Redesigning LSMs for Nonvolatile Memory with NoveLSM

Sudarsun Kannan, Nitish Bhat[*], Ada Gavrilovska[*],

Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

University of Wisconsin-Madison, Georgia Institute of Technology[*]

Key-Value Stores

Key-Value Stores

Keys	Arbitrary Value
100	{red car, honda, john}
200	

Widely used

Google facebook Quora Pinterest

mongoDB Lawrence Livermore National Laboratory redhat ethereum

LSM-based Key-Value Stores

Log-structured Merge Tree (LSM)

- Write optimized data structure used in key-value stores

Originally designed for slow hard drives

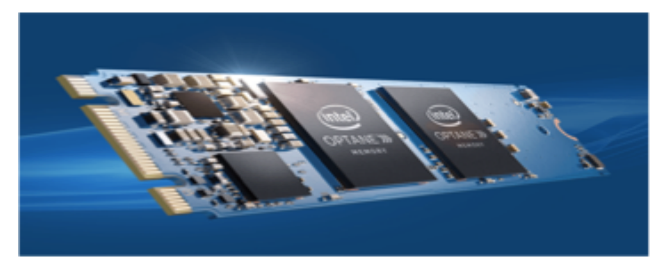
- In memory buffering, batched, and sequential writes to disk
- High write amplification

Several LSM implementations

- LevelDB (Google), RocksDB (Facebook), Cassandra
- SSD optimized LSMs WiscKey (FAST '16), VT-tree (FAST '13)

Moving Towards NVM Era

Fast byte-addressable and persistent NVM technologies expected soon

	Hard Drives	SSD	NVM	DRAM
				
BW:	2.6 MB/s	250 MB/s	**5-10 GB/s**	64 GB/s
H/W Lat:	7.1 ms	68 us	**500ns - 2us**	100ns
Persistence:	Blocks	Blocks	**Cache-line**	Cache-line

Adding NVM makes LSMs faster?

Why use LSMs in NVM?

– Expected to co-exist with block storage

– Rewriting production-level LSMs not easy!

Current LSMs are not designed to exploit storage byte-addressability

Our study shows significant software overheads

1. Serialization and deserialization cost
2. Compaction cost
3. Logging cost
4. Lack of read parallelism

Our Solution: NoveLSM

Use existing LSM and…

1. Reduce serialization – Persistent Skip List
2. Reduce compaction – Direct NVM mutability
3. Reduce logging cost – In-place commits
4. Improve parallelism – Read parallelism across levels

Evaluation Summary:

Evaluation with emulated NVM using benchmarks and application traces

NoveLSM reduces write latency by up to 3.8x and read latency by 2x

Orders of magnitude faster recovery

Outline

Introduction

Background on LevelDB

Motivation

- High serialization, compaction, and logging cost
- Lack of parallelism

NoveLSM Design

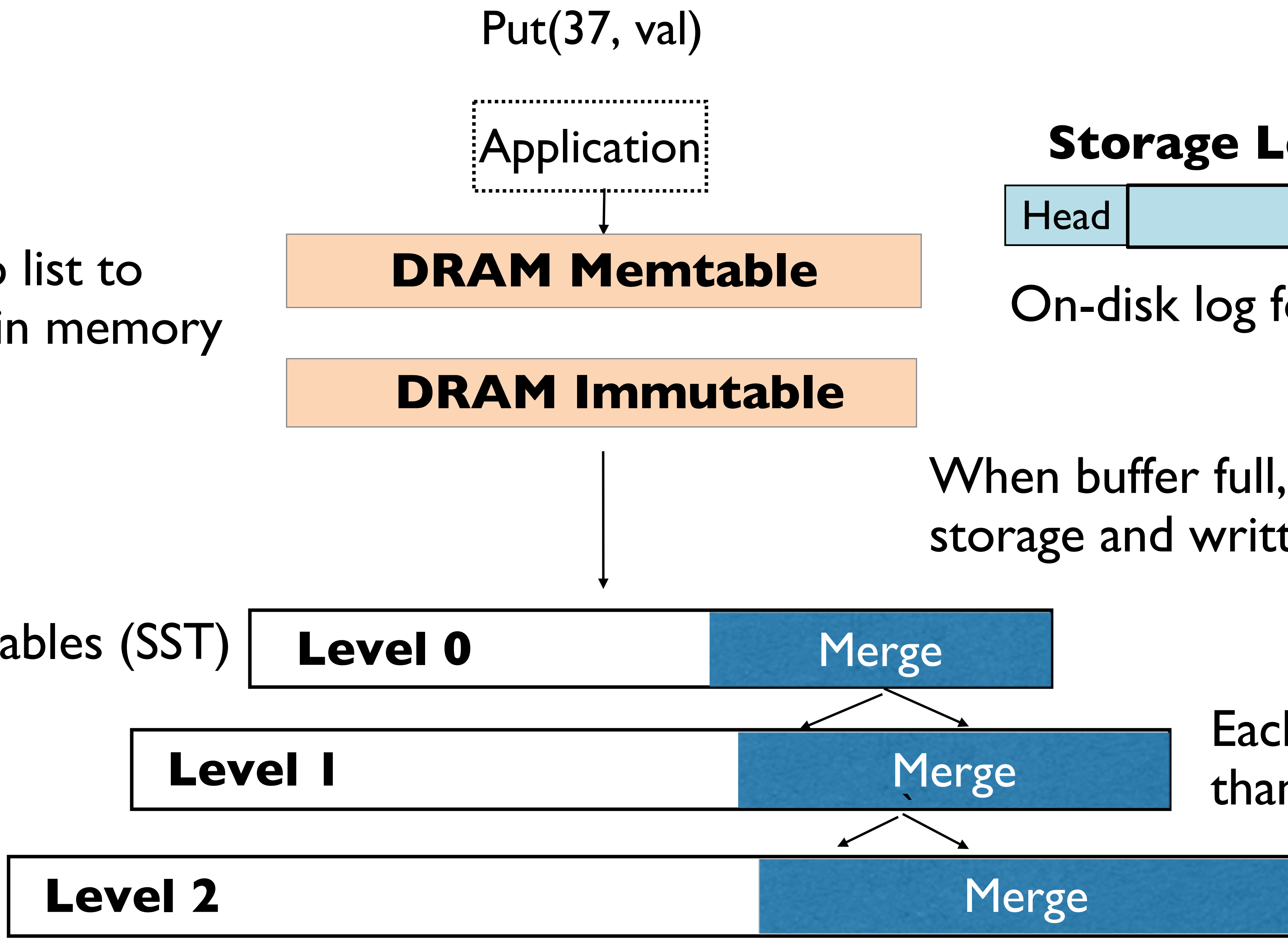
- Persistent memtable, NVM mutability, In-place commits
- Read parallelism

Evaluation

Conclusion

LSM-based LevelDB

We study (and extend) LevelDB due to its wider use and simplicity



In-memory skip list to buffer updates in memory

On-disk log for recovering from failure

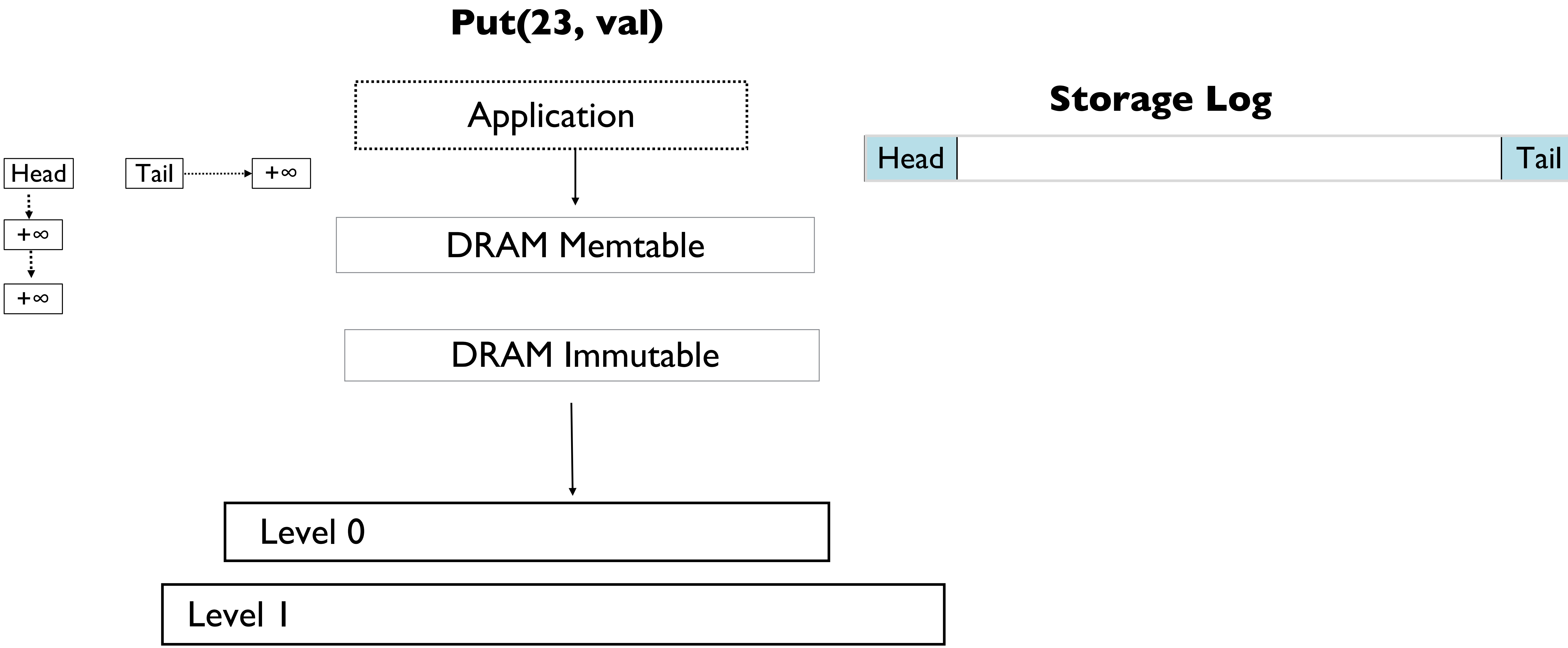
When buffer full, writes compacted to storage and written sequentially

String Sorted Tables (SST)

Each level is 10x larger than previous level

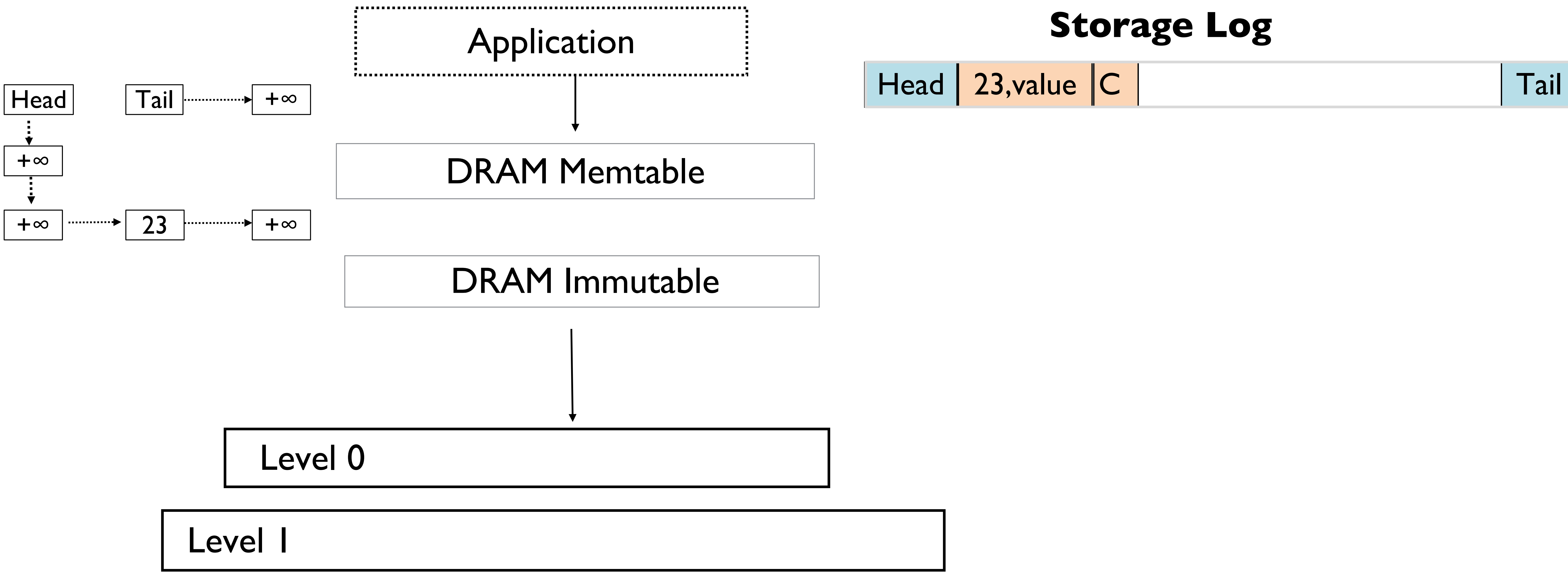
When a level is full, data moved to next level by merging

Write Operation

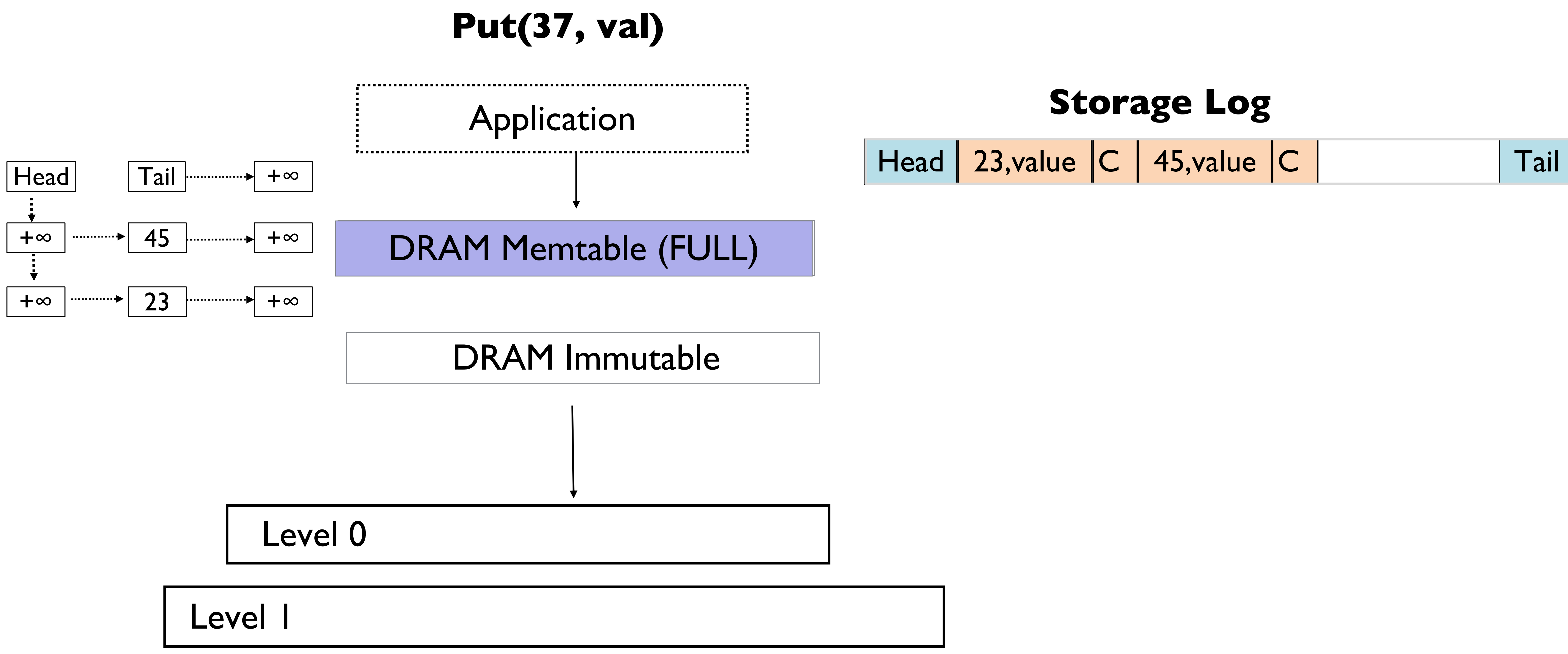


Write Operation

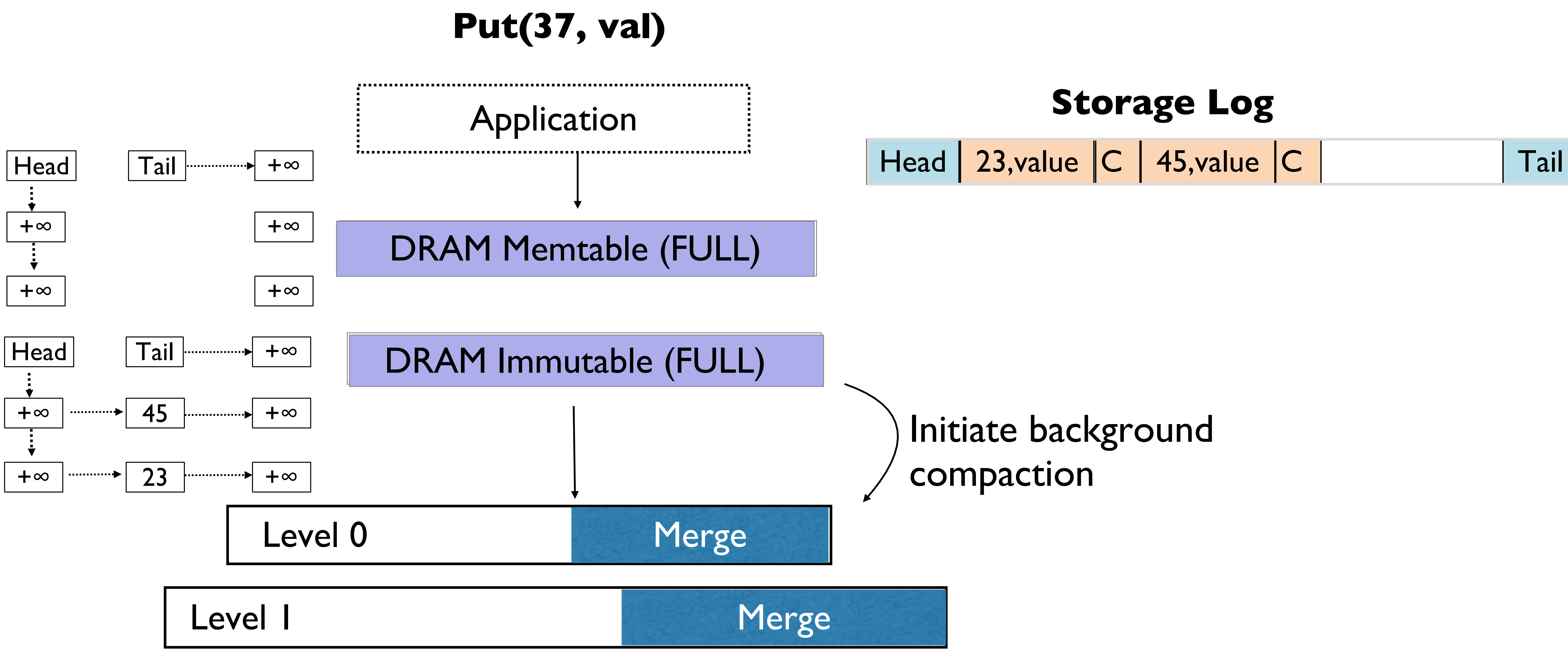
Put(45, val)



