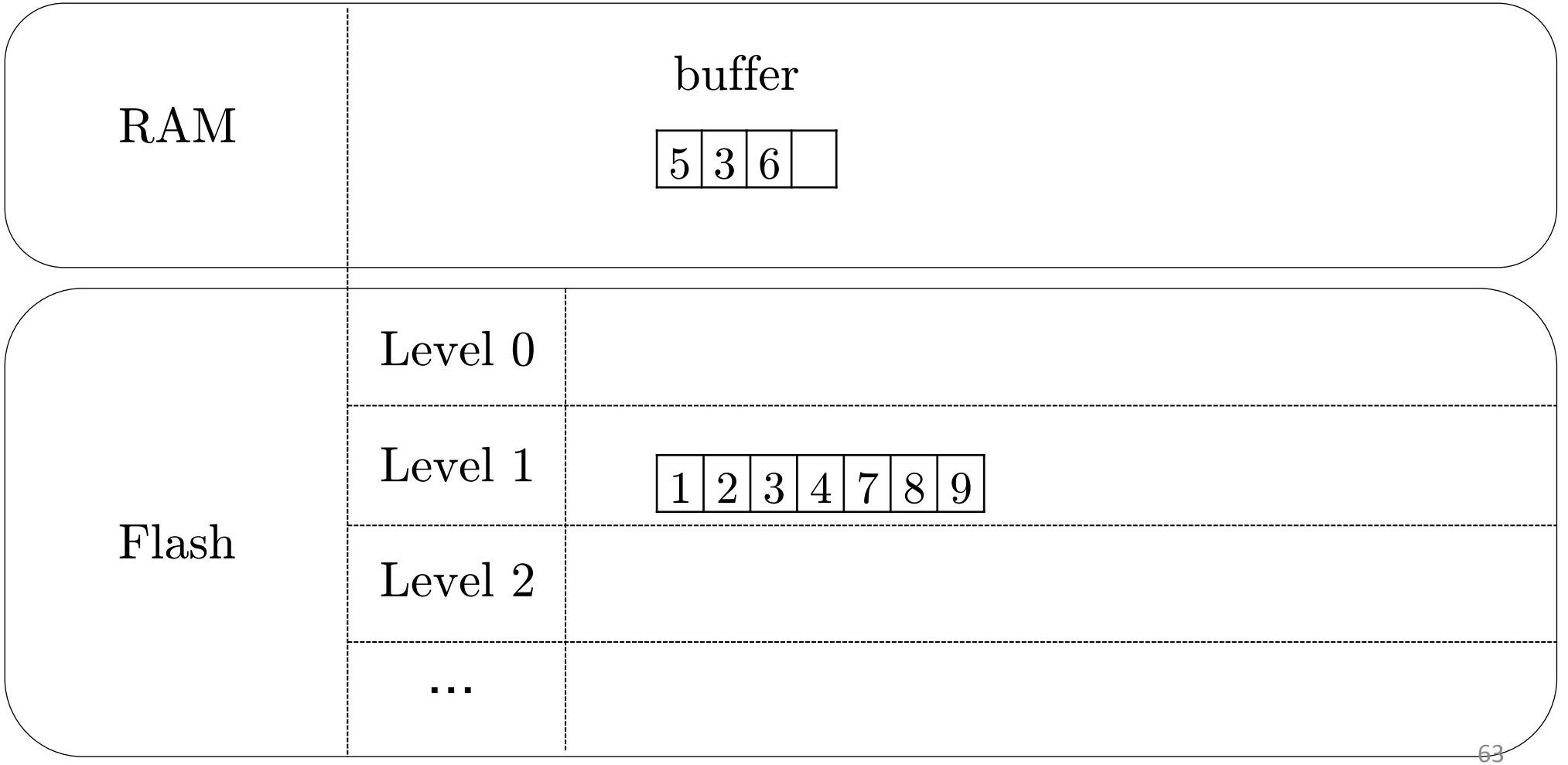
Solution



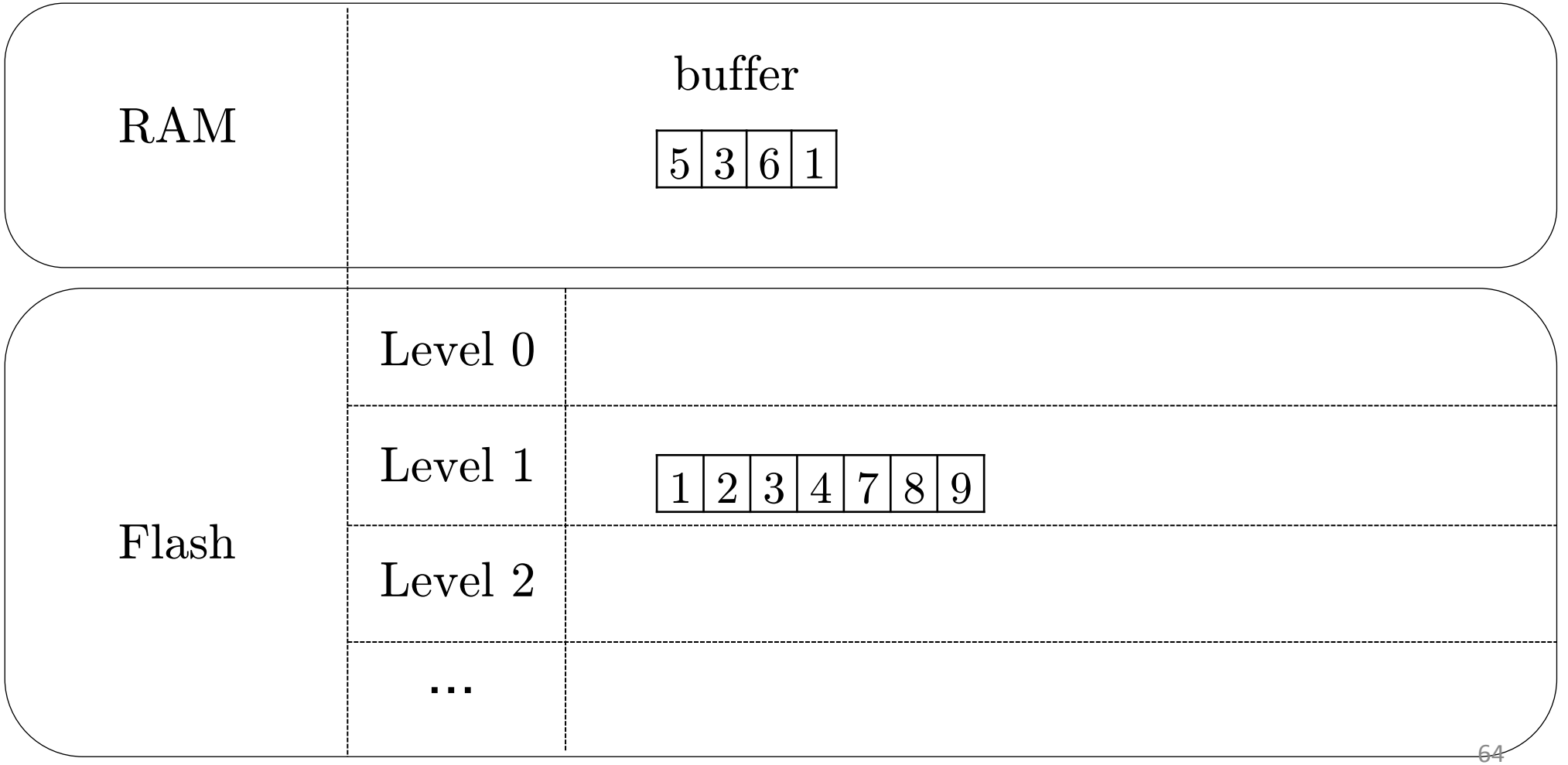
Solution



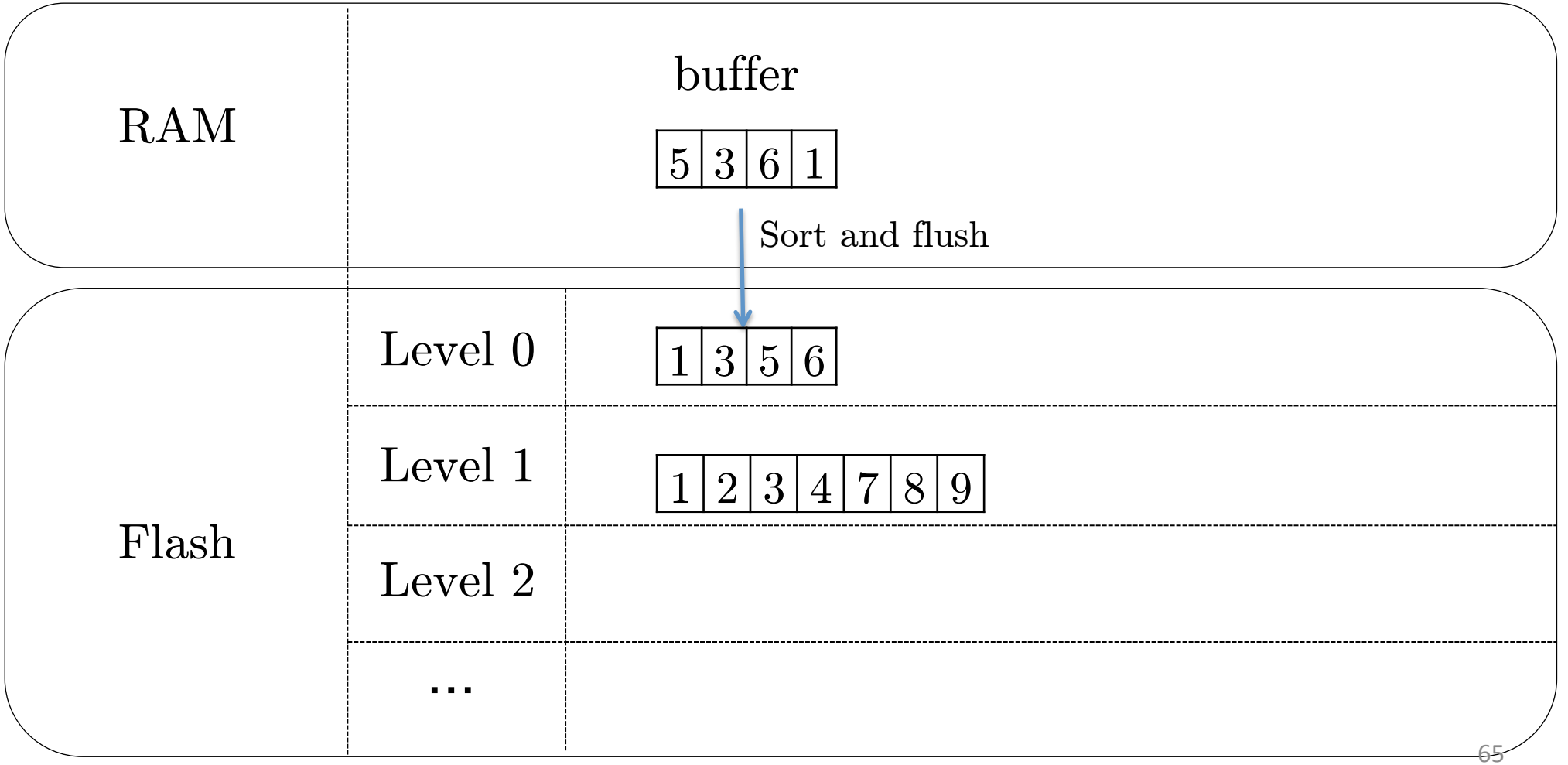
Solution