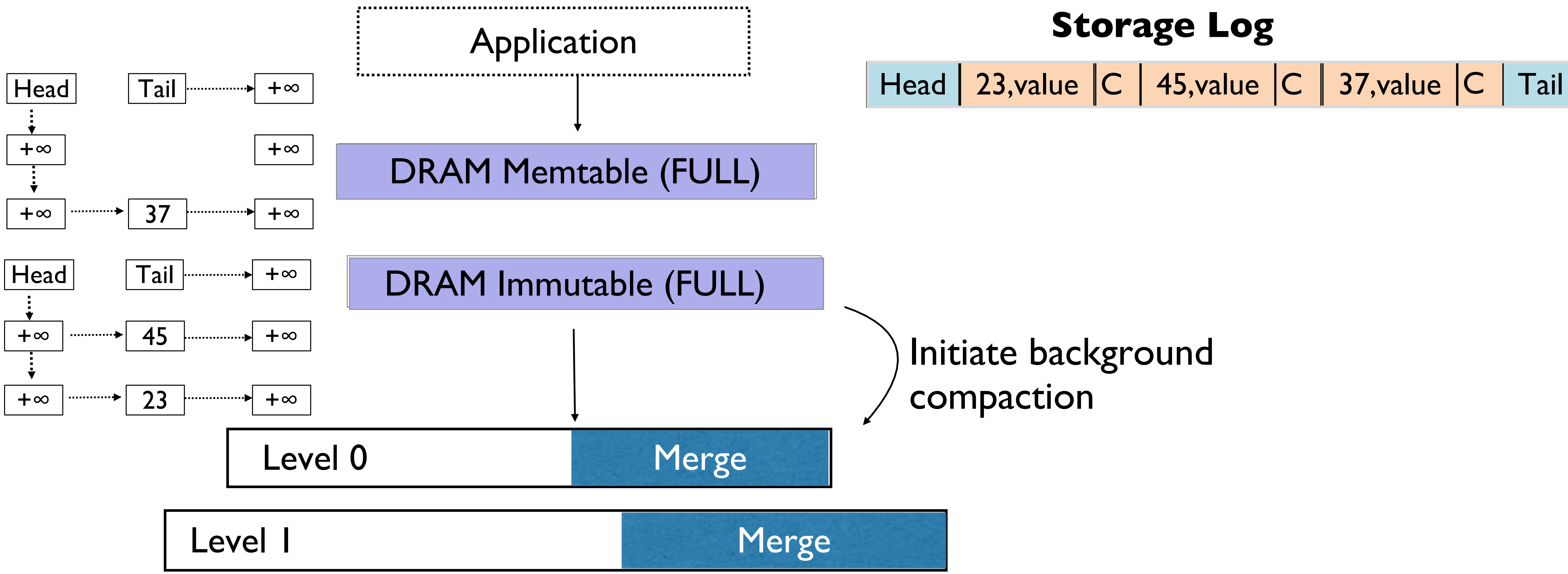
Write Operation



Write Operation

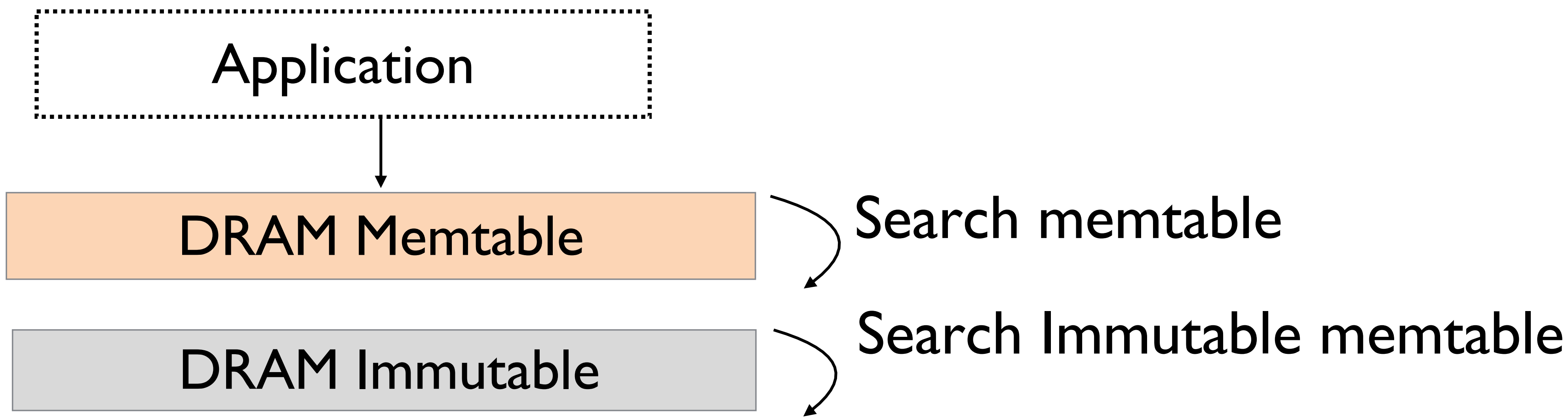


Write Operation



Read Operation

Get ("107")



Level 0	0	1	2	3	….

Range	Blocks
d	12
a	5-6

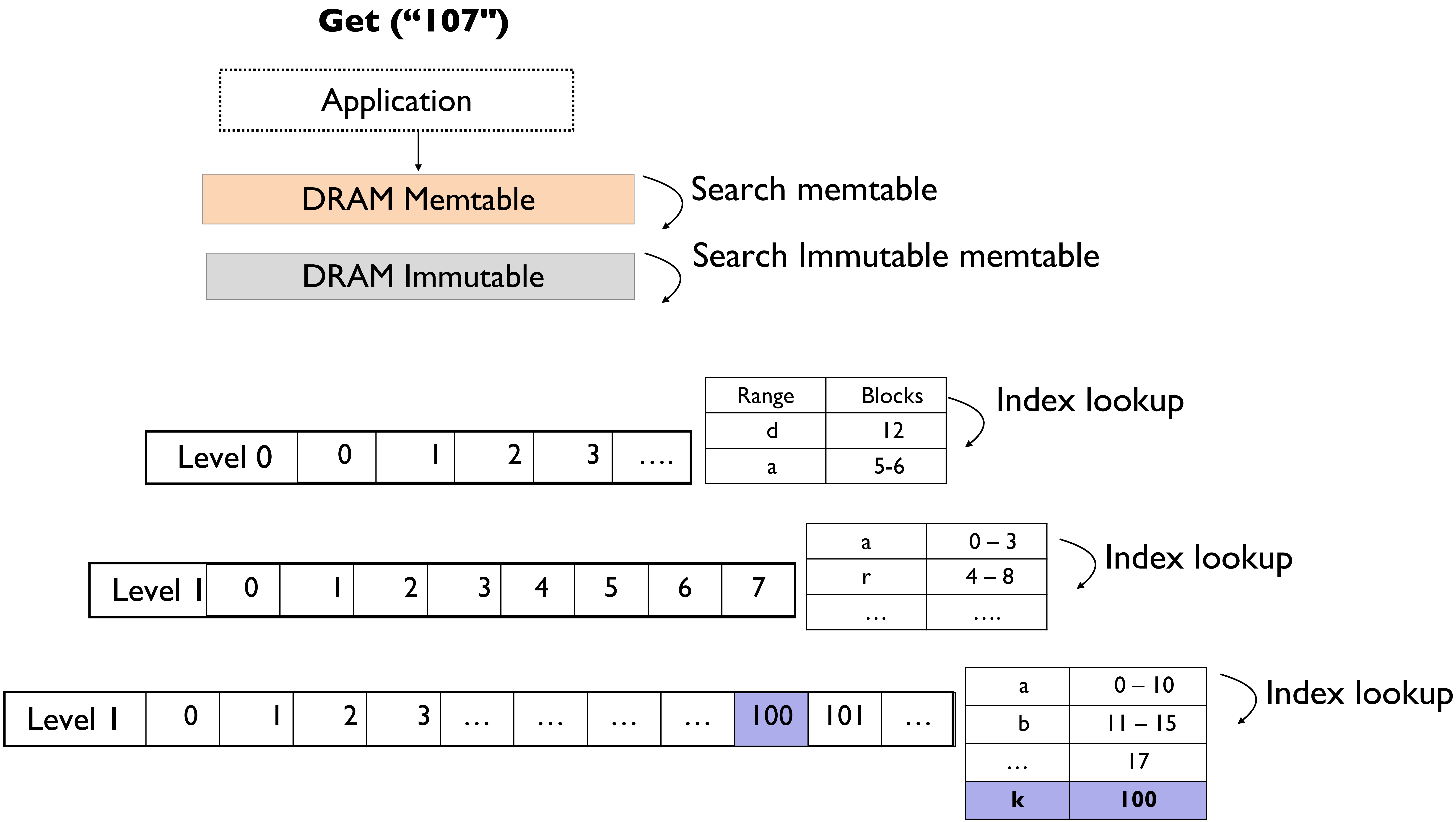
Level 1	0	1	2	3	4	5	6	7

a	0 – 3
r	4 – 8
…	….

Level 1	0	1	2	3	…	…	…	…	100	101	…

a	0 – 10
b	11 – 15
…	17
k	**100**

Read Operation

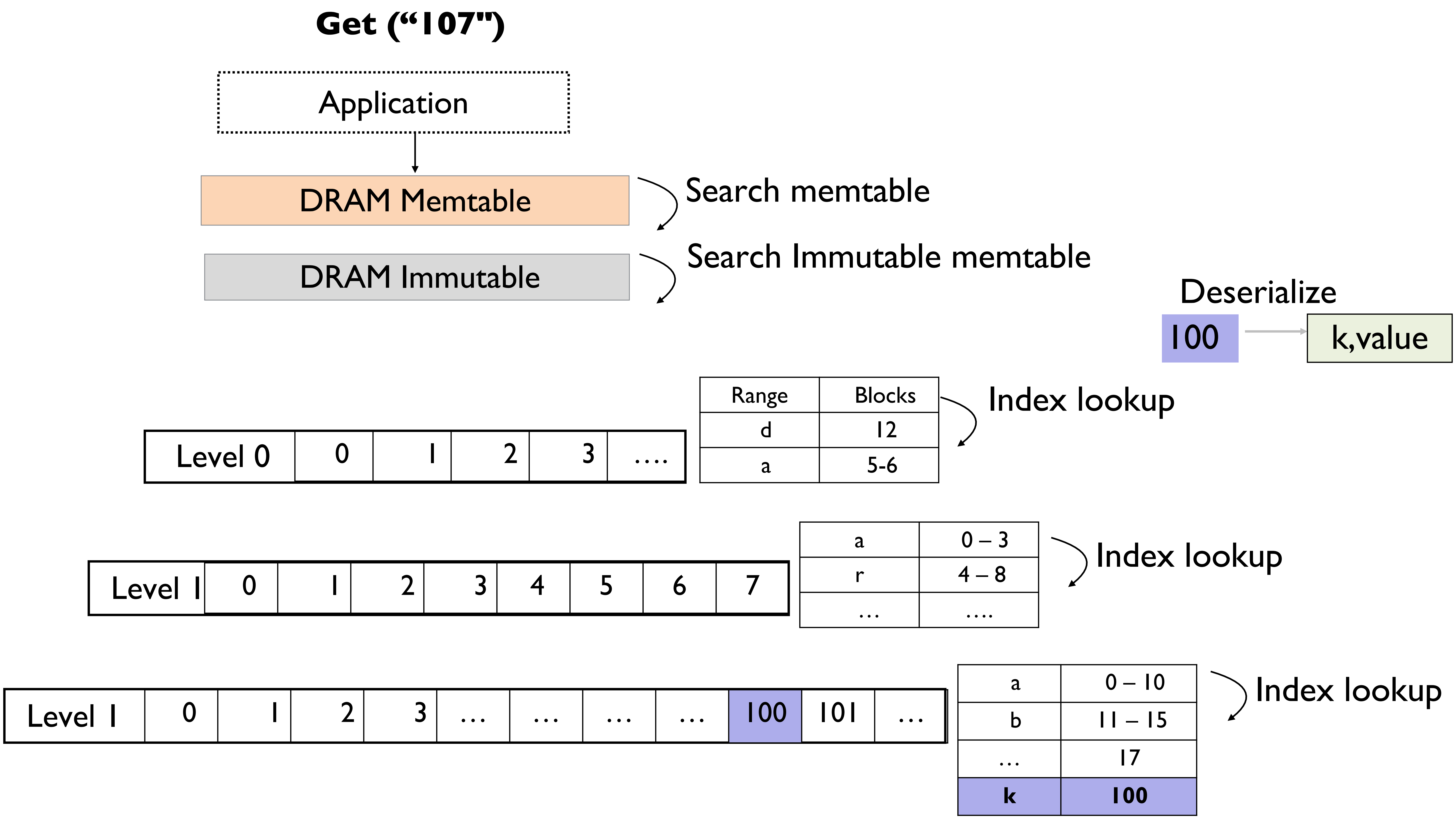


Range	Blocks
d	12
a	5-6

a	0 – 3
r	4 – 8
...	....

a	0 – 10
b	11 – 15
...	17
k	**100**

Read Operation



Outline

Introduction

Background on LevelDB

Motivation

- High serialization, compaction, and logging cost
- Lack of parallelism

NoveLSM Design

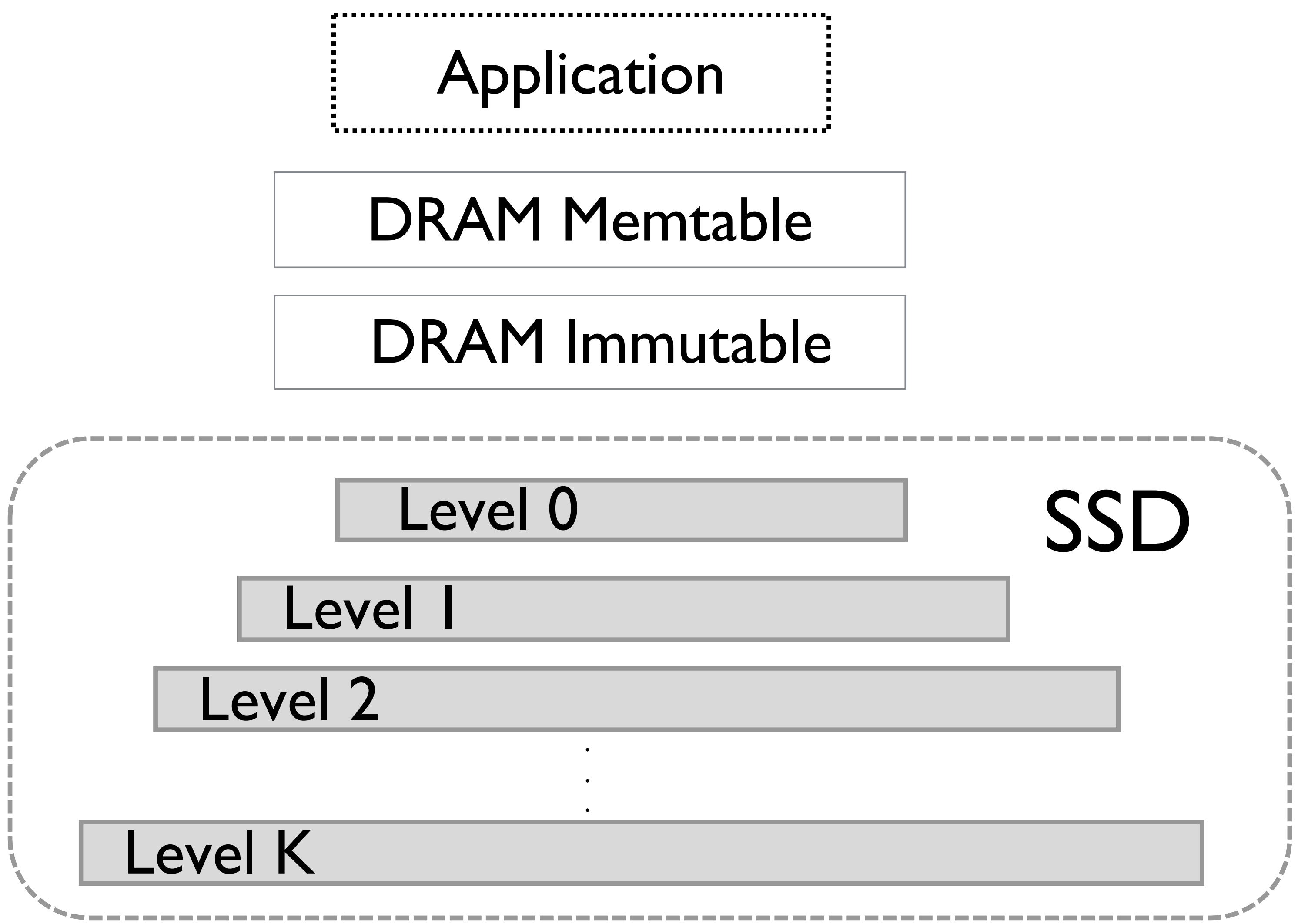
- Persistent memtable, NVM mutability, In-place commits
- Read parallelism

Evaluation

Conclusion

How do LSMs perform on NVM?

LevelDB: Use NVM instead of SSD for storing on-disk SSTable

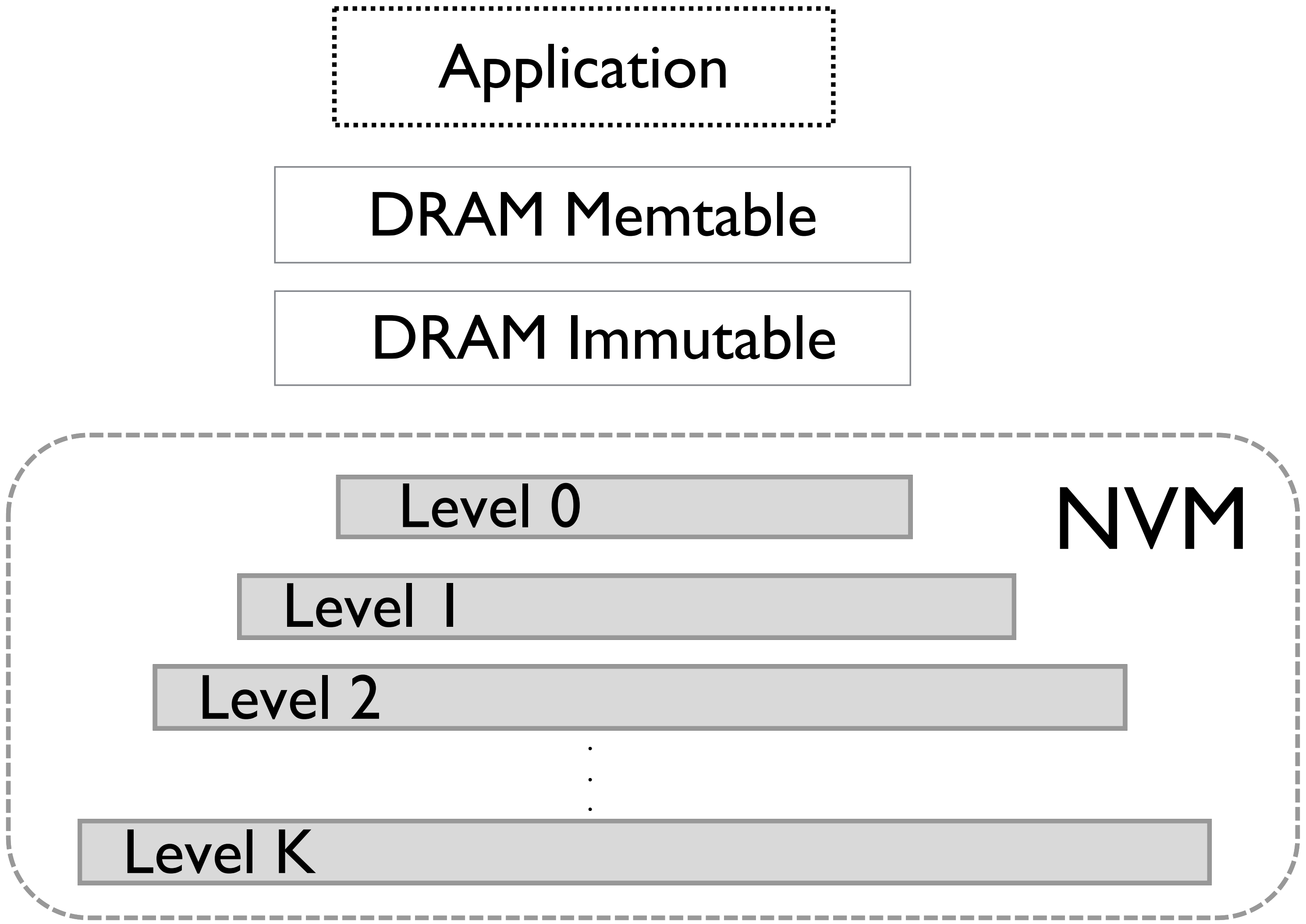


How do LSMs perform on NVM?

LevelDB: Use NVM instead of SSD for storing on-disk SSTable

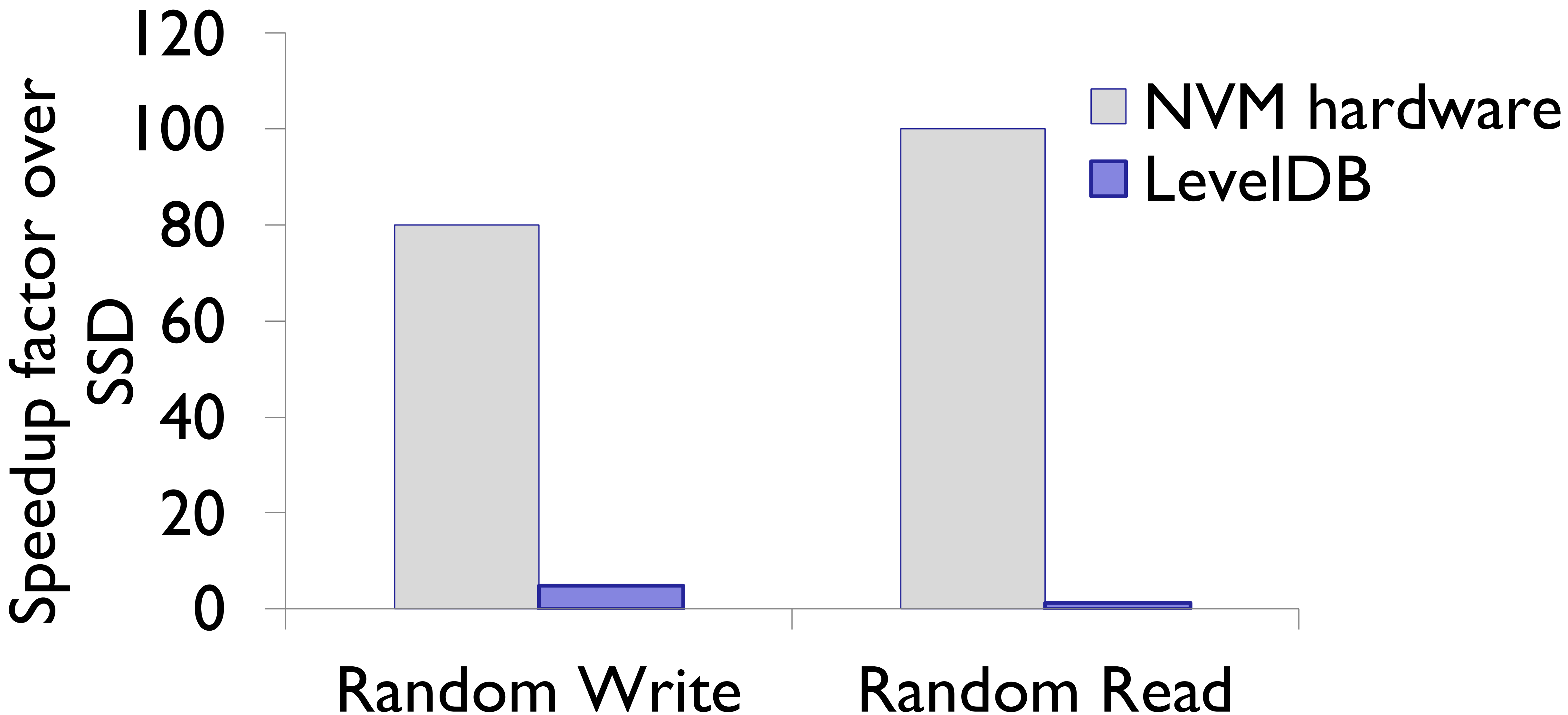
Problem: No byte addressable commercial NVM

- Use DRAM and increase latency by 5x (delay writes)
- Use thermal throttling to reduce NVM bandwidth



NVM Gains when Replacing SSD

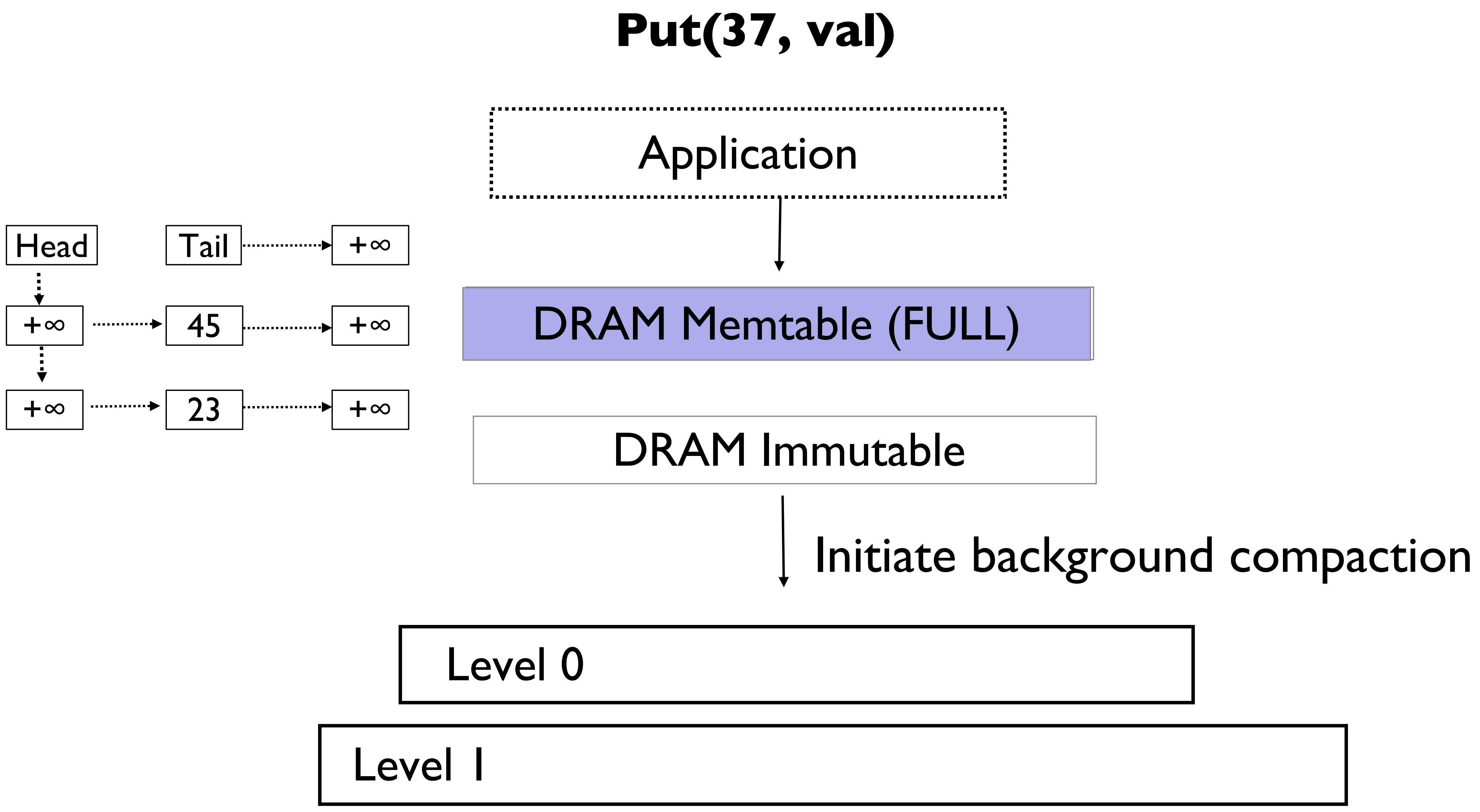
Analyze with 4 KB value size and 16 GB total data size



Random write gains only 4x even with 80x faster NVM

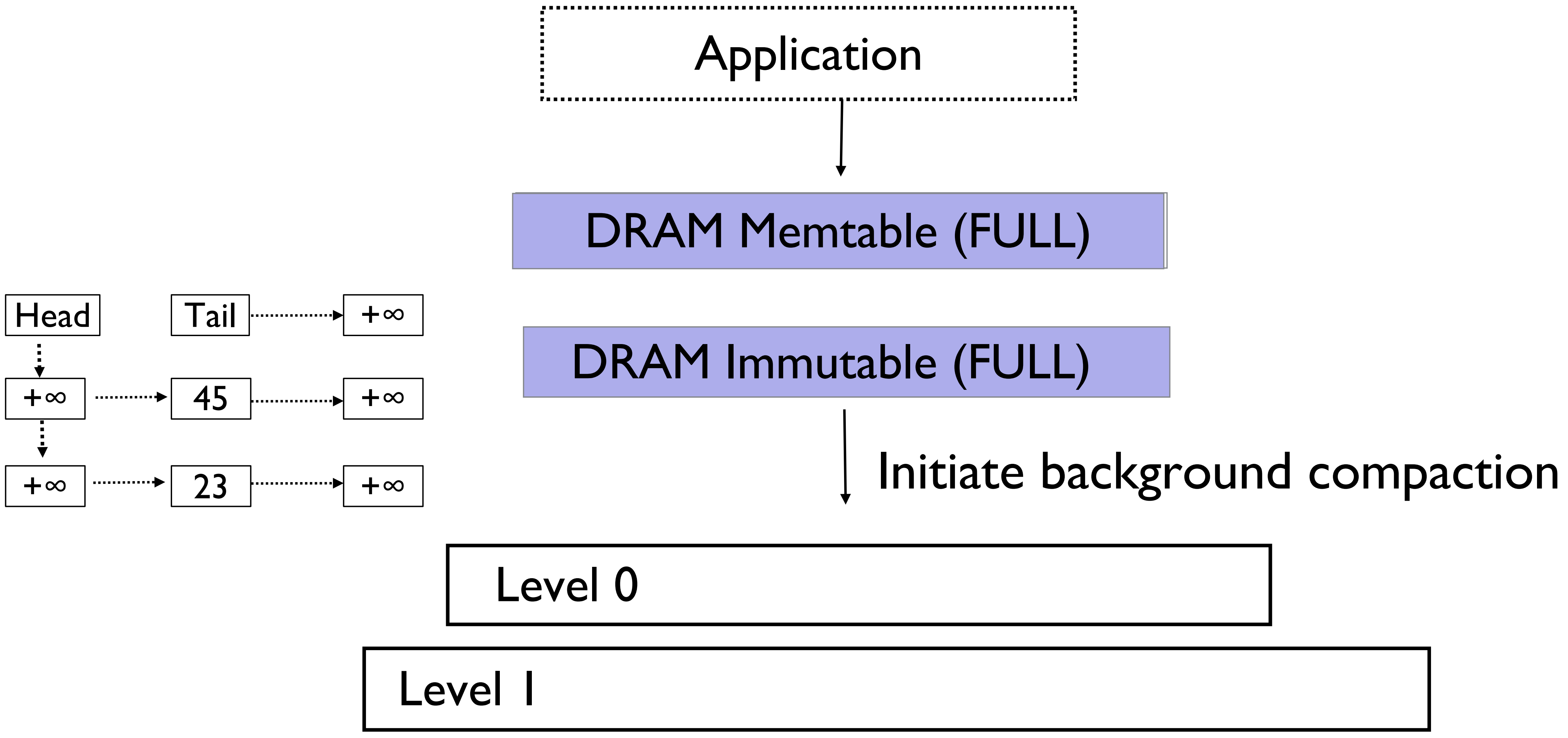
Read latency gains less than 1.5x

1. High (De)Serialization Cost



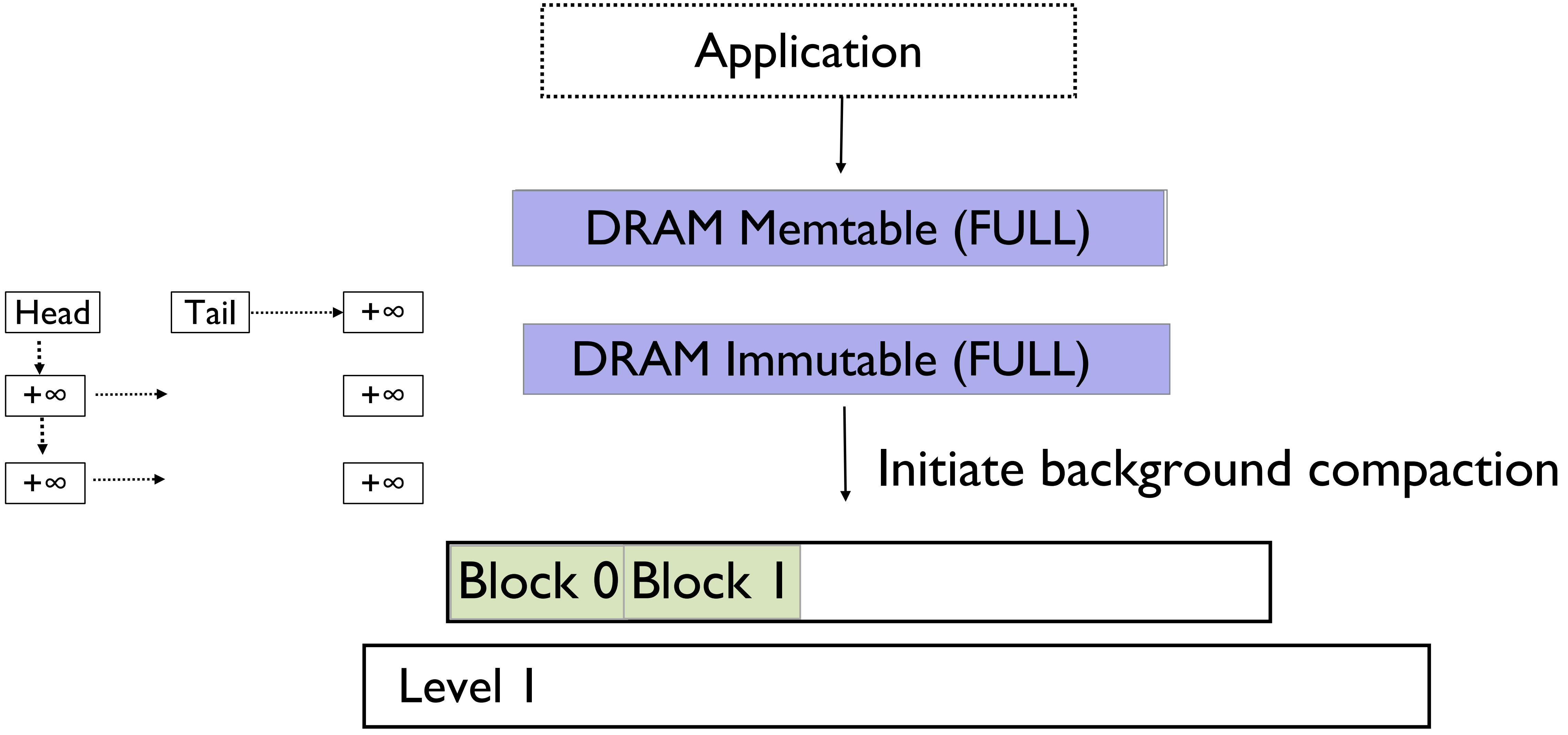
1. High (De)Serialization Cost

Put(37, val)