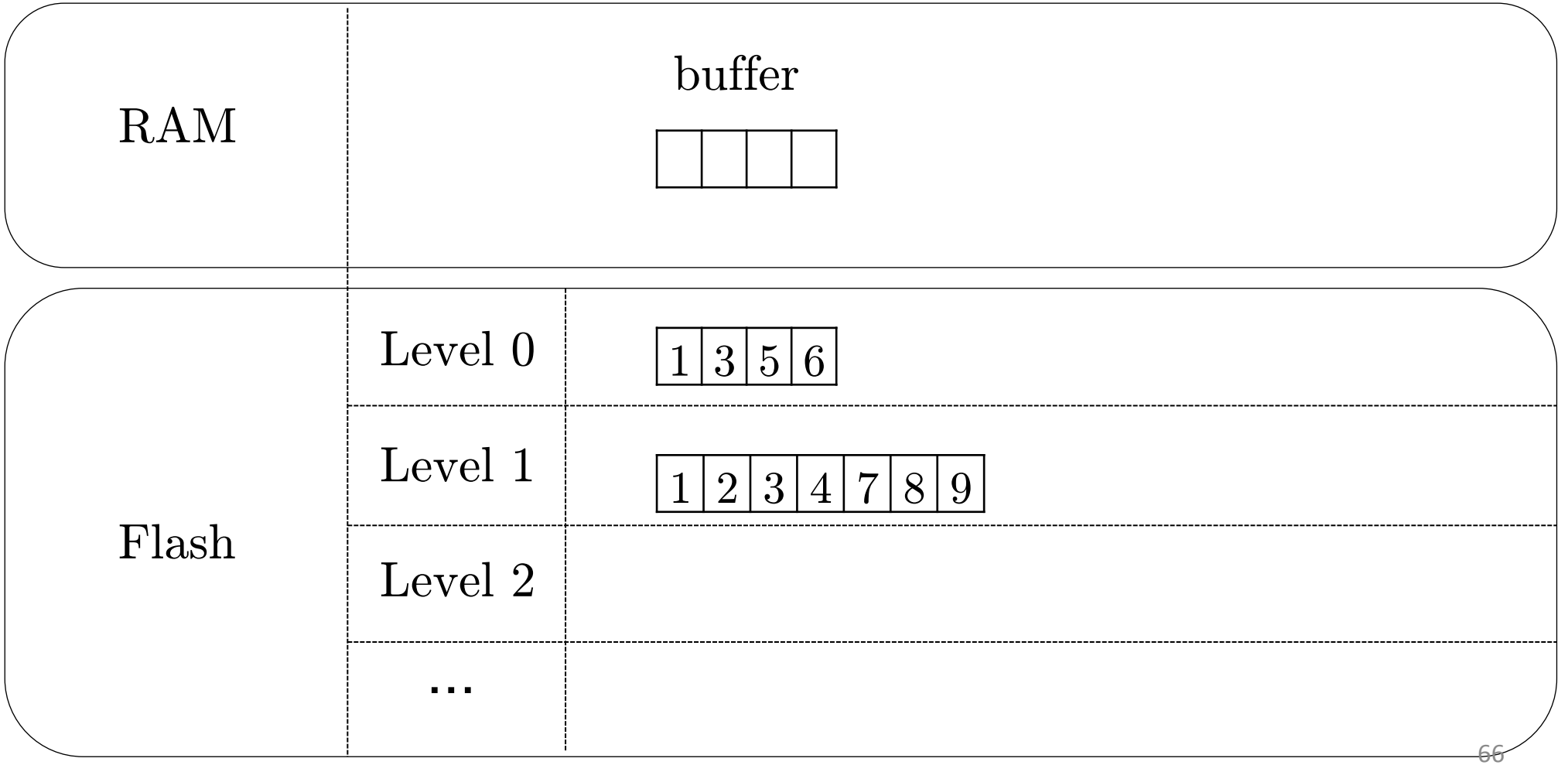
Solution



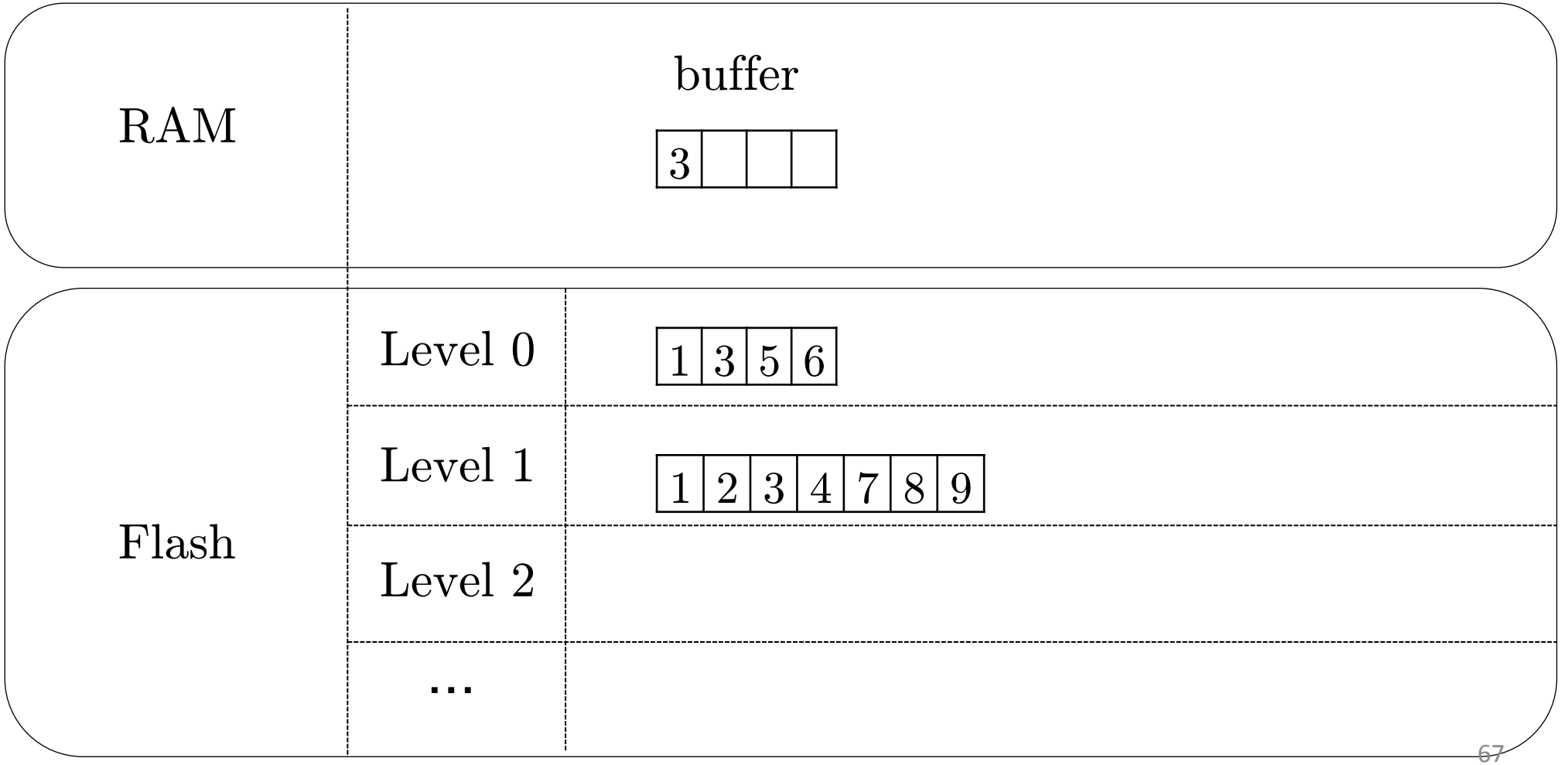
Solution


RAM
buffer
5 3 6 1
Sort and flush
Flash
Level 0
1 3 5 6
Level 1
1 2 3 4 7 8 9
Level 2
...