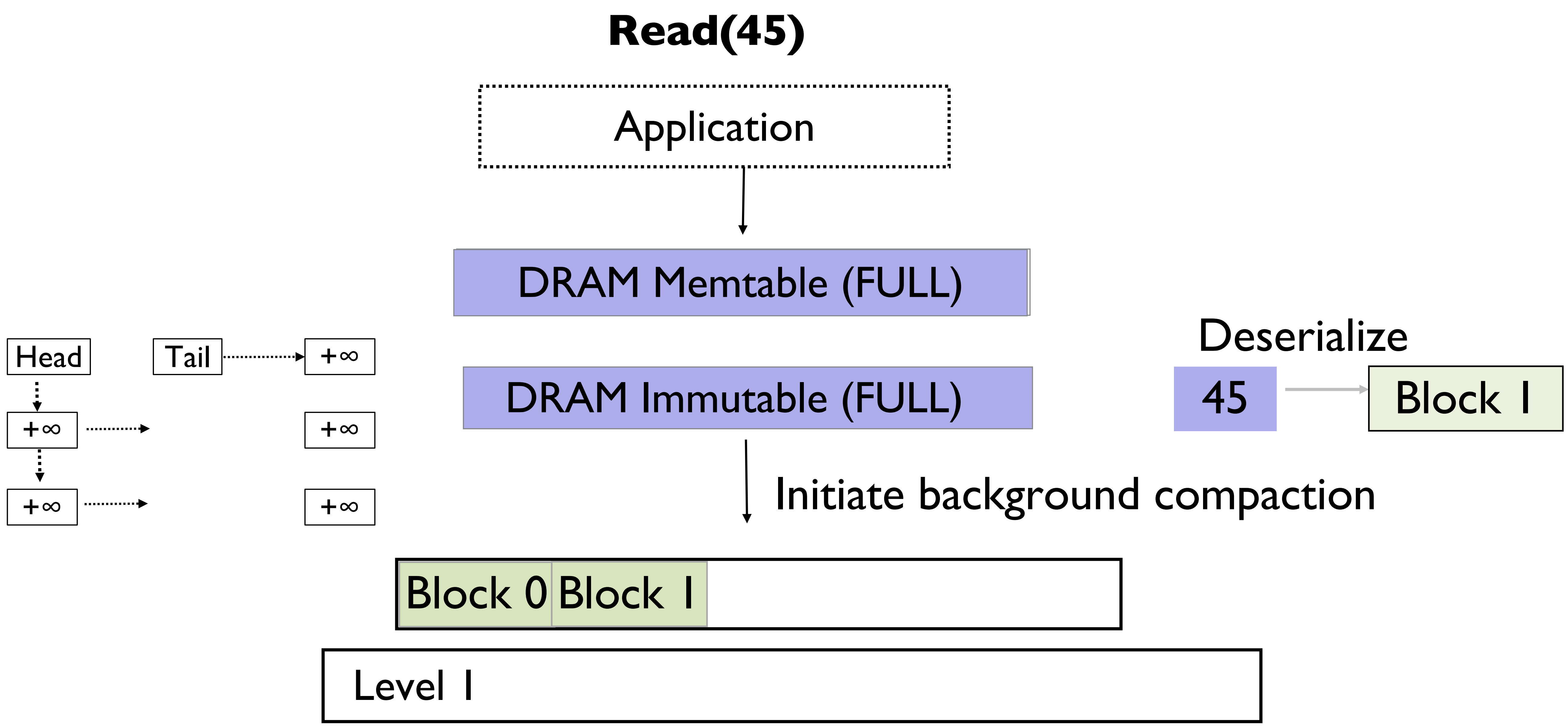
1. High (De)Serialization Cost

Put(37, val)



Serialization of in-memory data to SSTable storage blocks

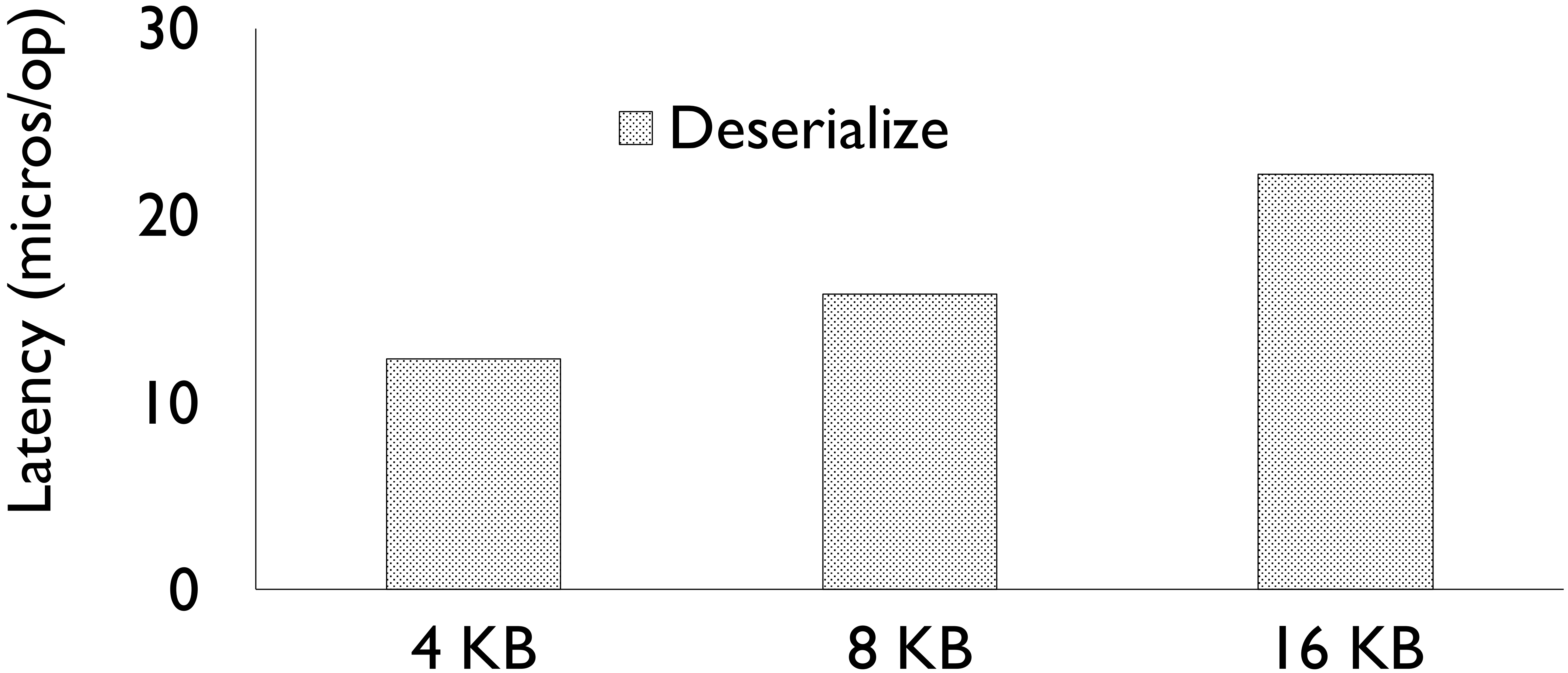
1. High (De)Serialization Cost



Serialization of in-memory data to SSTable storage blocks

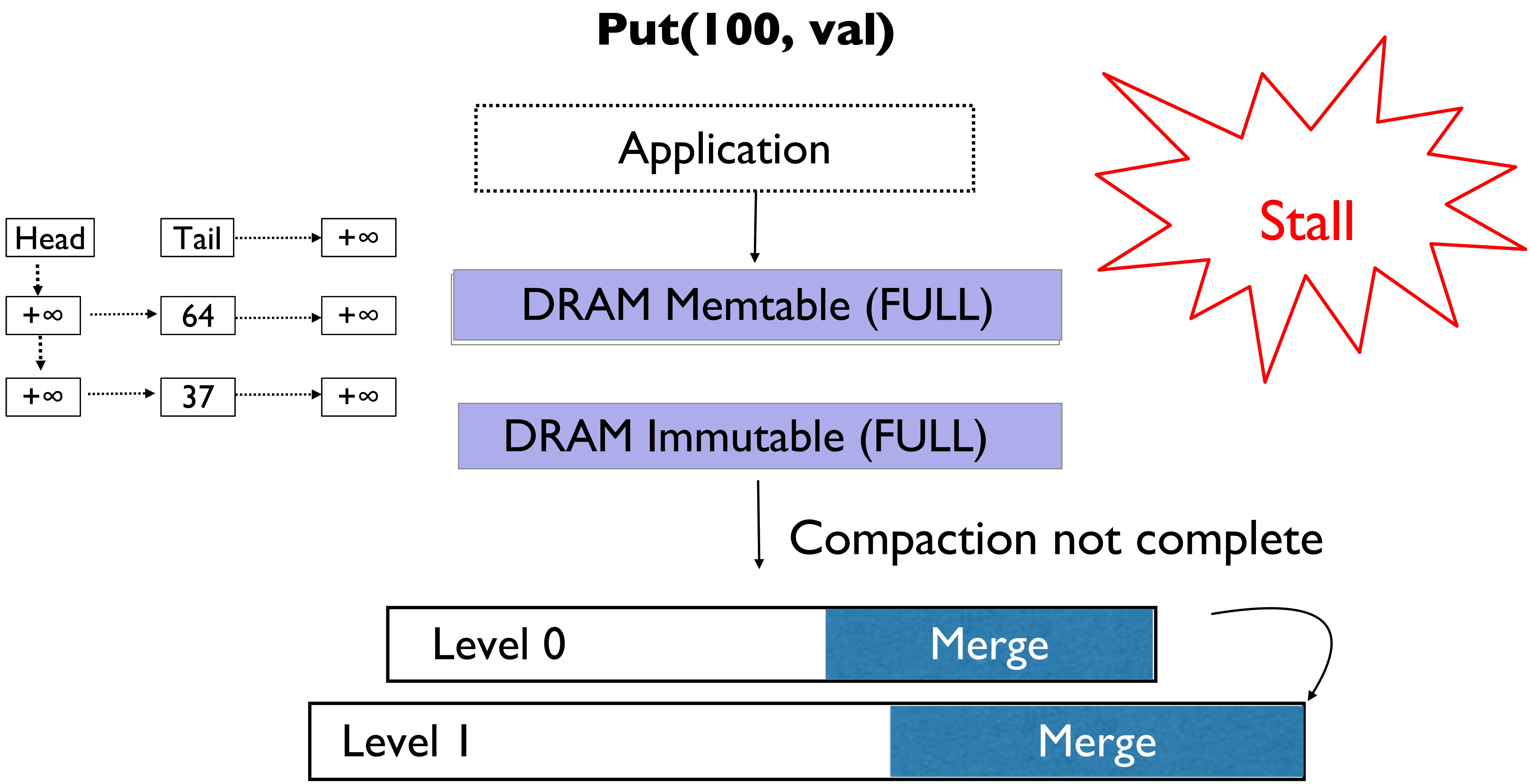
Deserialization of block data to in-memory data during read

1. Deserialization Cost – Read Operation



Deserialization and its related data copy cost increases with value size

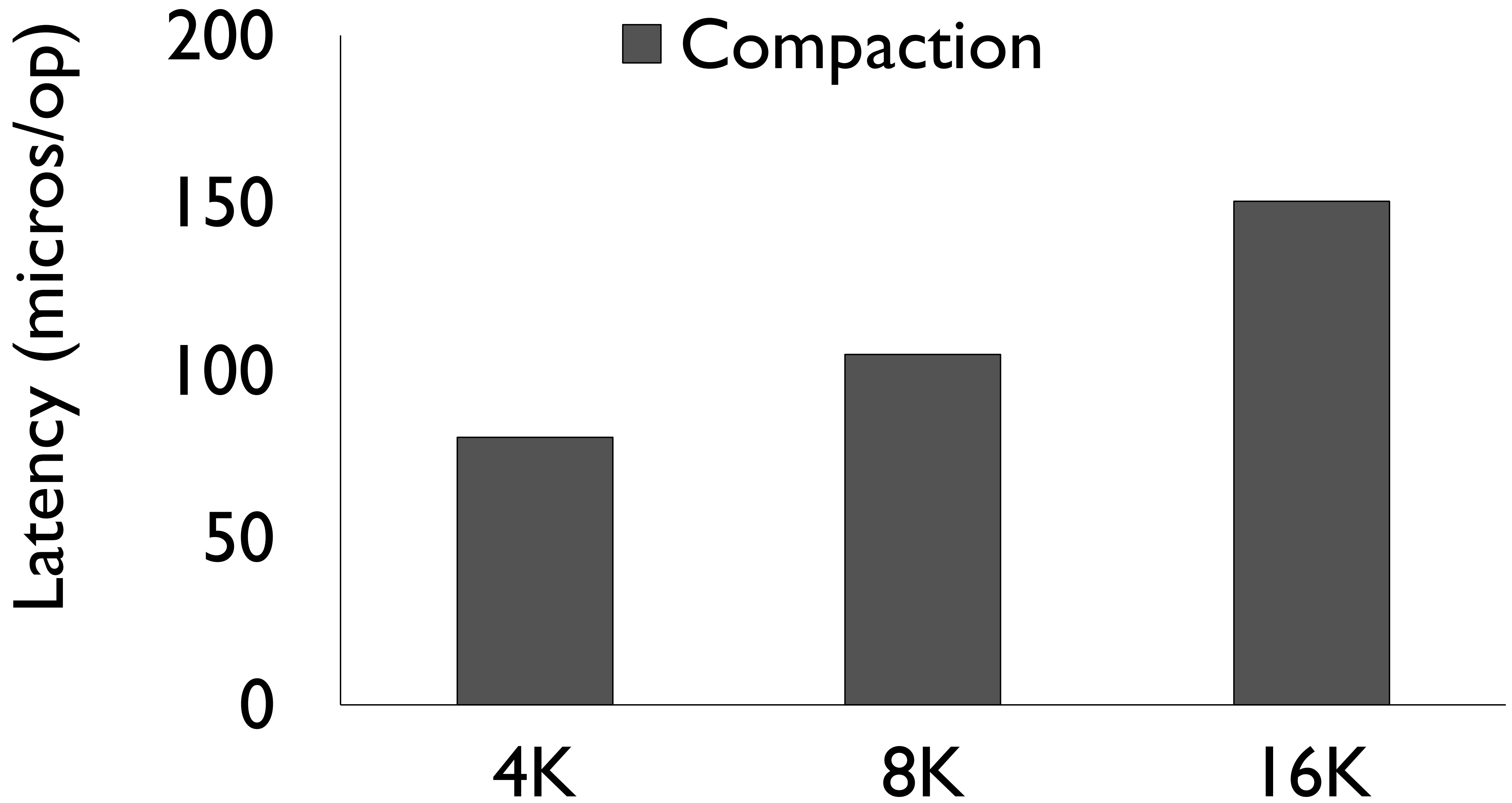
2. High Write Compaction Cost



Compaction time consuming and high overhead

- In-memory structures must be serialized to block format
- Can trigger chain compactions across lower levels

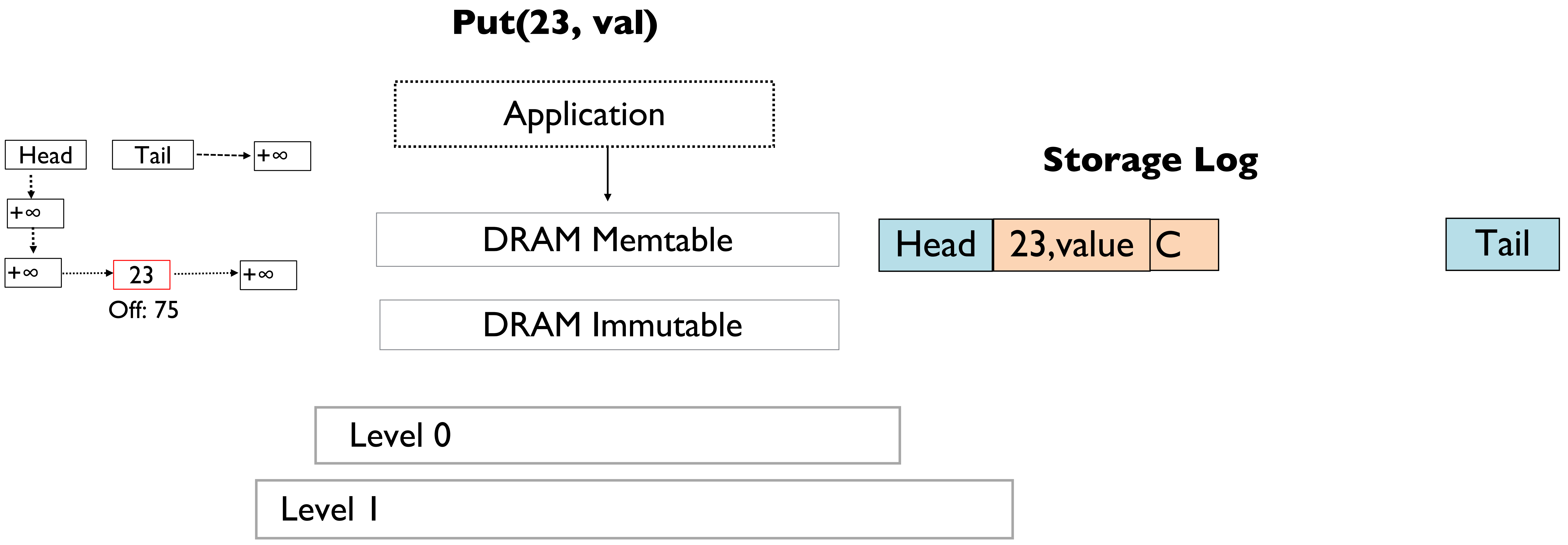
2. High Write Compaction Cost



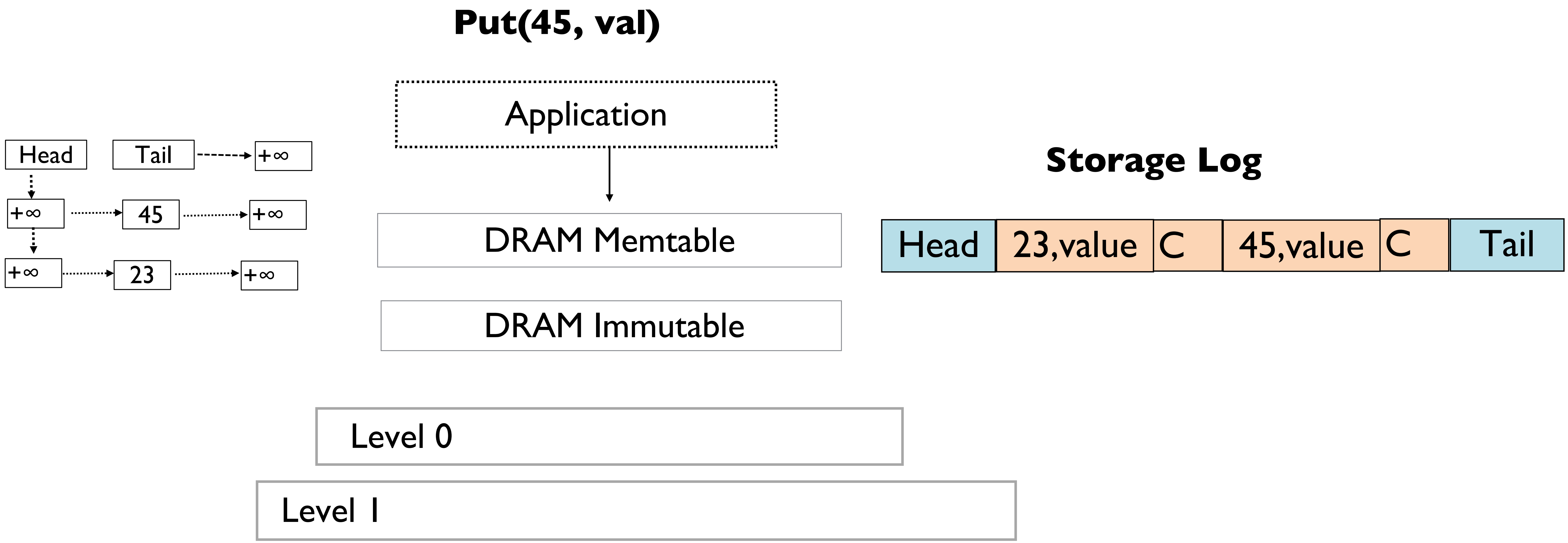
Compaction cost increases with value size

50% - 88% spent just waiting on compaction stall

3. High Write Logging Cost



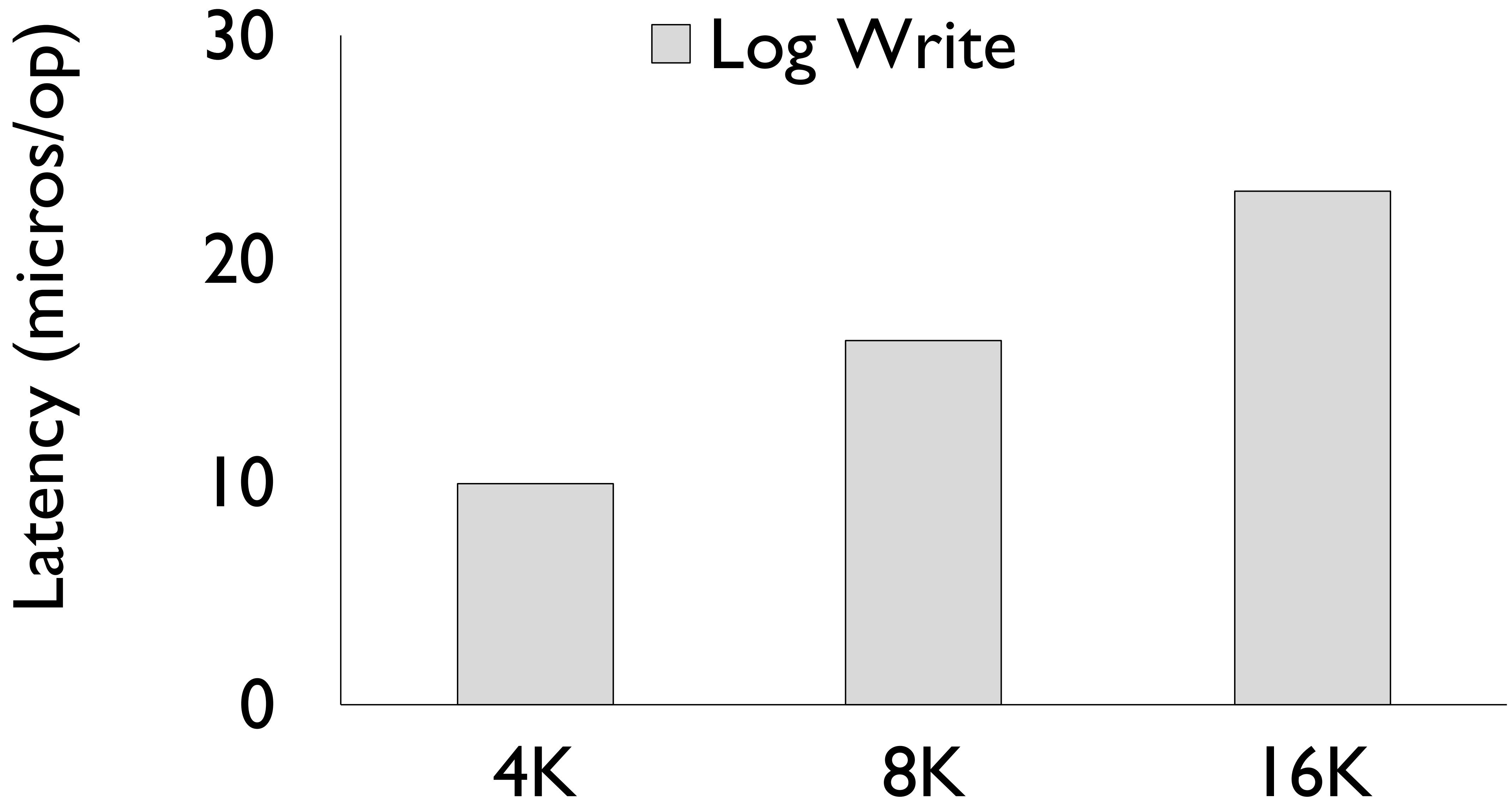
3. High Write Logging Cost



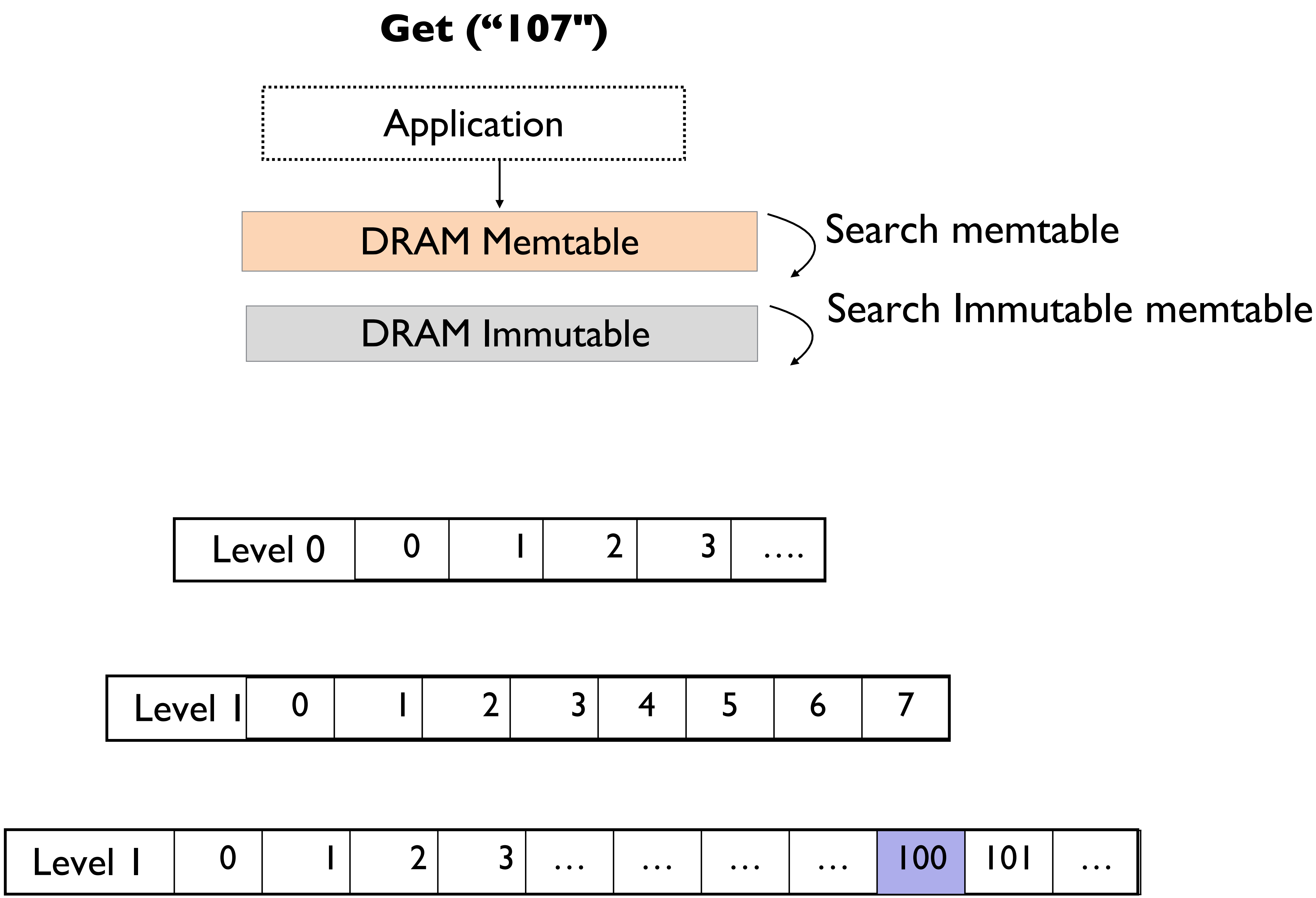
Amplification: LSM updates are written to log, memtable, and SSTable

- LevelDB does not sync log updates for performance
- Log updates are appended with a checksum

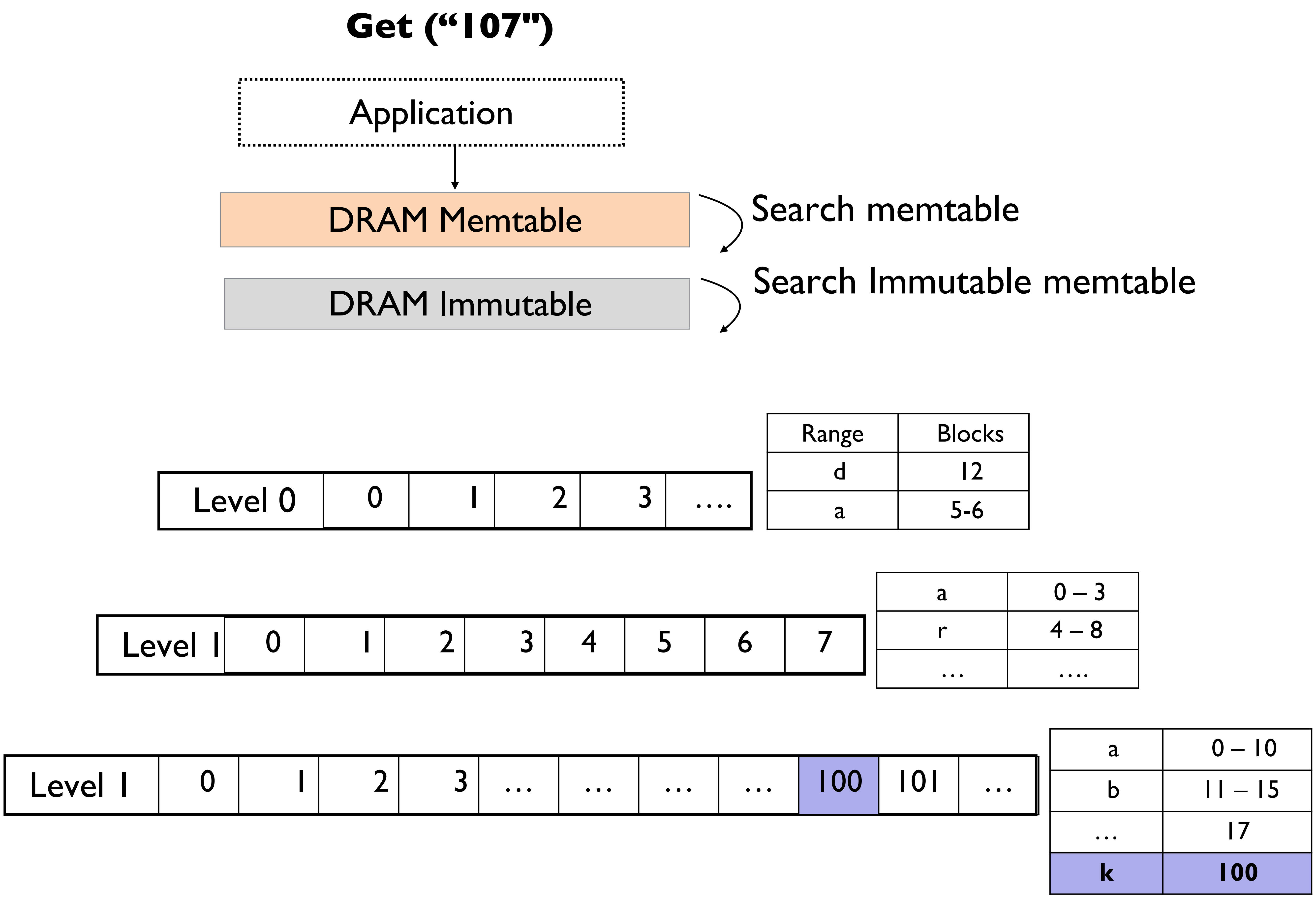
3. High Write Logging Cost



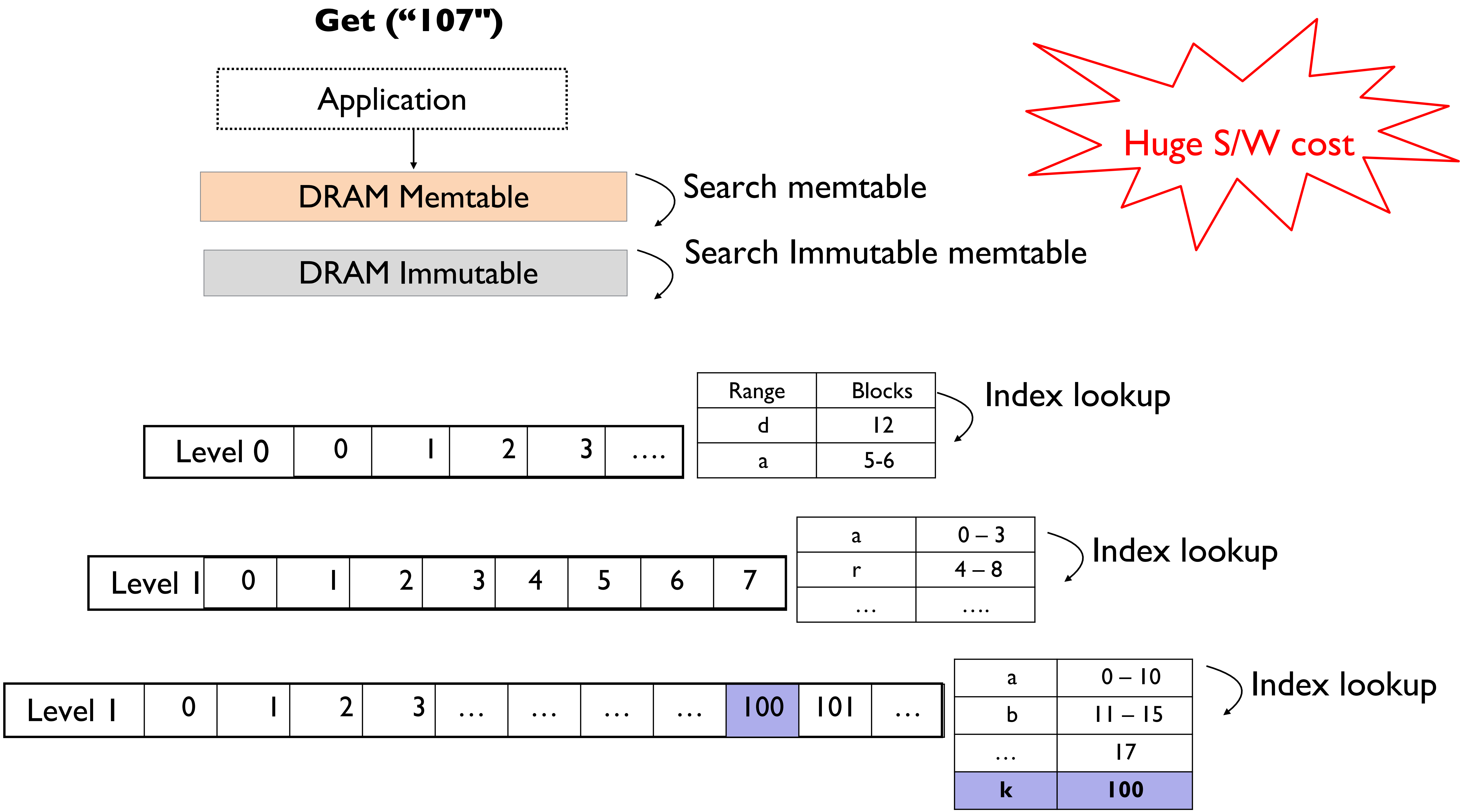
4. Lack of Parallelism – Sequential Reads