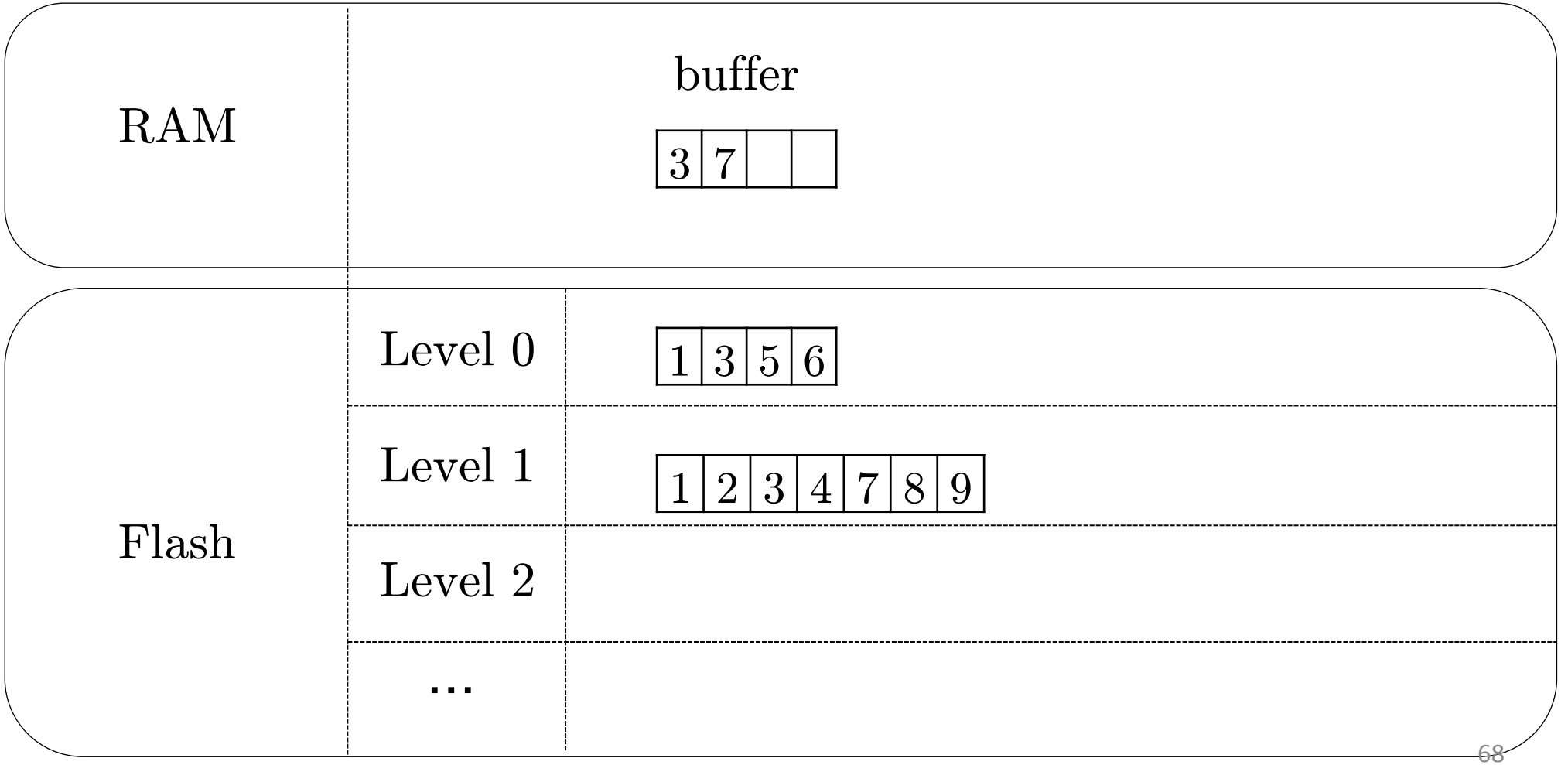
Solution



Solution

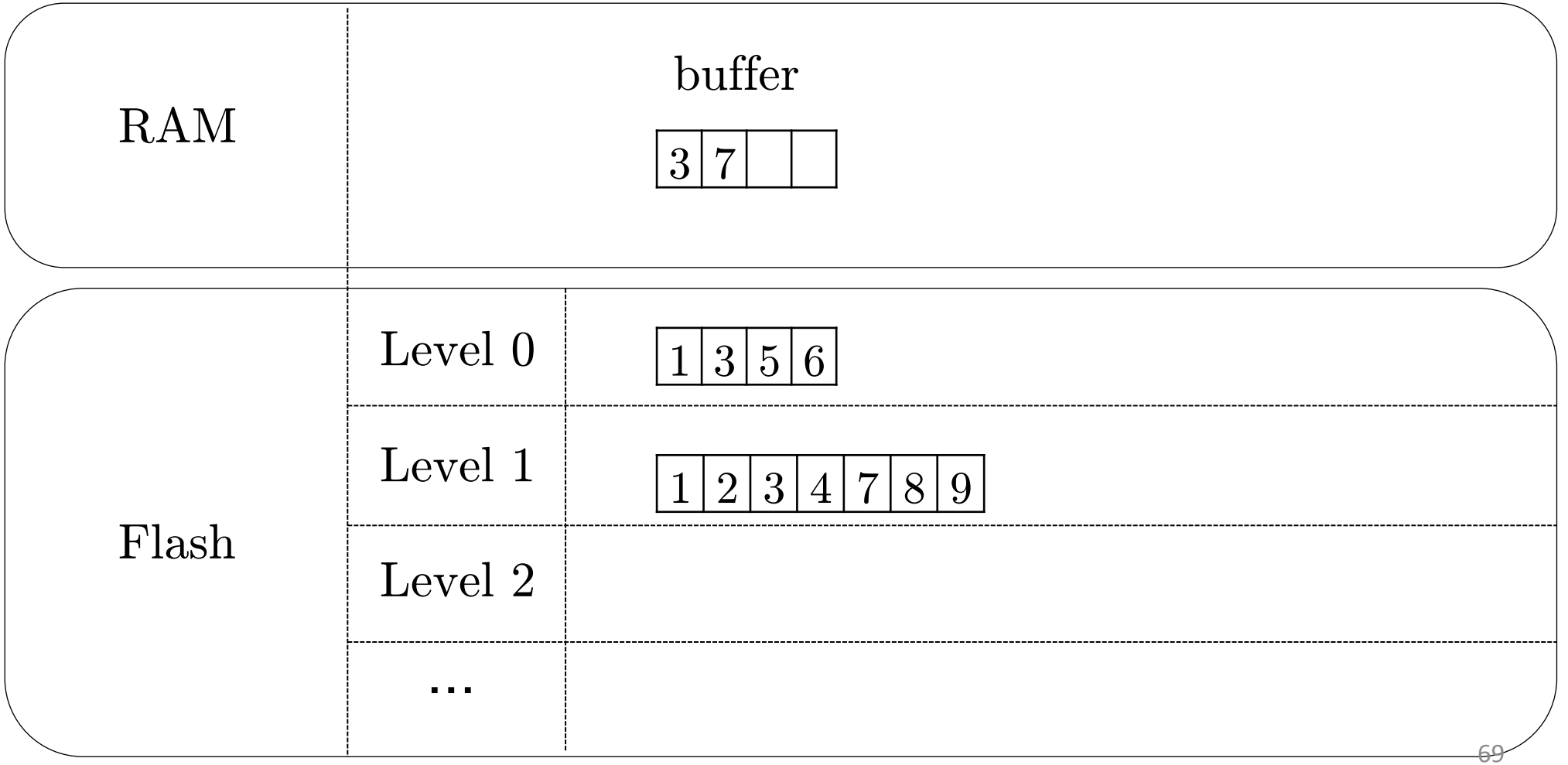


Solution



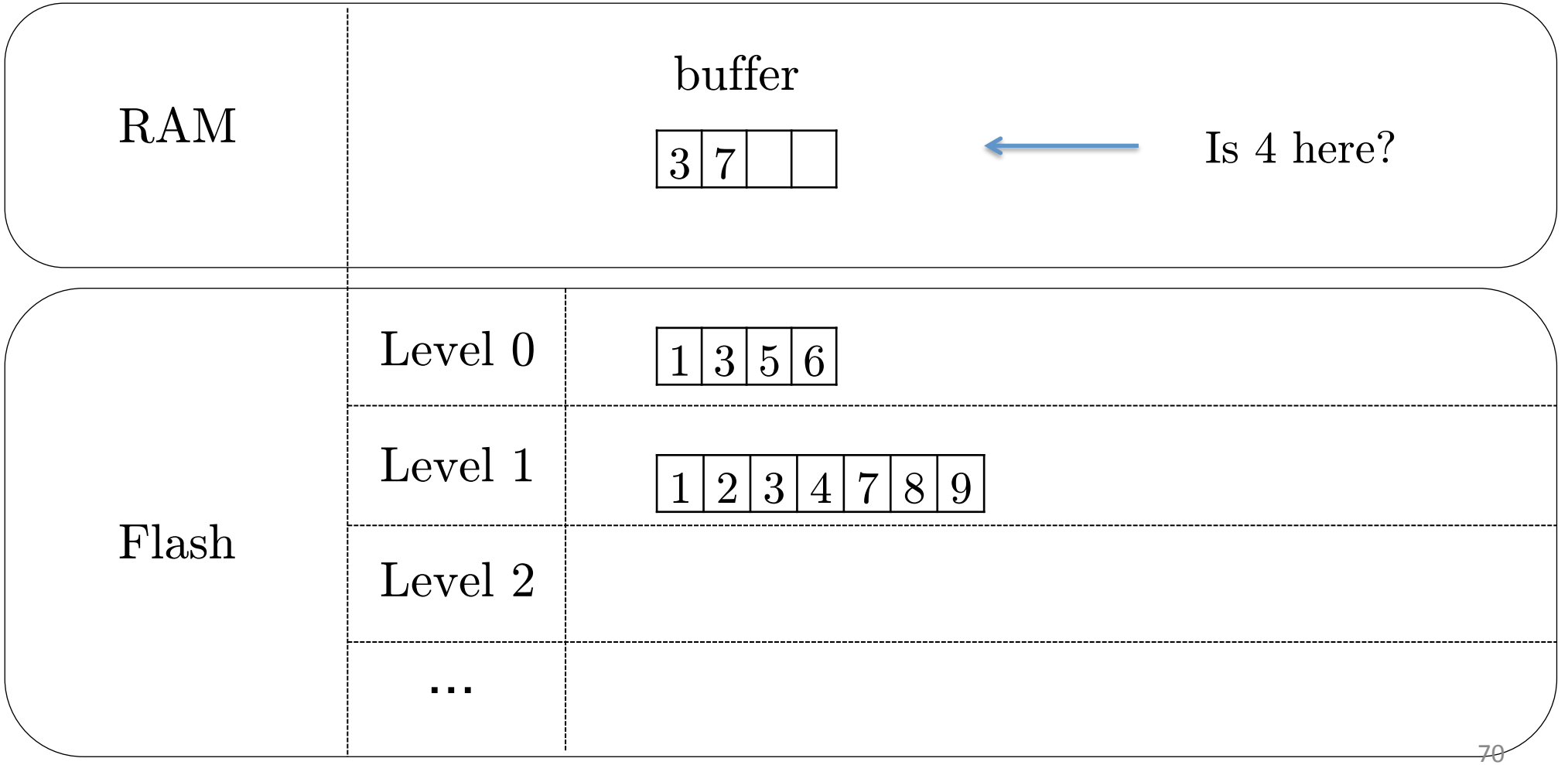
Solution

- Garbage-collect block 4



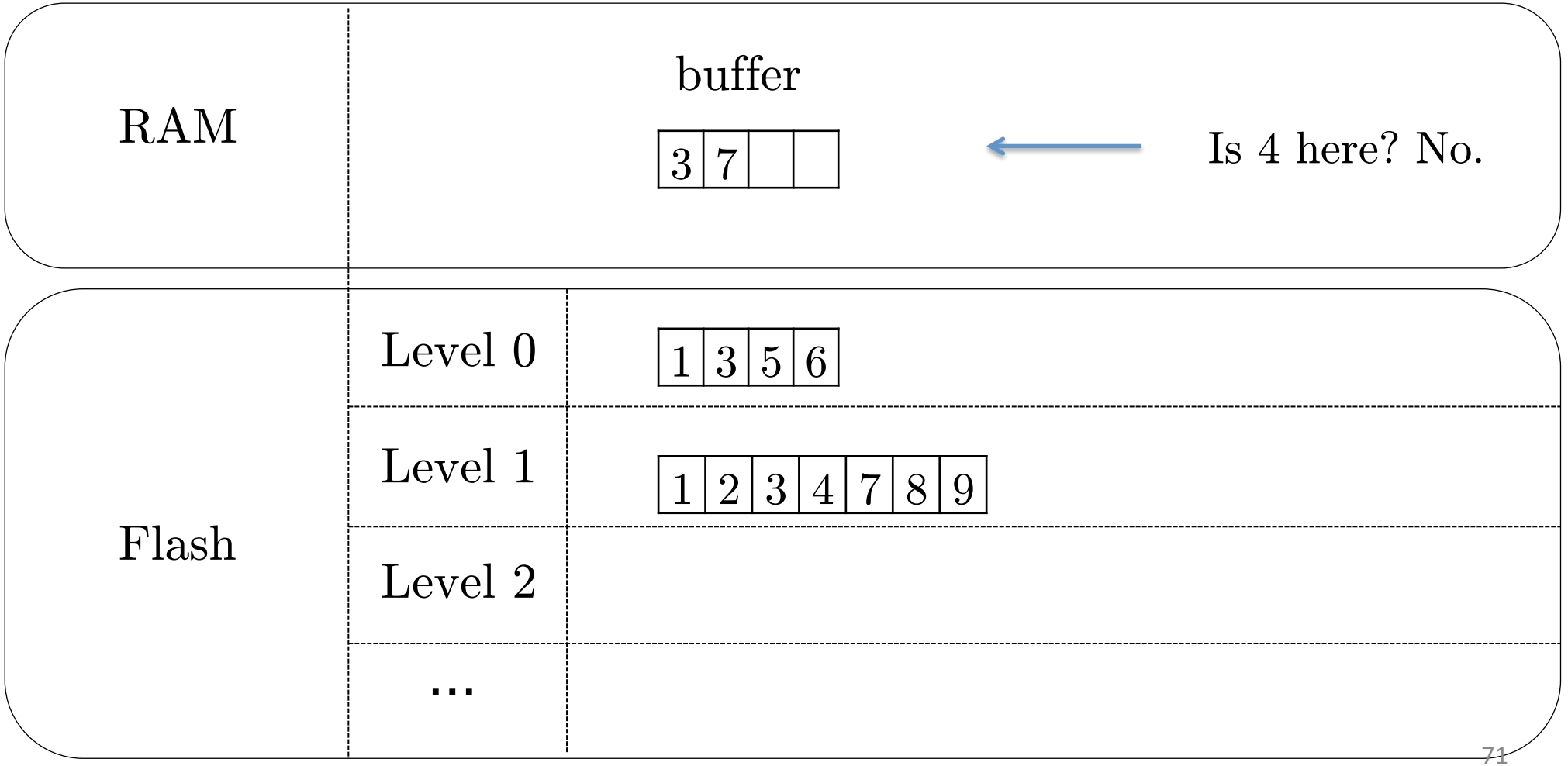
Solution

- Garbage-collect block 4



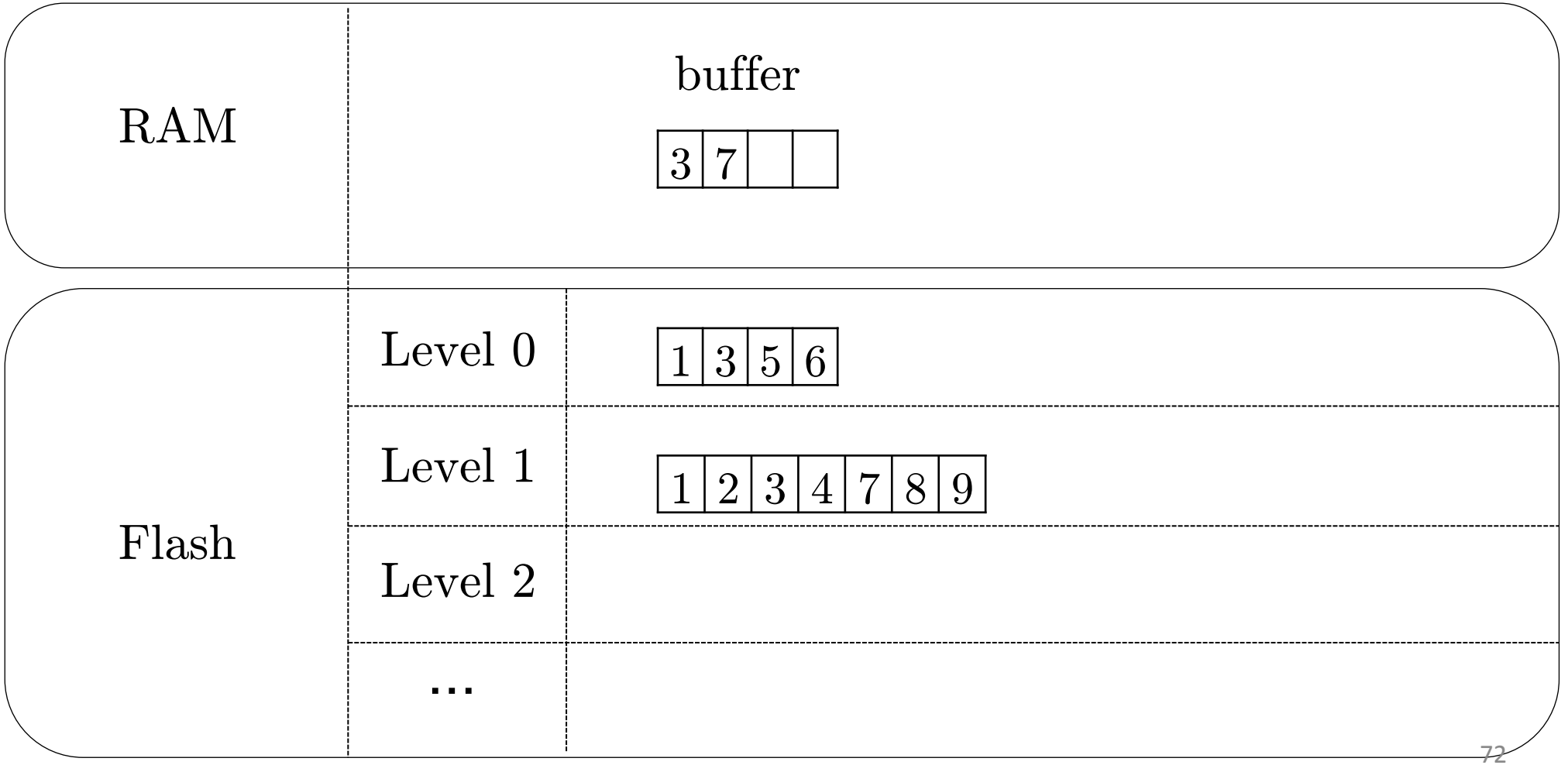
Solution

- Garbage-collect block 4



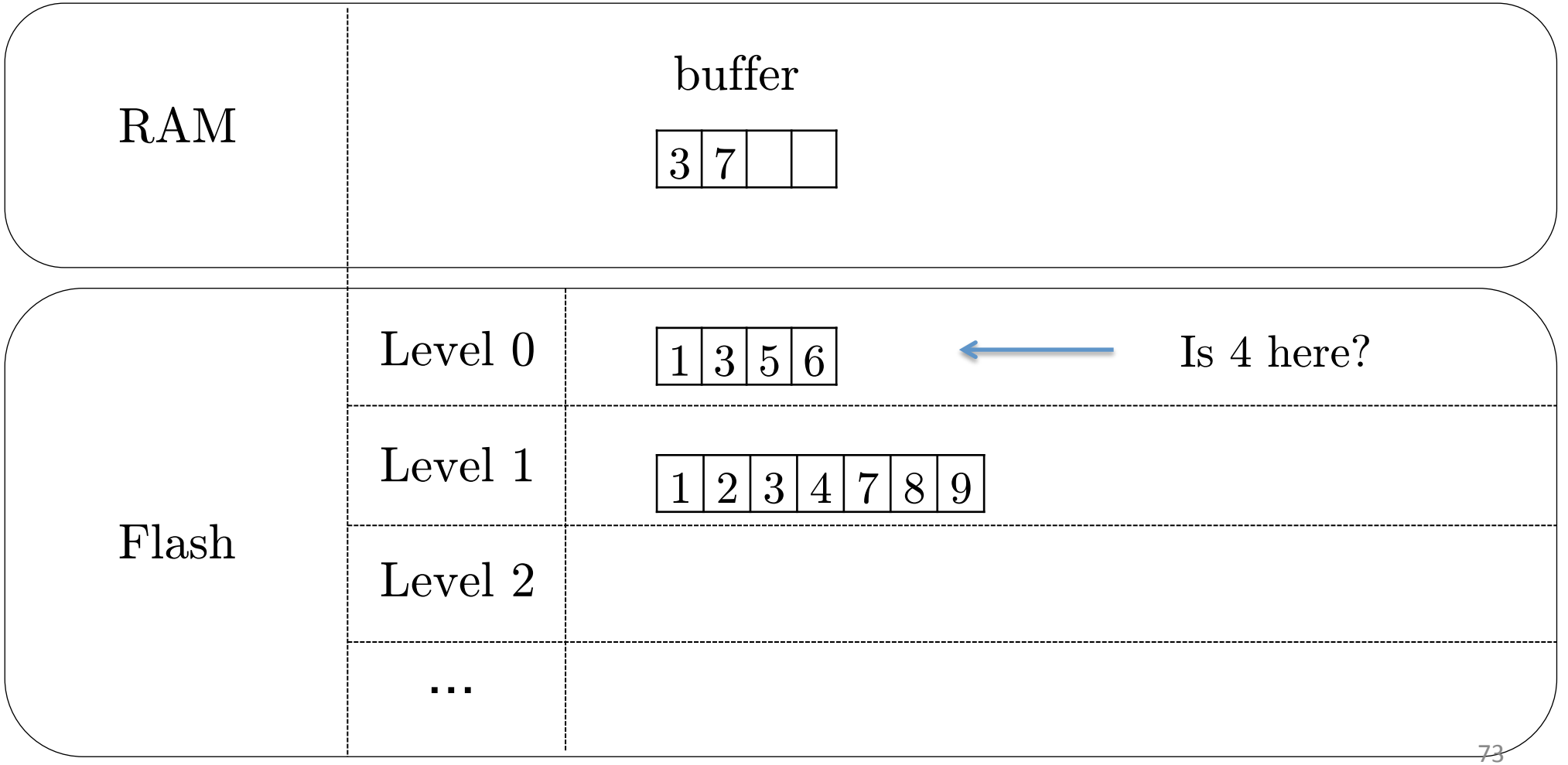
Solution

- Garbage-collect block 4