4. Lack of Parallelism – Sequential Reads



4. Lack of Parallelism – Sequential Reads



Outline

Introduction

Background on LevelDB

Motivation

- High serialization, compaction, and logging cost
- Lack of parallelism

NoveLSM Design

- Persistent memtable, NVM mutability, In-place commits
- Read parallelism

Evaluation

Conclusion

NonVolatile Memory LSM (NoveLSM)

Reduce serialization – NVM memtable designed with persistent skip list

Reduce compaction – Enable direct mutability on NVM

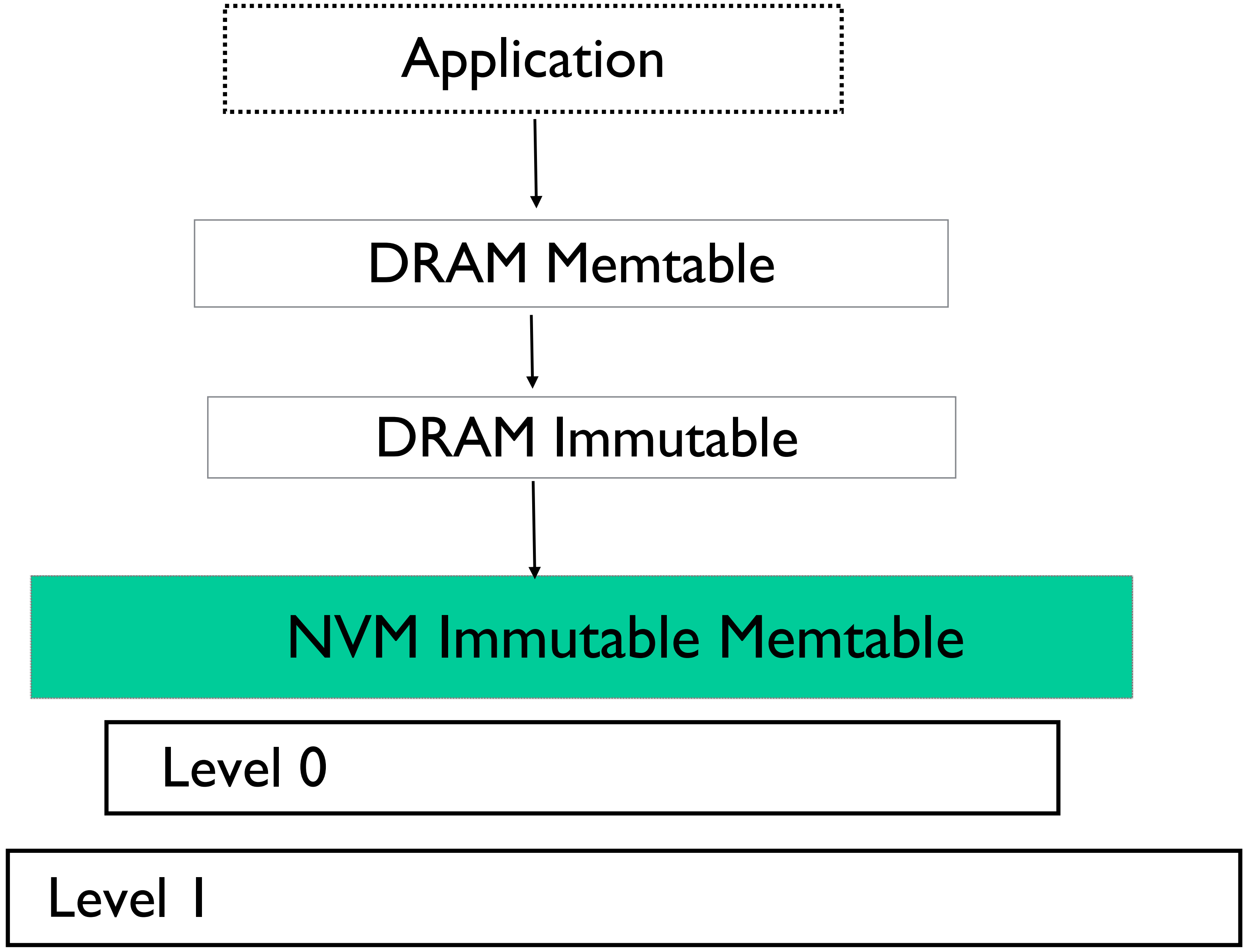
Reduce logging cost – In-place transactional commits to NVM memtable

Improve read parallelism – Read LSM levels in parallel

1. Reduce Serialization: Immutable NVM

High DRAM memtable to storage SSTable serialization cost

Idea: Introduce byte-addressable persistent NVM skip list

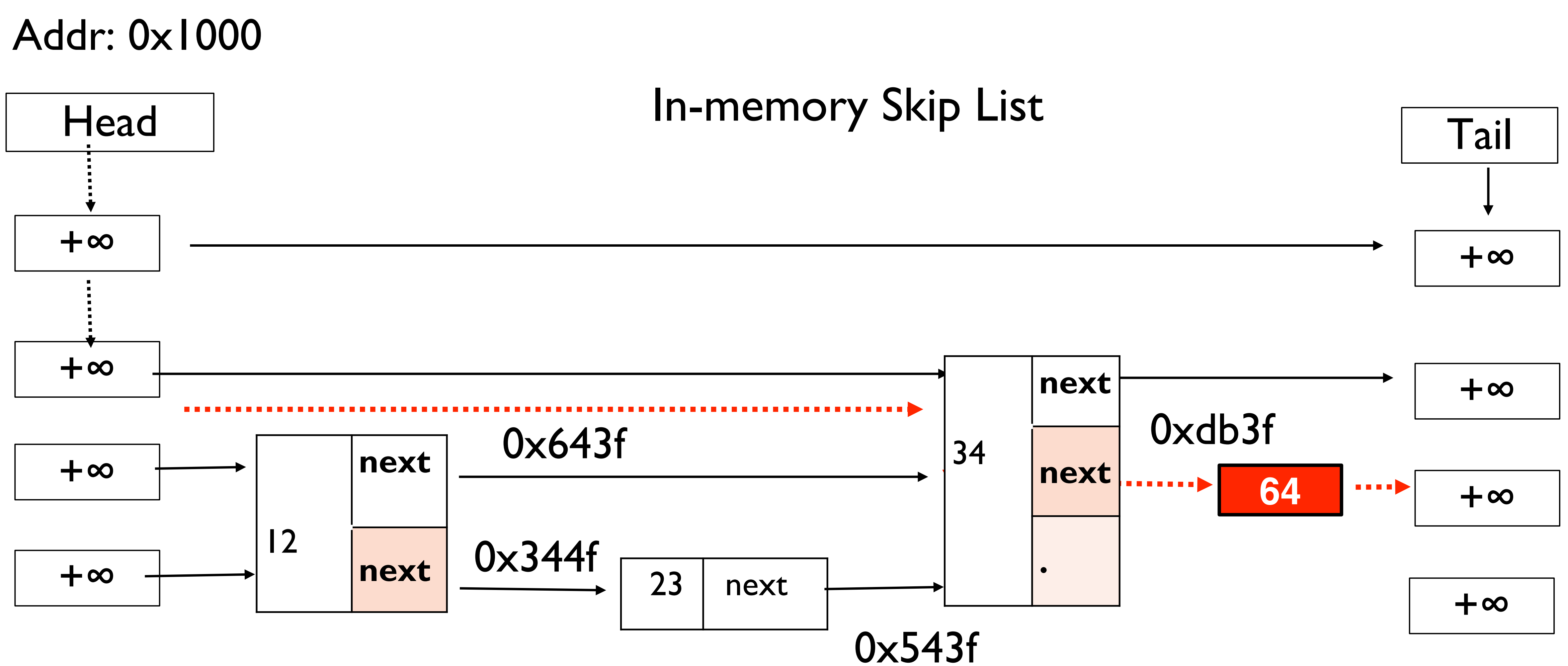


Immutable Memtable: Persistent Skip List

Skip lists - non-persistent structures with fast probabilistic writes and read

Our goal: make skip lists persistent for exploiting NVM byte-addressability

Insert ("64", val)



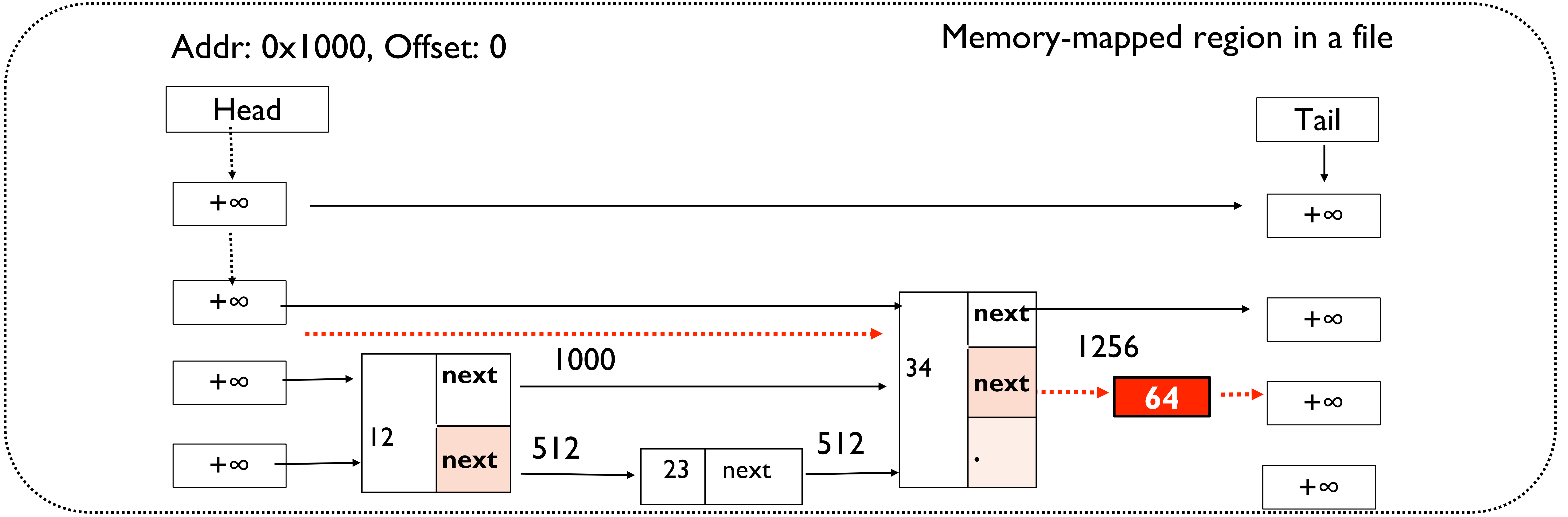
Designing Persistent Skip List

Persistent skip list created by mapping memory from NVM

Uses offset in the mapped memory instead of virtual address

To read/recover, simply get the root offset and traverse using offsets

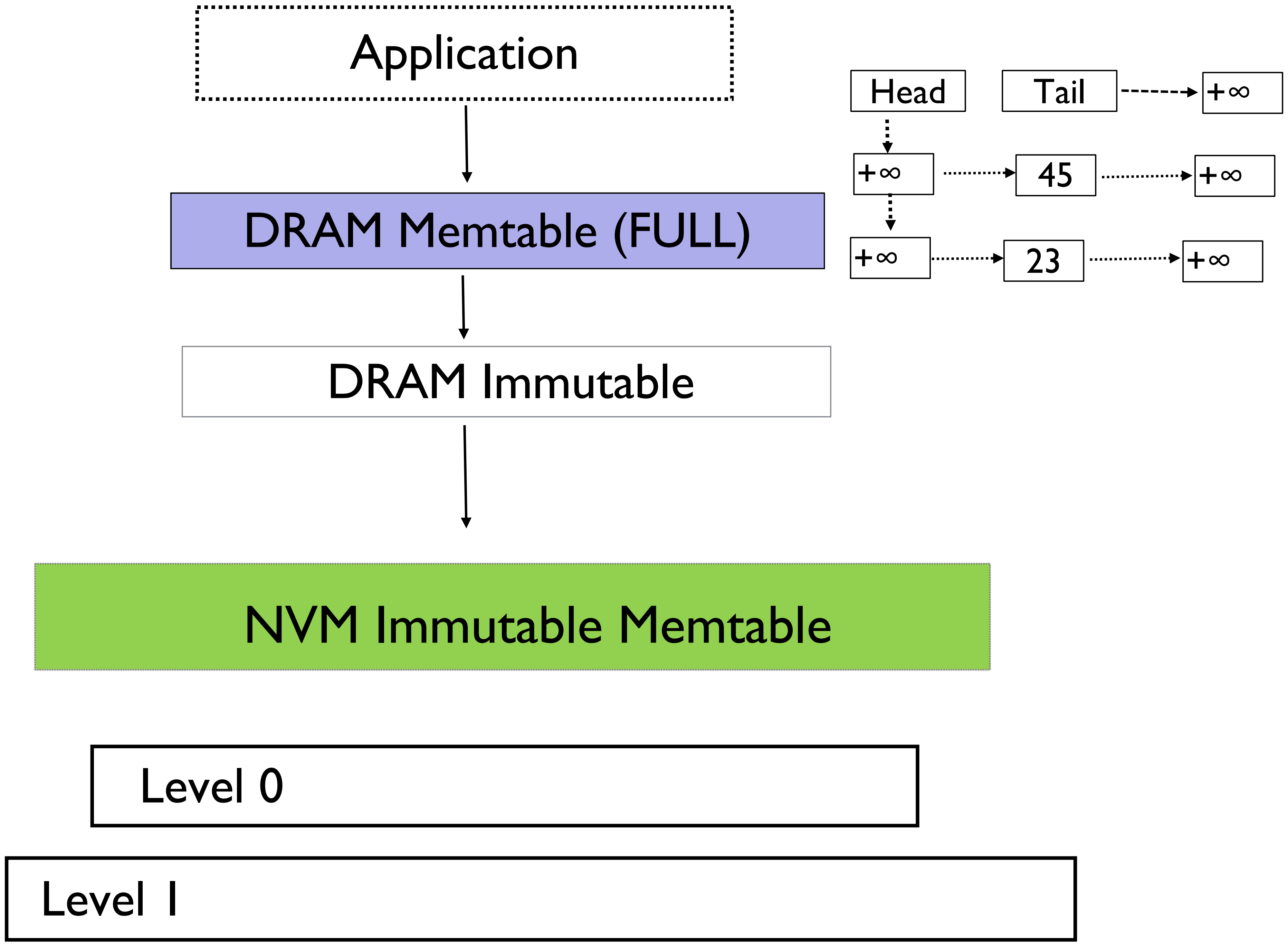
Insert ("64", val)



Immutable NVM Design

Reduce serialization with a immutable persistent skip list

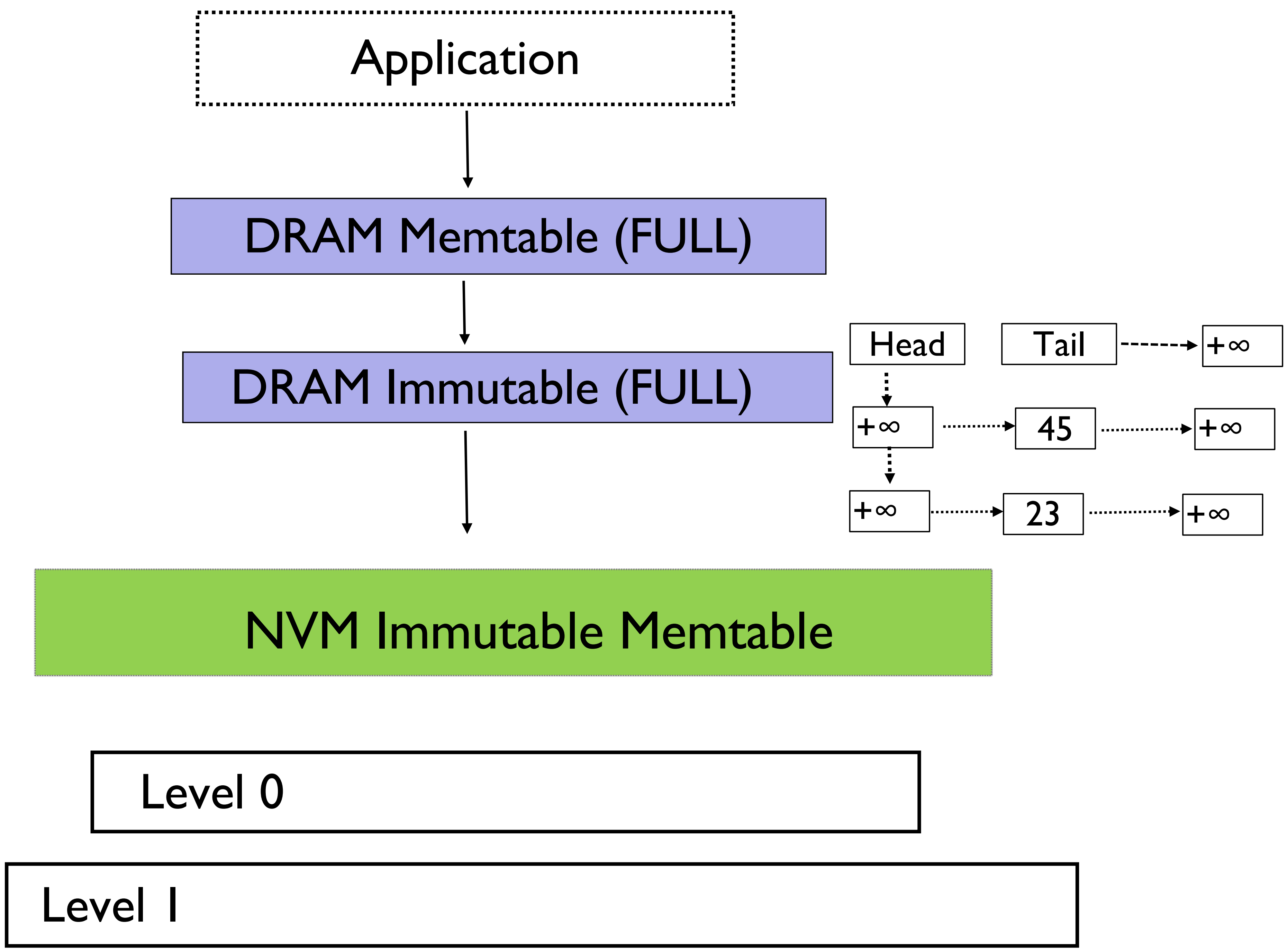
Put(37, val)



Immutable NVM Design

Reduce serialization with a immutable persistent skip list

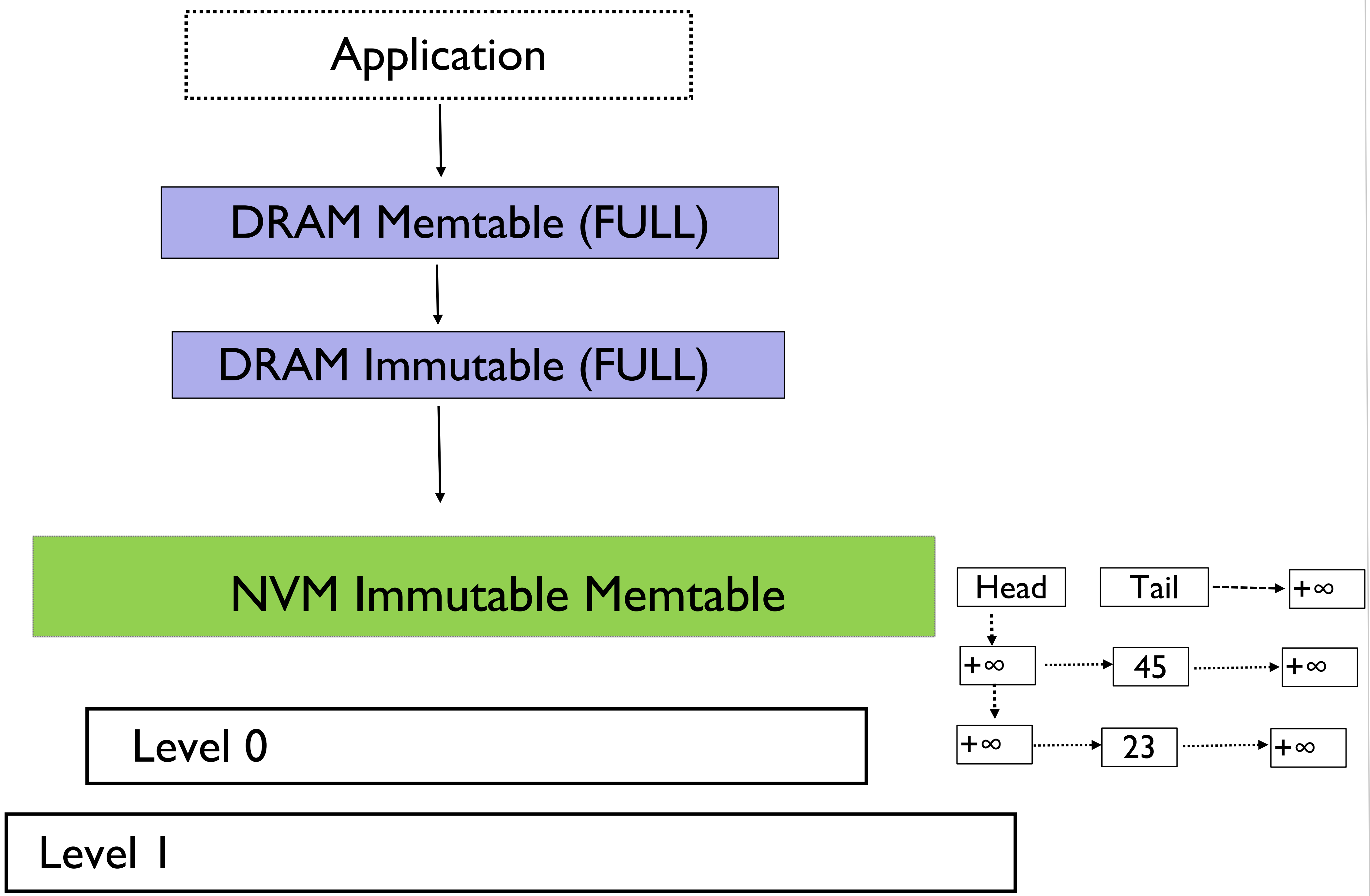
Put(37, val)



Immutable NVM Design

Reduce serialization with a immutable persistent skip list

Put(37, val)

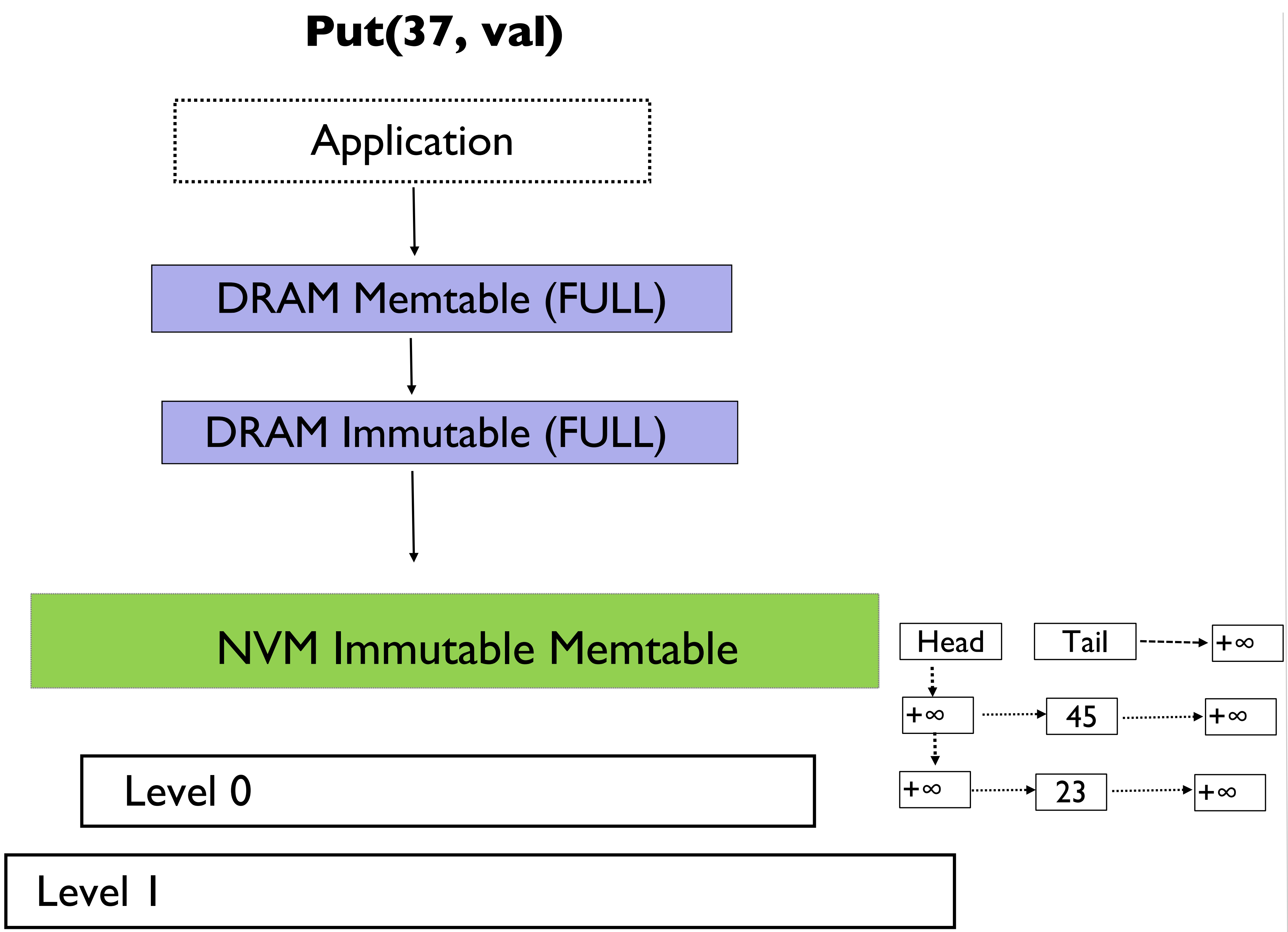


Copy data to large NVM memtable w/o serialization

Reads avoid deserialization

Immutable NVM Design

Reduce serialization with a immutable persistent skip list



Copy data to large NVM memtable w/o serialization

Reads avoid deserialization

Compaction frequency dependent on DRAM memtable size

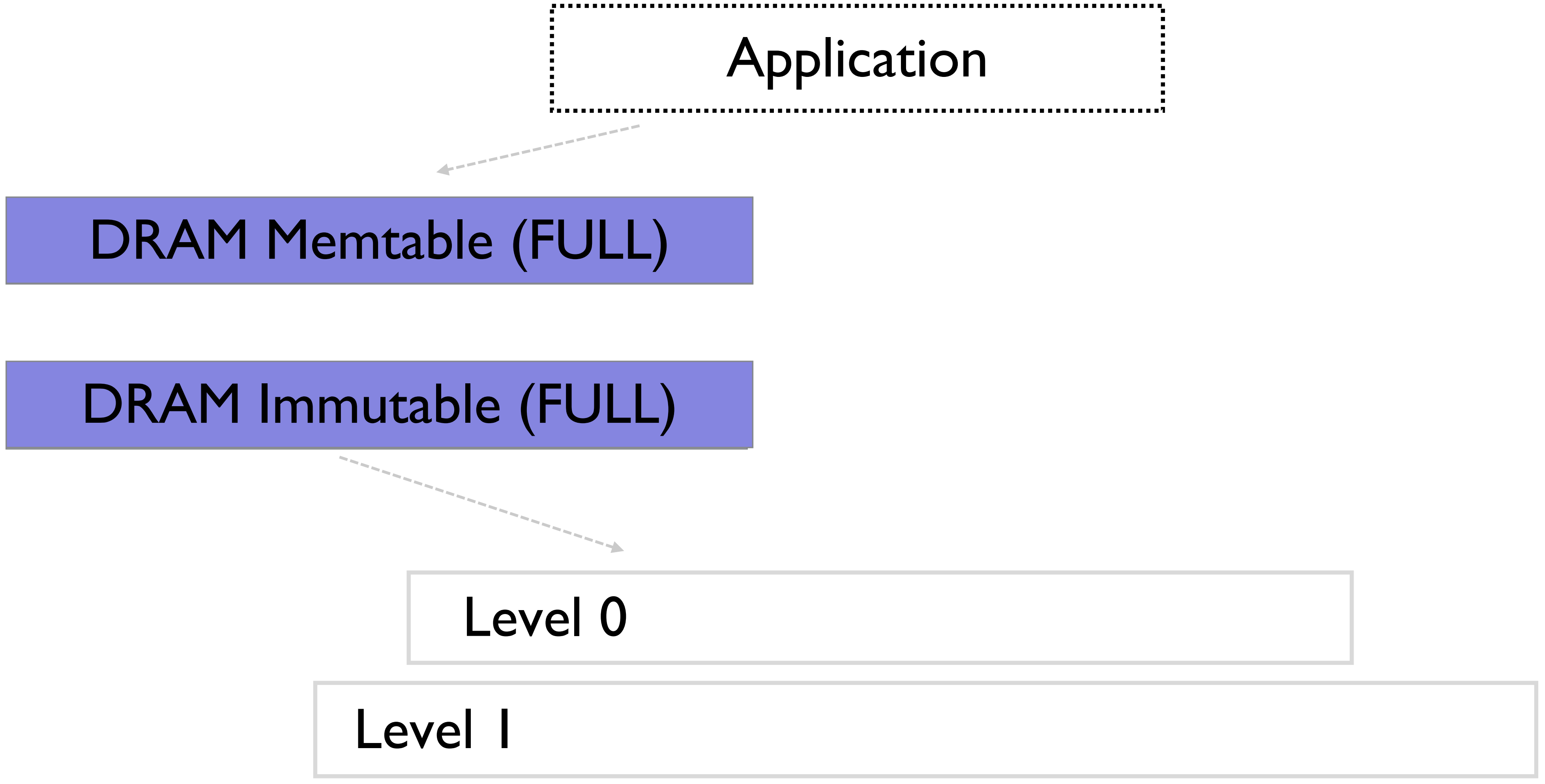
Increasing DRAM buffer increases memory use by 2x

Recovery cost increases

Log not committed - data loss!

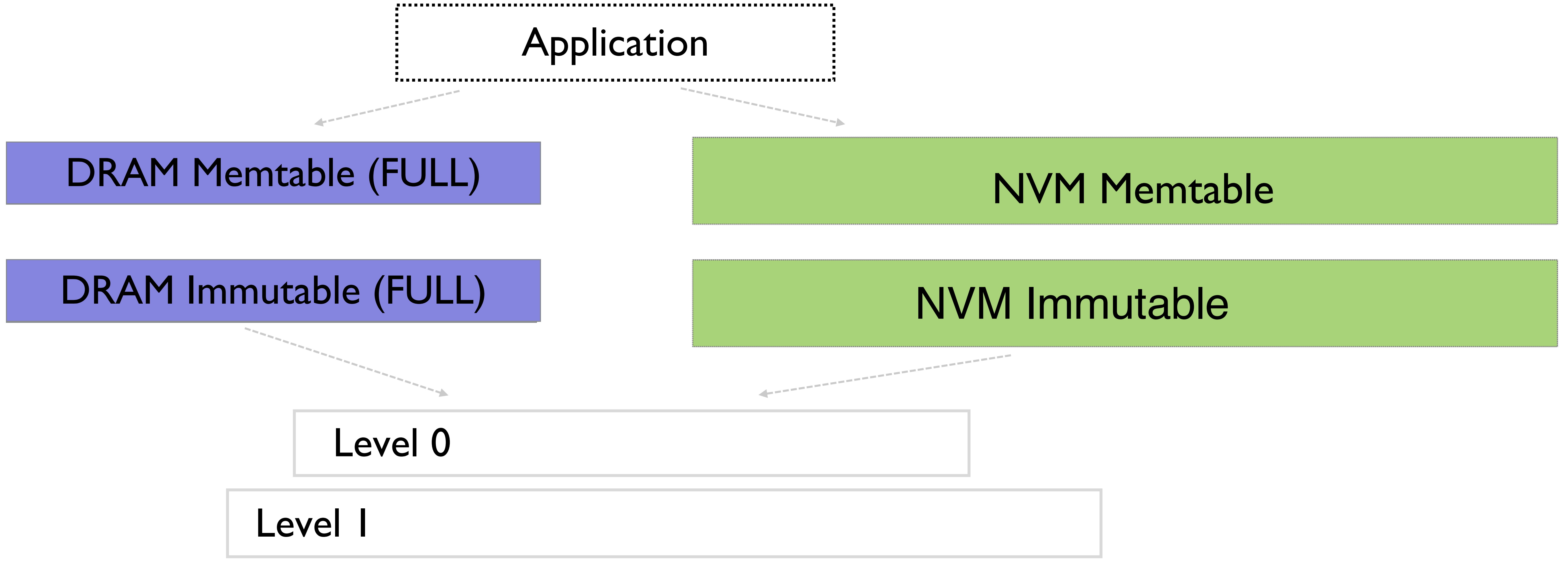
2. Reducing Compaction: NVM Mutability

High compaction cost even with immutable memtable design



2. Reducing Compaction: NVM Mutability

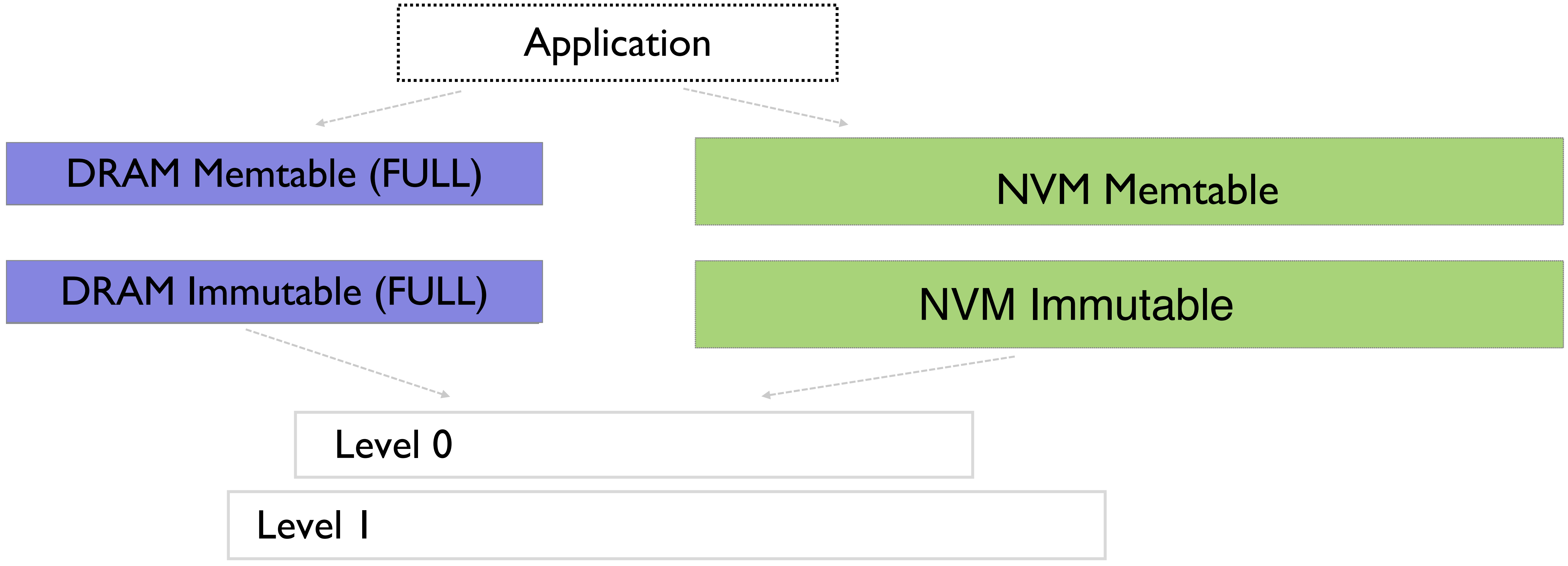
High compaction cost even with immutable memtable design