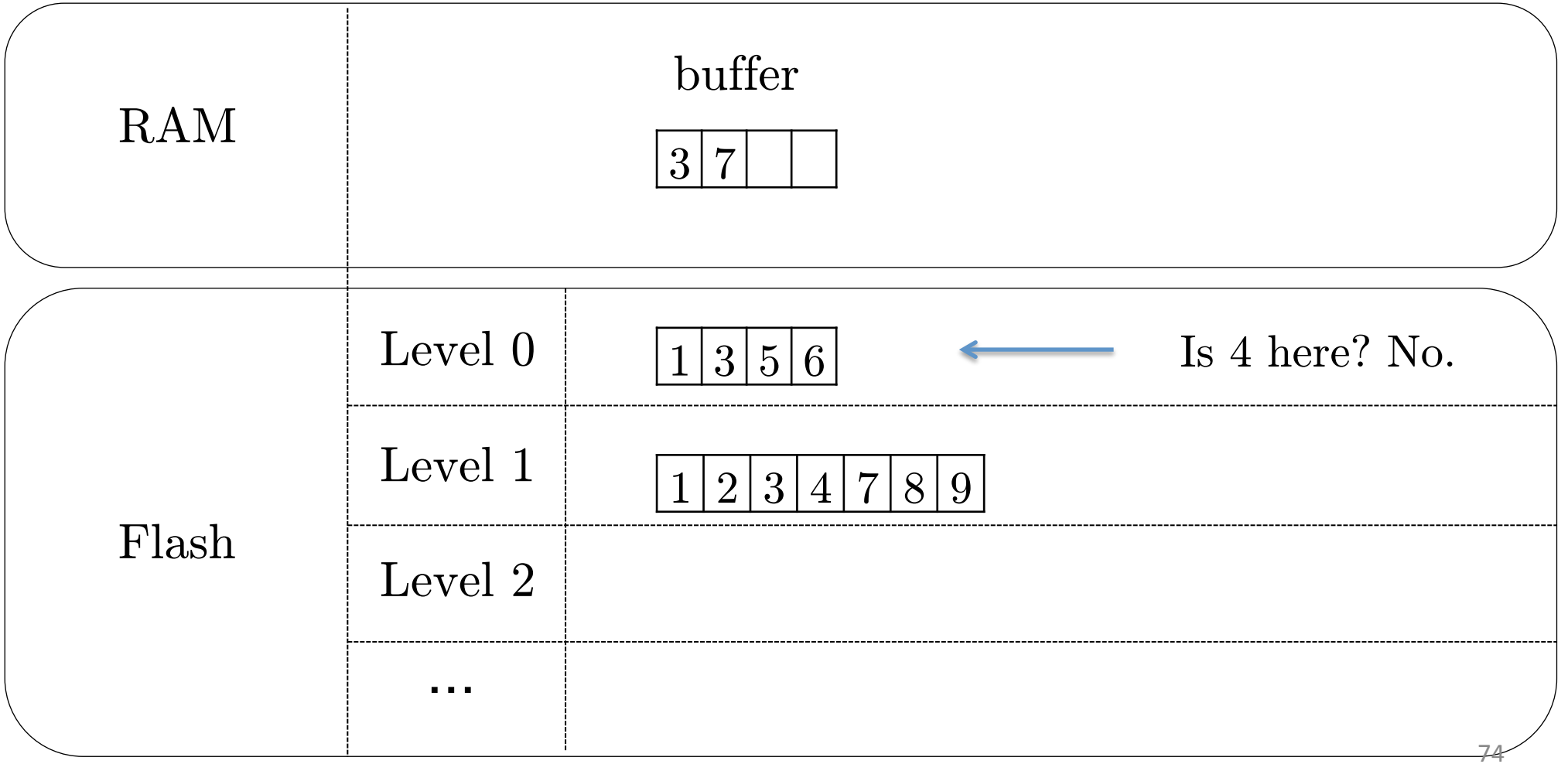
Solution

- Garbage-collect block 4



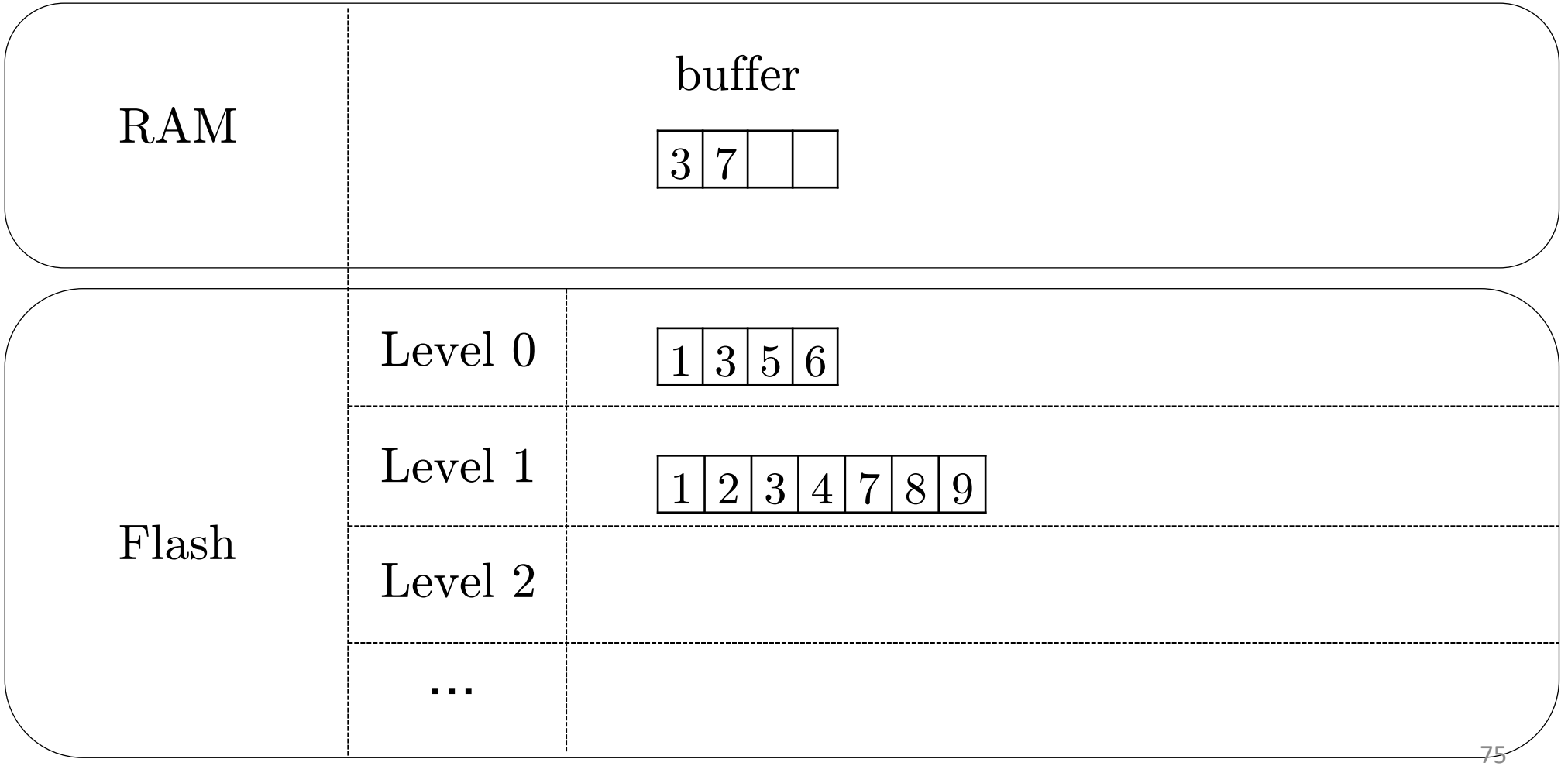
Solution

- Garbage-collect block 4



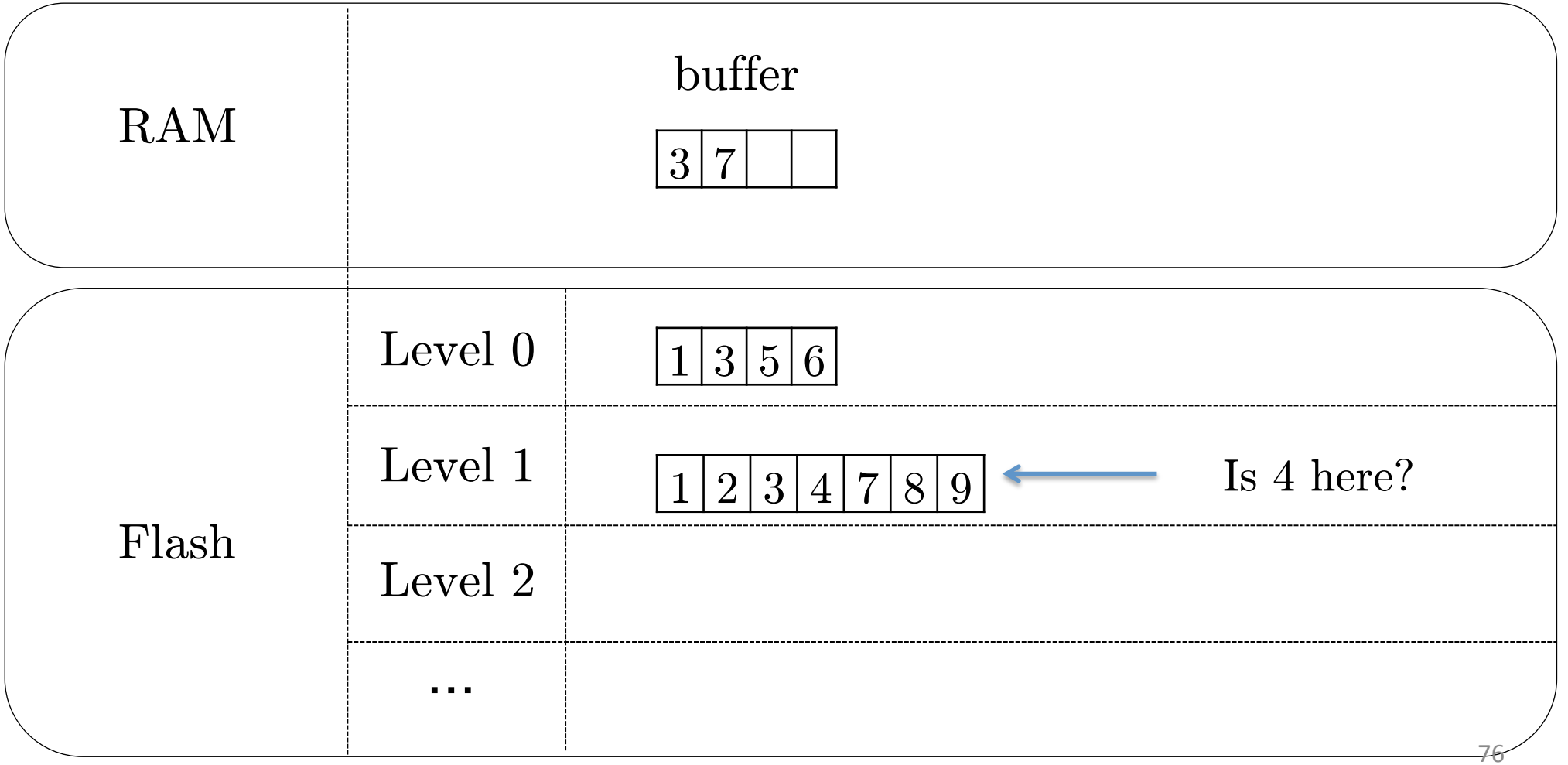
Solution

- Garbage-collect block 4



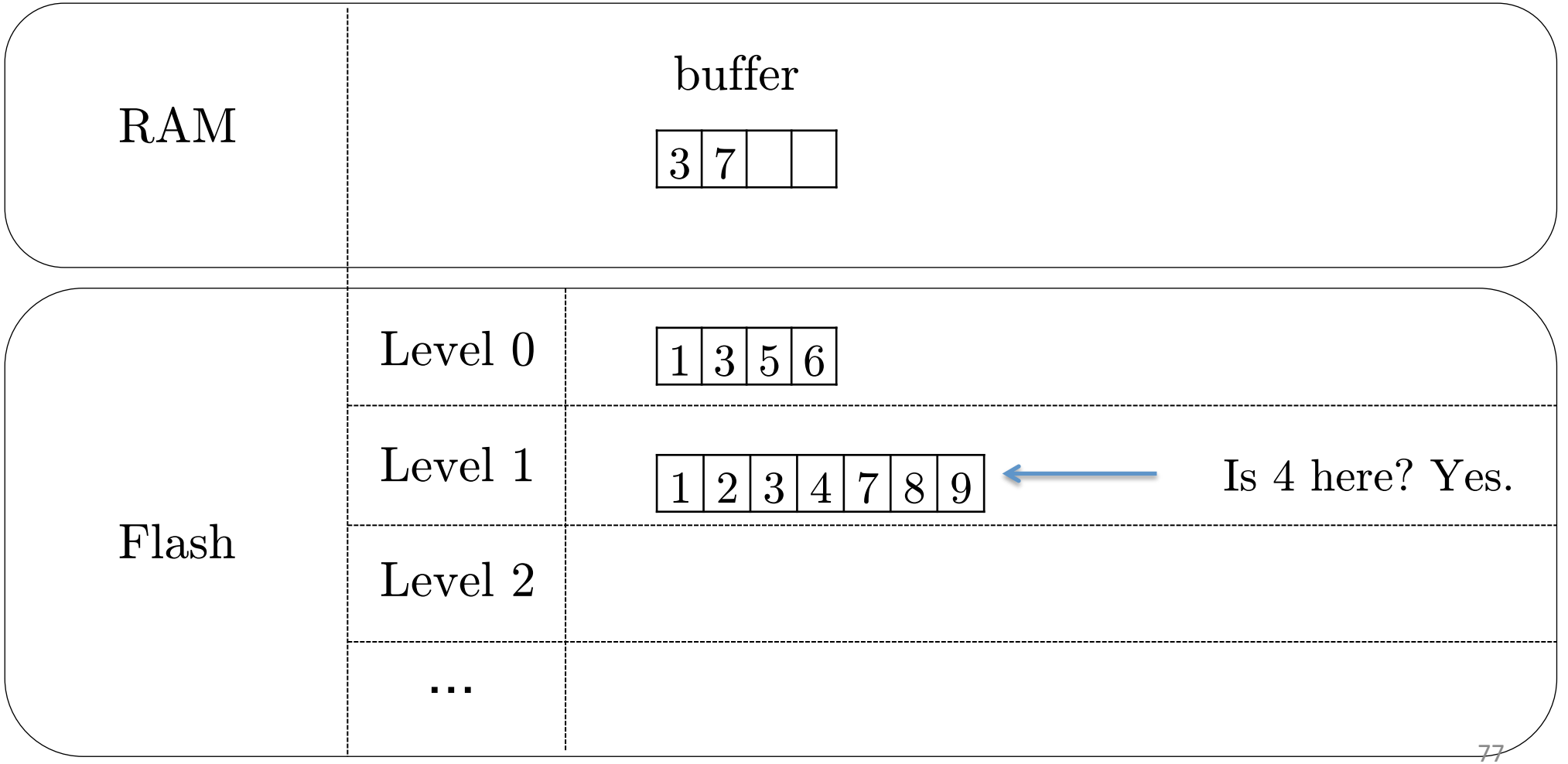
Solution

- Garbage-collect block 4



Solution

- Garbage-collect block 4



Solution

- **Lookup:** In general, search all substructures from smallest to largest, or until we find entry with erase flag set to 1
- Merge bitmaps
- We now have an image of which pages in the block are invalid
- Garbage-collection can take place

Analysis

Analysis

- K: number of flash blocks
- V: number of entries in the buffer
- T: LSM-tree's size ratio

- The number of levels is: $O(\log_T(\frac{K}{V}))$

Analysis

- Each entry is copied O(T) times within a level
- Each entry is copied $O(\log_T(\frac{K}{V}))$ times across levels
- Copying an entry takes $O(\frac{1}{V})$ flash write
- Write-amplification: $O(\frac{T}{V}\log_T(\frac{K}{V}))$

Analysis

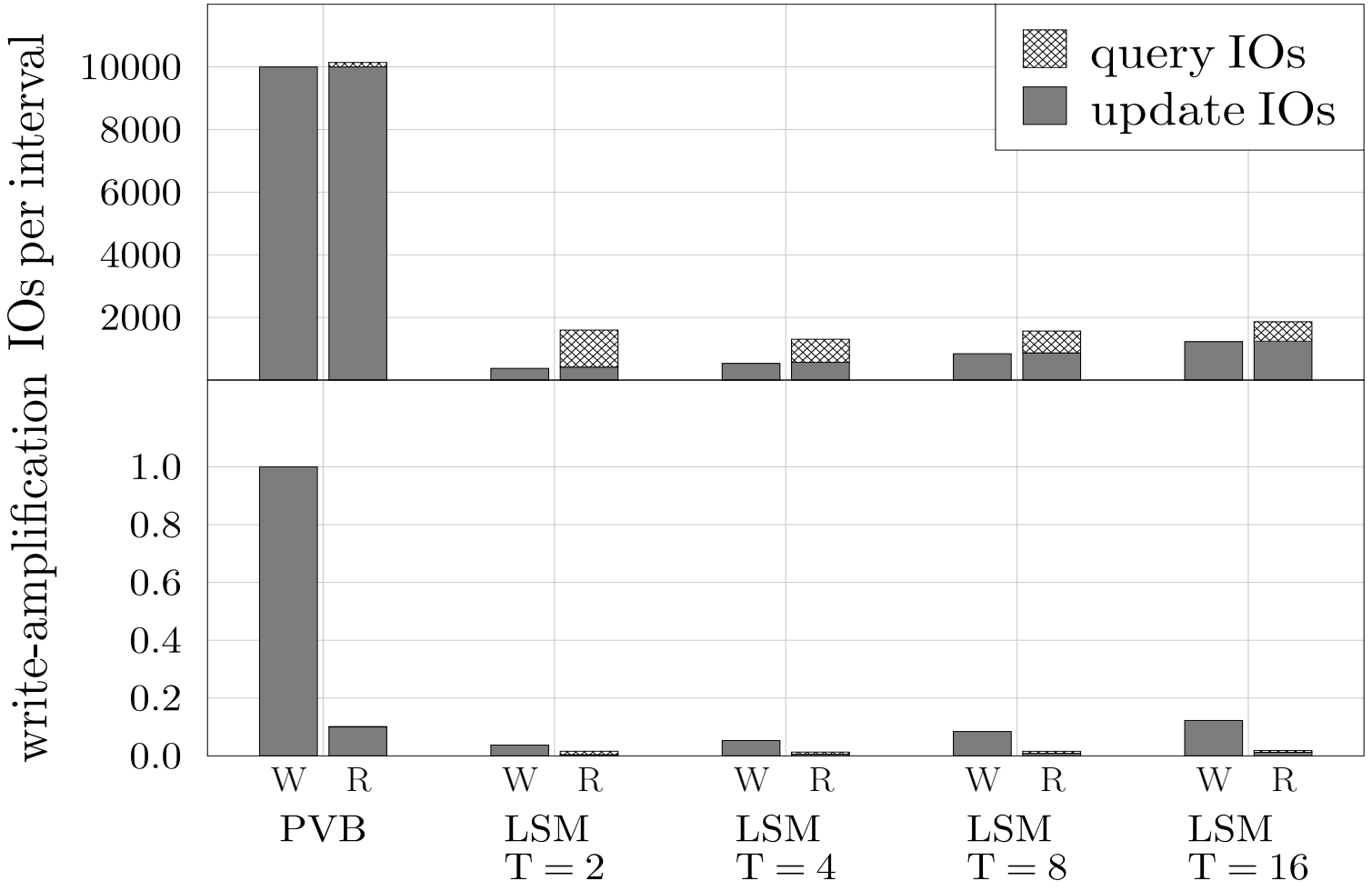
- This is sub-constant $O(\frac{T}{V}\log_T(\frac{K}{V}))$
- For typical values ≈0.02

- During garbage-collection $O(\log(\frac{K}{V}))$ reads
- But reads are cheap in flash
- And garbage-collection happens infrequently

Evaluation

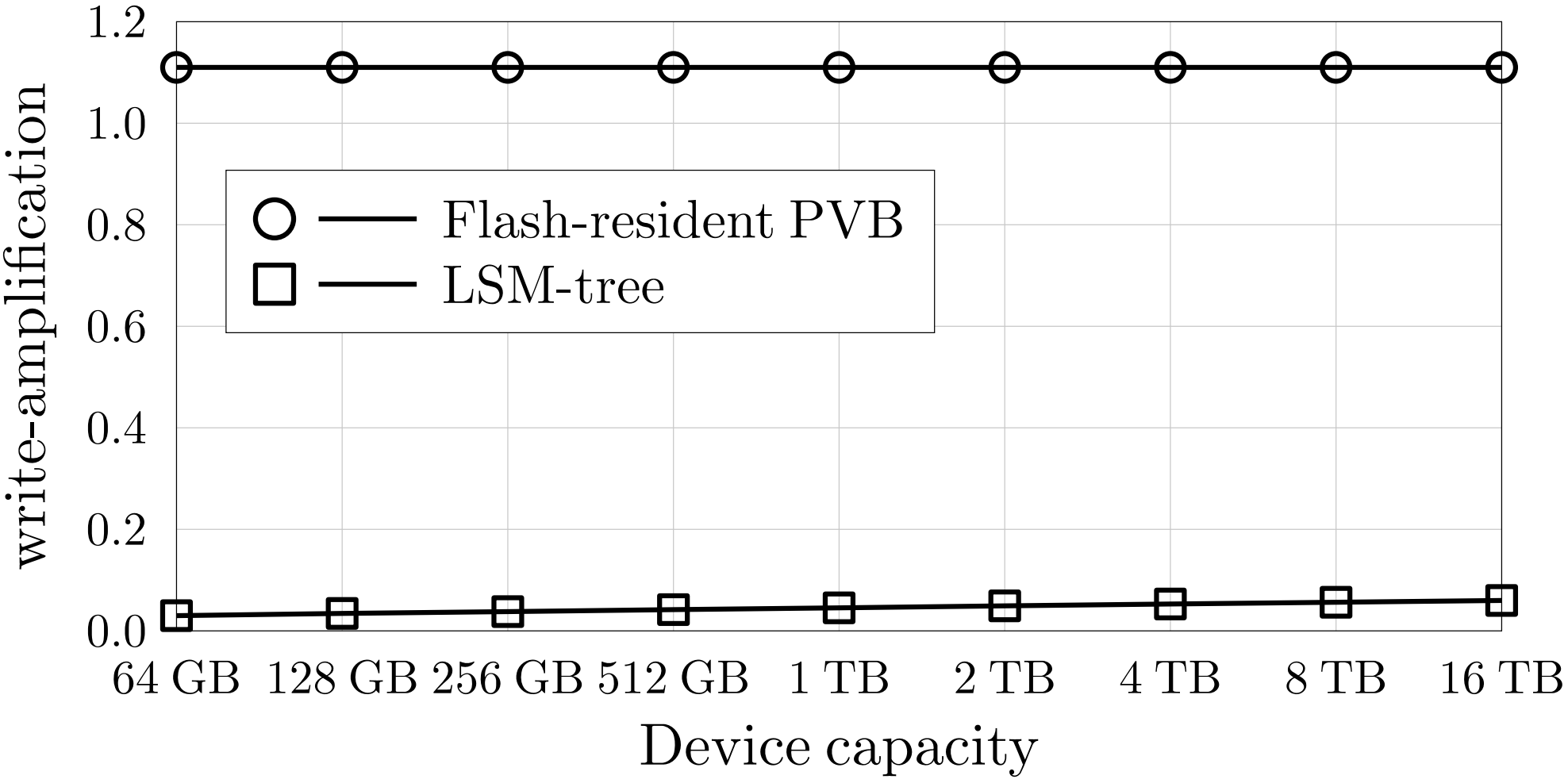
Analysis

- Tuning the LSM-tree's size ratio



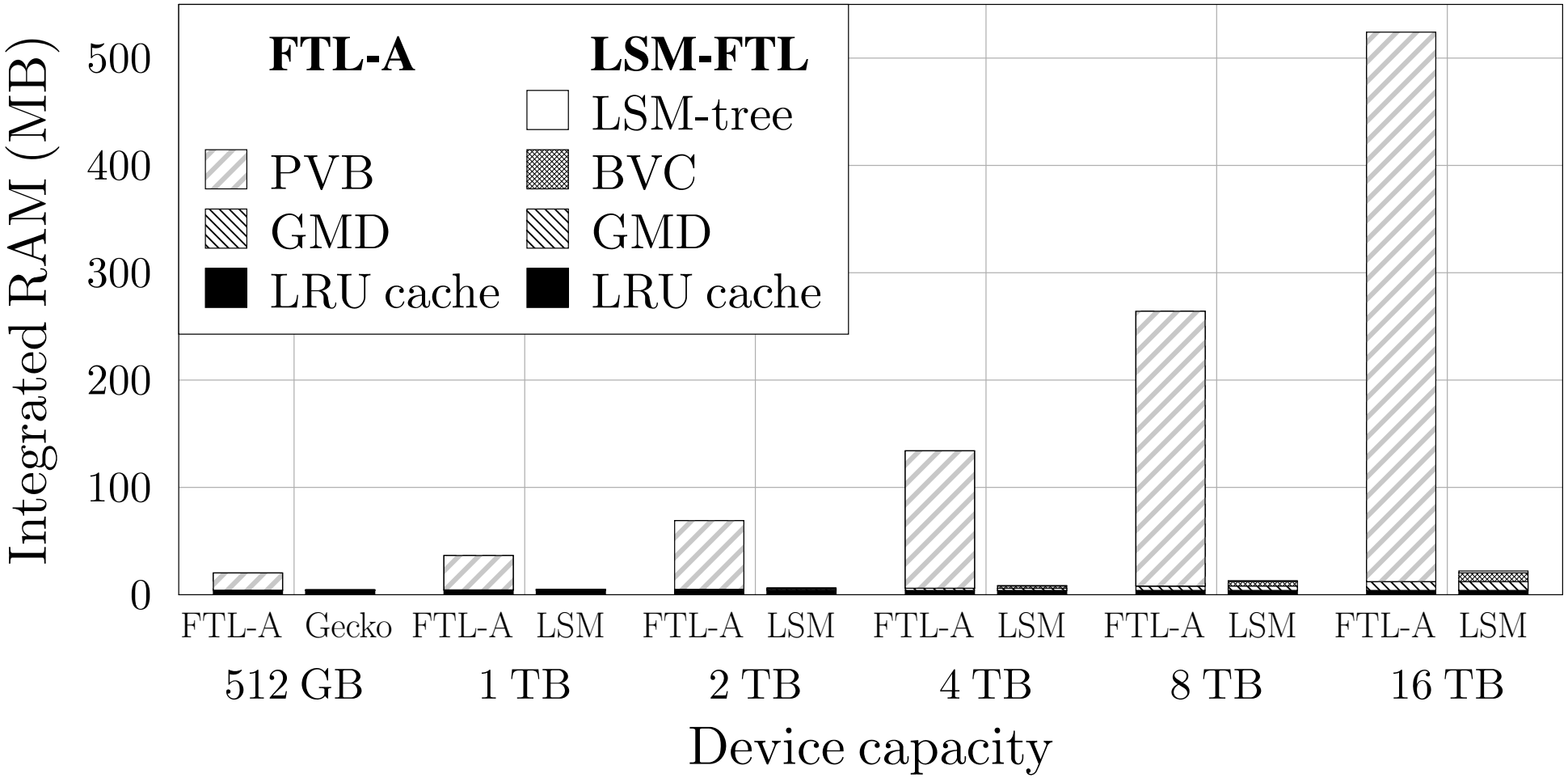
Analysis

- Write-amplification is low and scales



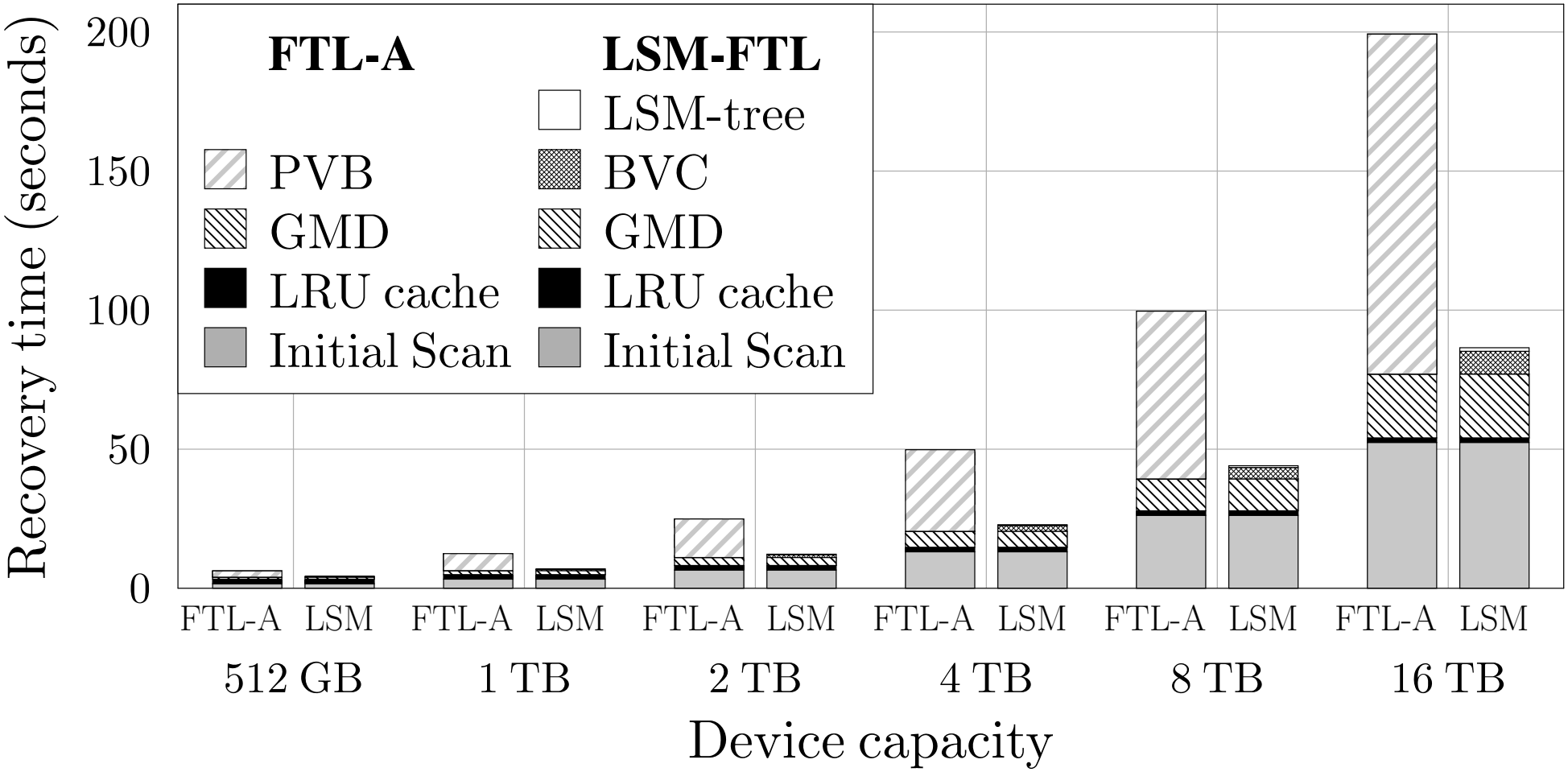
Analysis

- RAM-requirement is 95% lower



Analysis

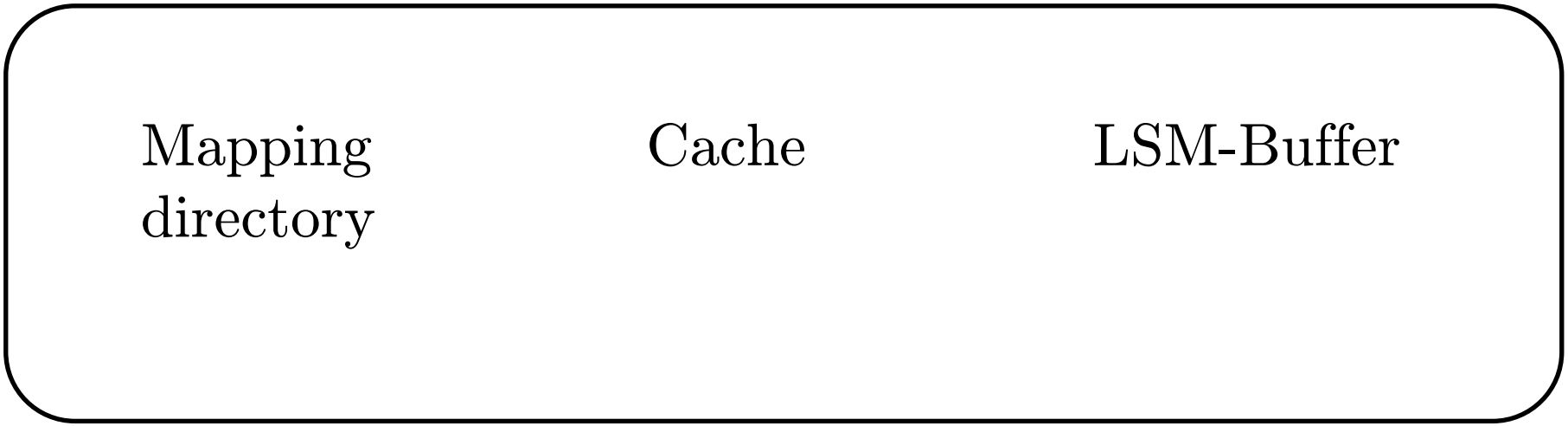
- Recovery time is 51% lower



Other Concerns

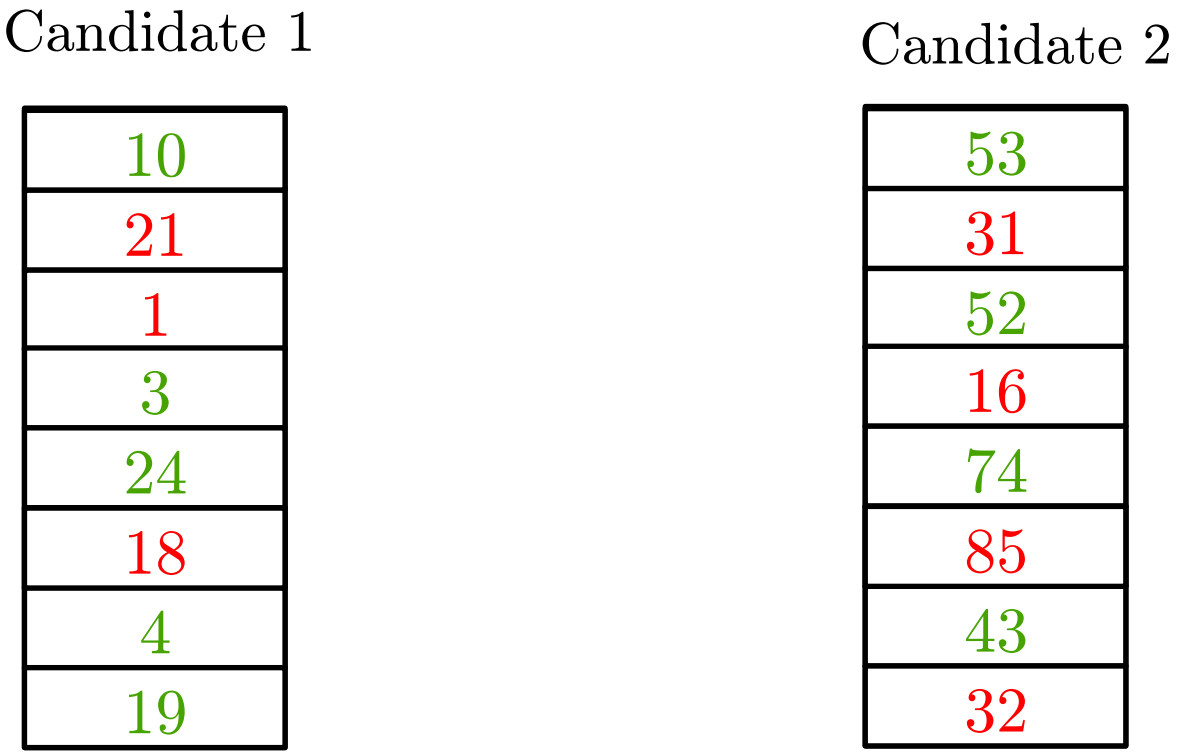
Other Concerns

- **Power Failure**
  - We lose the contents of the buffer
  - We lose dirty mapping entries in the cache



Other Concerns

- **Picking Garbage-Collection Victims**
  - Greedy approach: least number of live pages
  - Non-optimal



Other Concerns

- **Reason:** suppose blocks in block 2 are extremely frequently updated.
- Most would soon be invalidated anyways.

Candidate 1

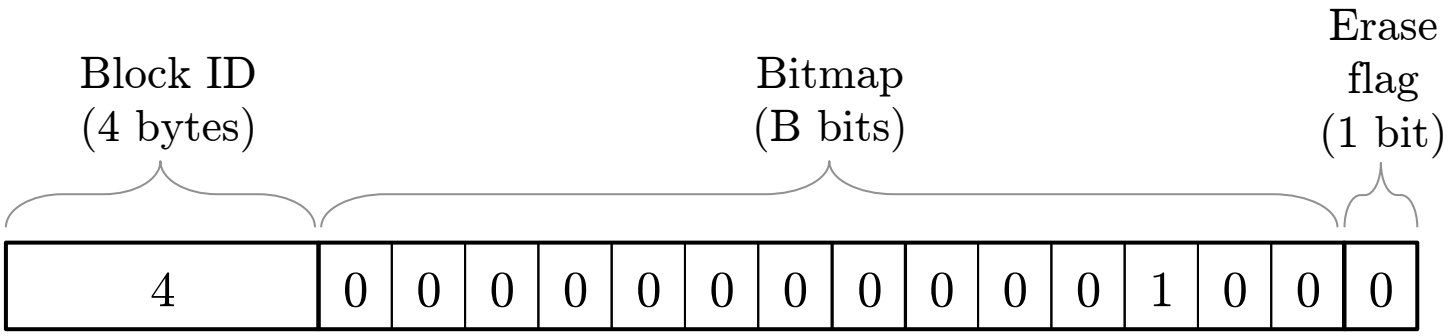
10
21
1
3
24
18
4
19

Candidate 2

53
31
52
16
74
85
43
32

Other Concerns

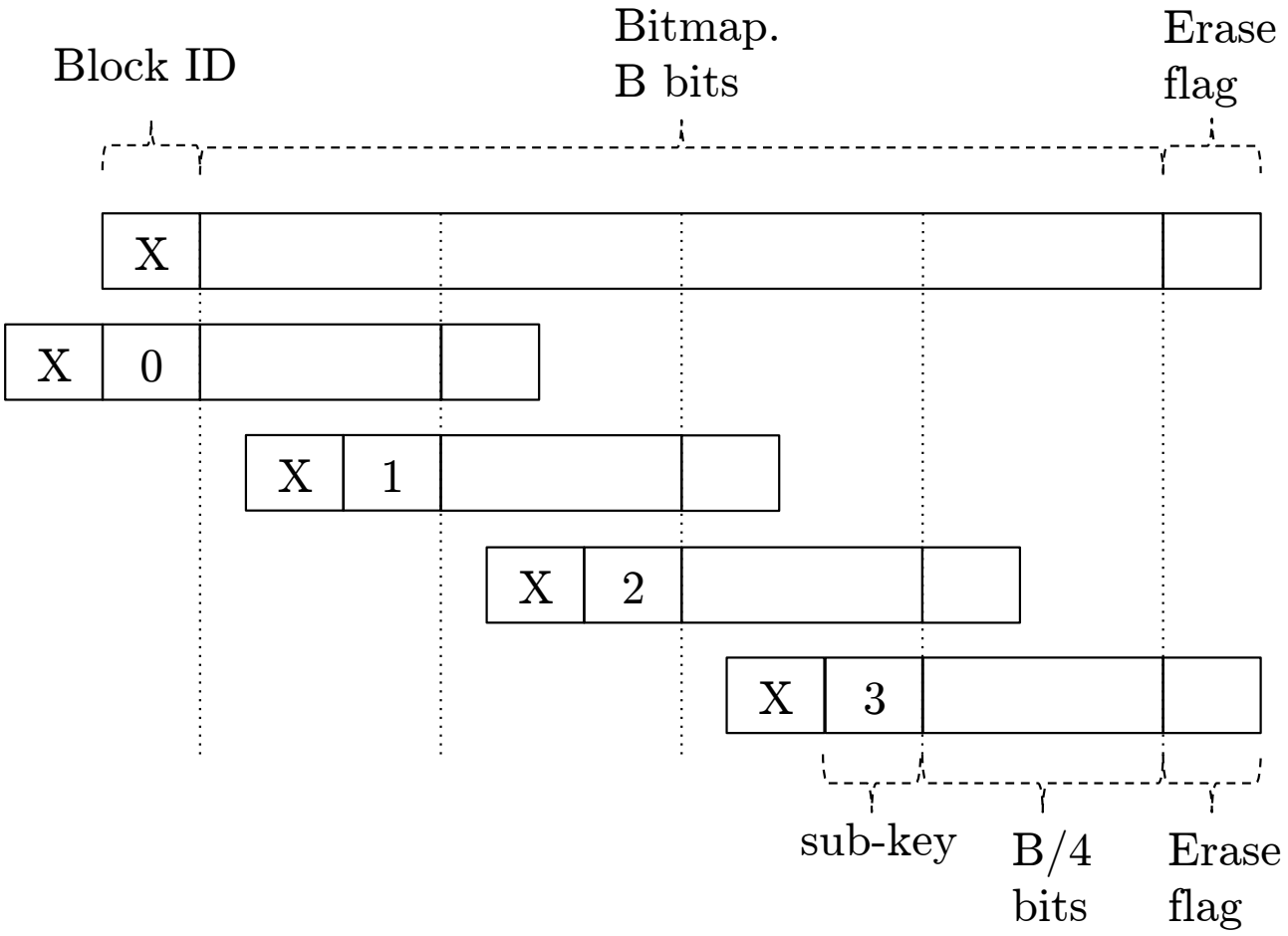
- Write-amplification depends on V: $O(\frac{T}{V}\log_T(\frac{K}{V}))$
- V is the number of entries fitting into buffer
- Entry size depends on block size



- As devices scale, block size increases, so V decreases

Other Concerns

- Solution: entry-partitioning



Conclusion

Conclusion

- We can store page validity metadata in flash while keeping write-amplification low
- Reduces RAM requirement by 95%
- Reduces recovery time by 51%
- Write-amplification increases by 0.02%

Conclusion

- LSM-trees can be applied beyond their usual application for key-value storage

- Thanks. Q&A.