2. Reducing Compaction: NVM Mutability

High compaction cost even with immutable memtable design

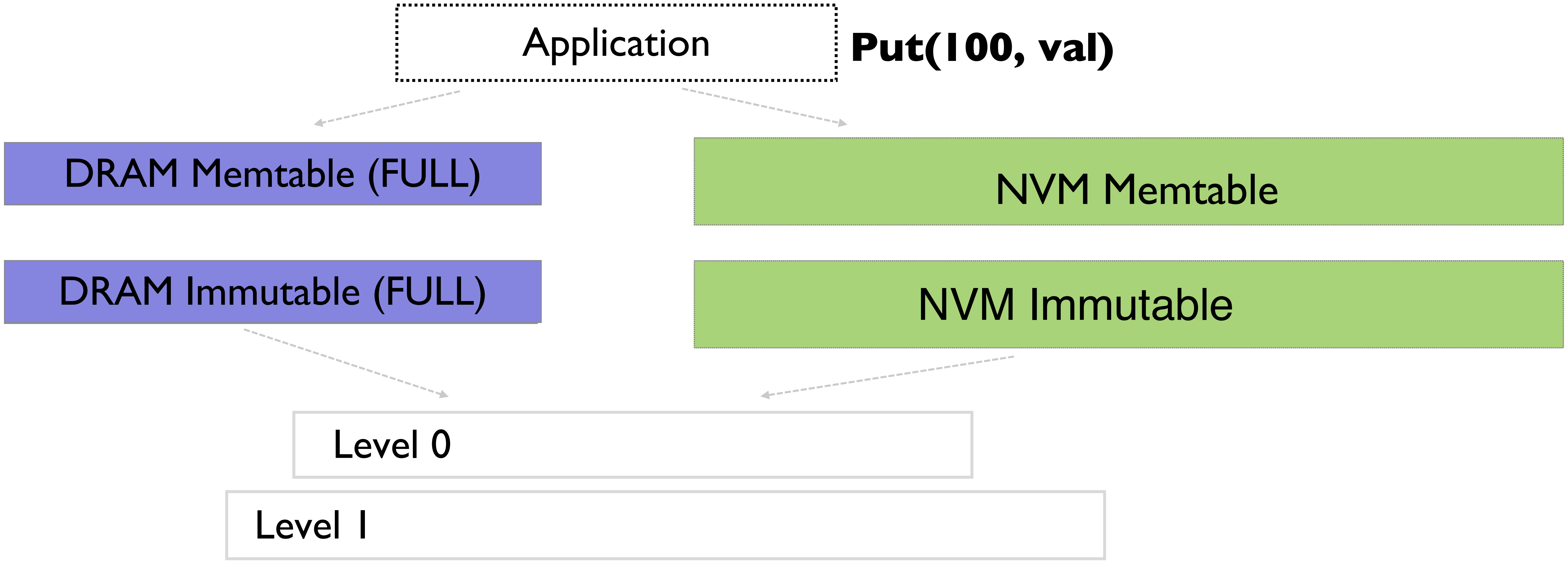
Idea: Exploit byte addressability and directly update NVM memtable



2. Reducing Compaction: NVM Mutability

High compaction cost even with immutable memtable design

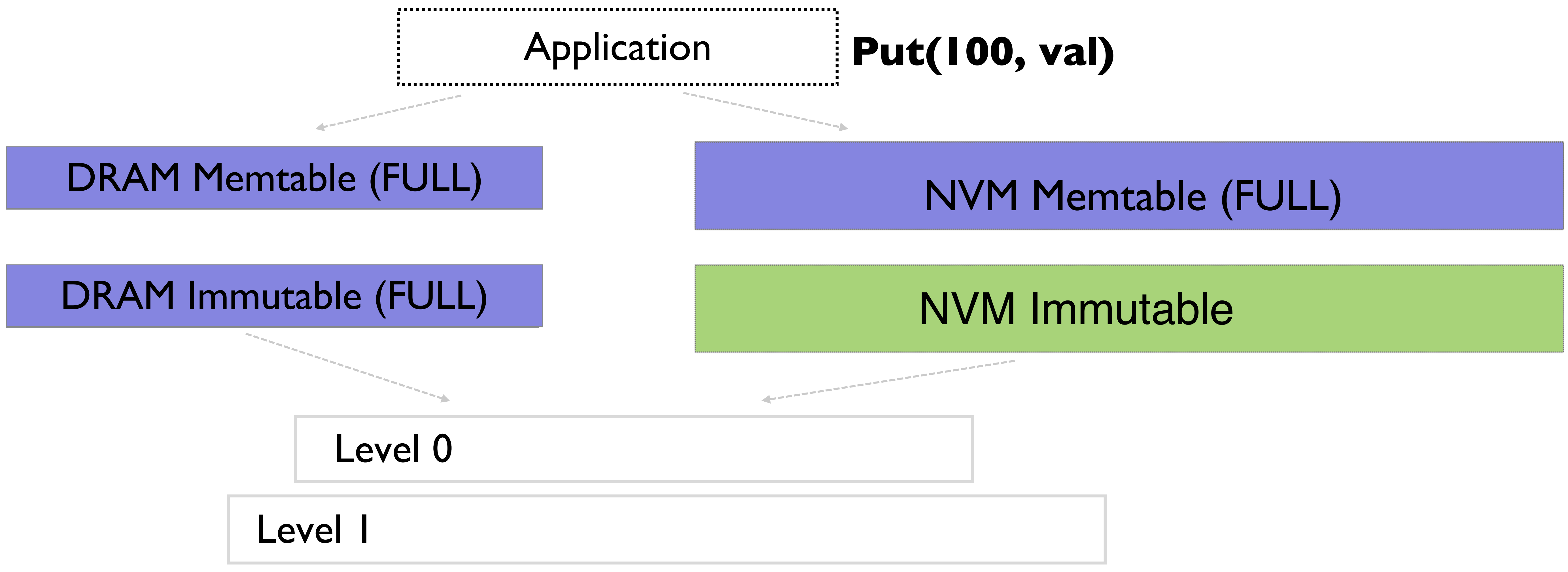
Idea: Exploit byte addressability and directly update NVM memtable



2. Reducing Compaction: NVM Mutability

High compaction cost even with immutable memtable design

Idea: Exploit byte addressability and directly update NVM memtable



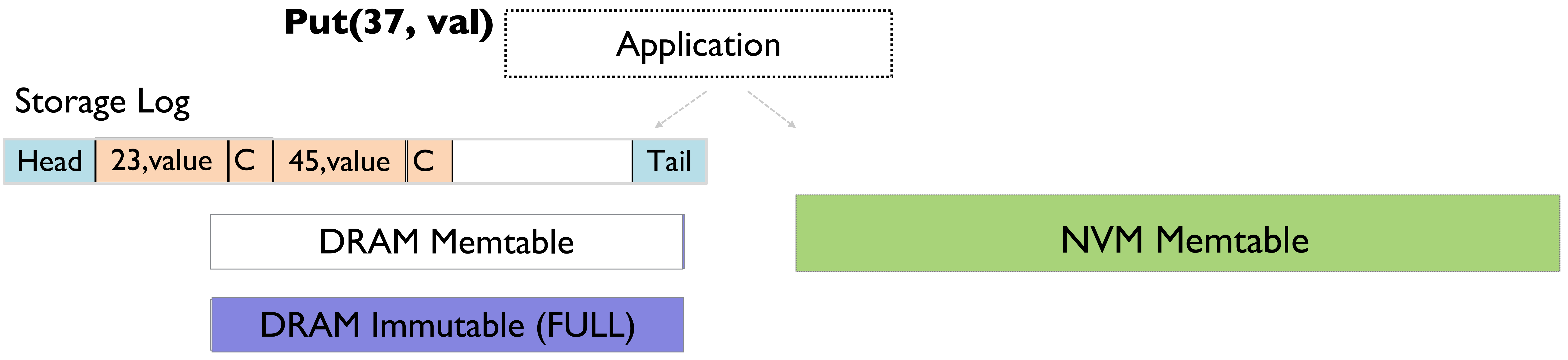
Direct NVM mutability provides sufficient time for DRAM compaction

- Reduces foreground stall

NVM memtable persistent – data not lost after failure

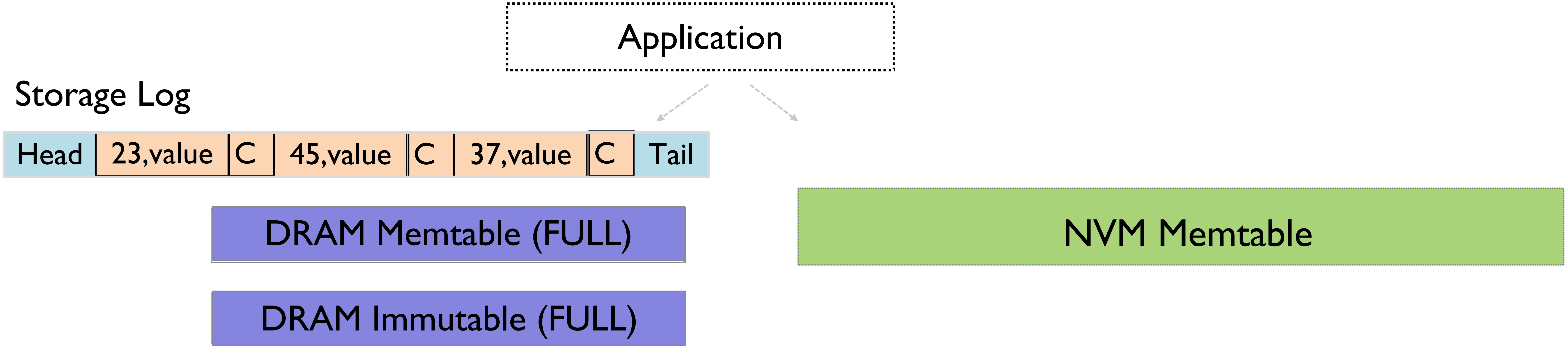
3. Reducing Logging Cost: In-place Commits

Problem: Writing to log before memtable has high overhead



3. Reducing Logging Cost: In-place Commits

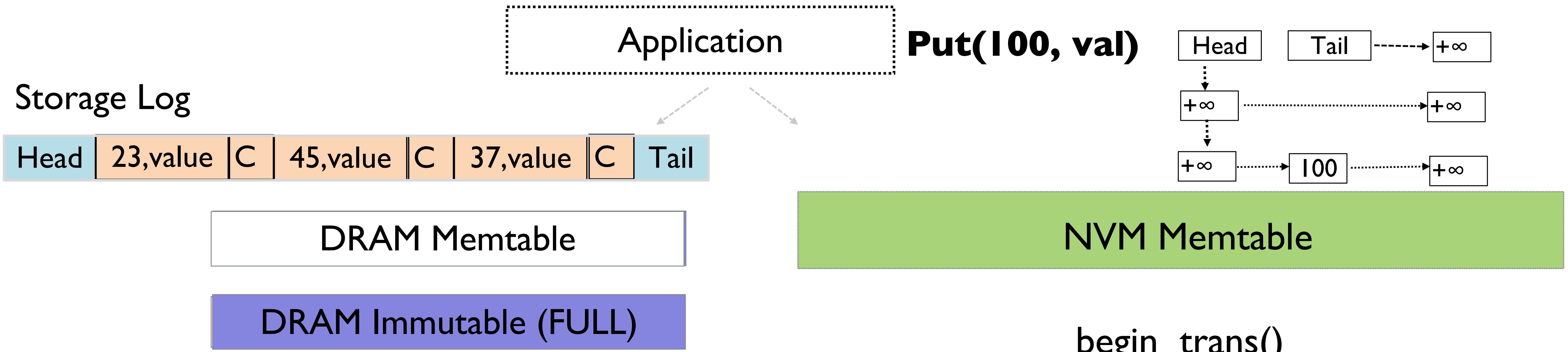
Problem: Writing to log before memtable has high overhead



3. Reducing Logging Cost: In-place Commits

Problem: Writing to log before memtable has high overhead

Idea: Avoid logging for NVM memtable with in-place commits

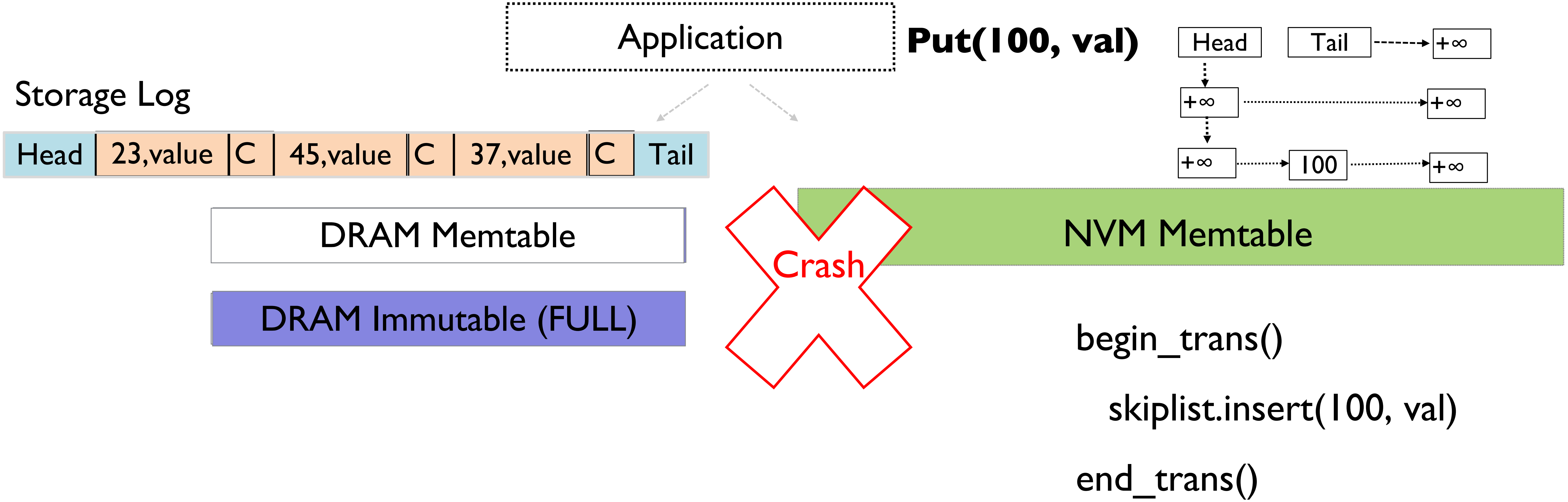



```
begin_trans()
  skiplist.insert(100, val)
end_trans()
```

3. Reducing Logging Cost: In-place Commits

Problem: Writing to log before memtable has high overhead

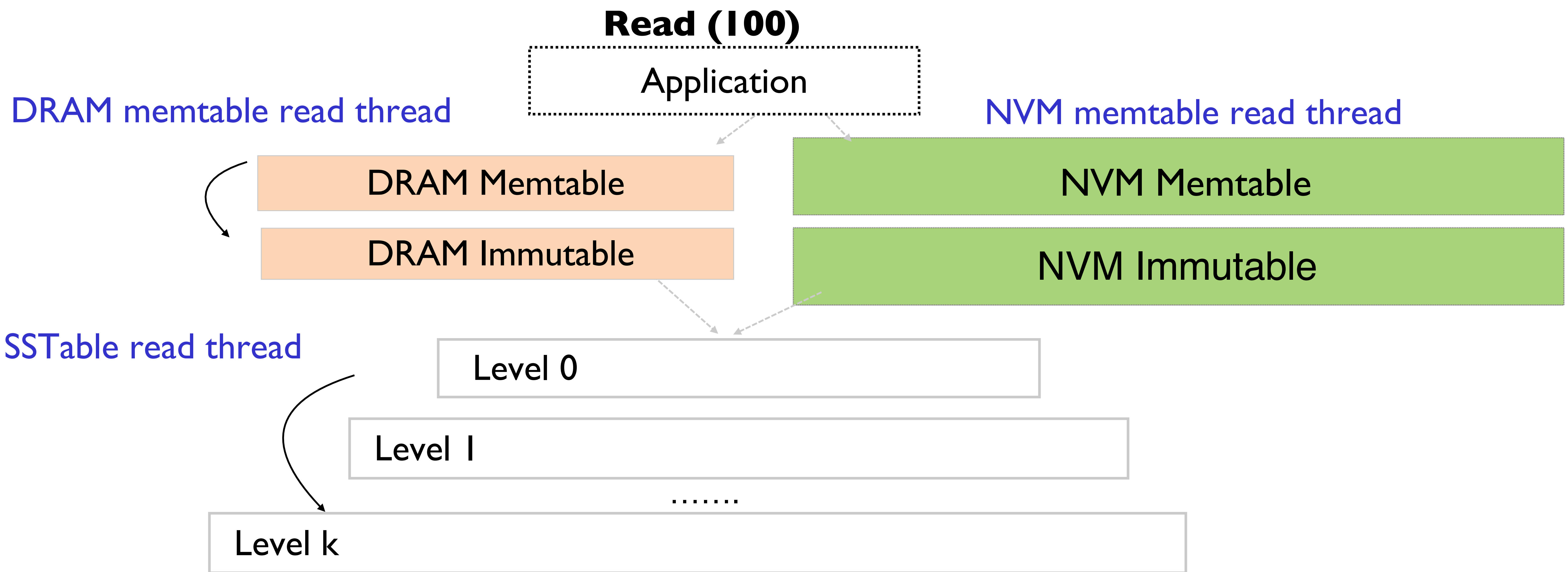
Idea: Avoid logging for NVM memtable with in-place commits



NVM memtable recovery – remap map file and find root pointer

4. Increase Parallelism: Read Threading

Solution: Parallelize search using dedicated threads



Thread management overhead can be expensive

- Bloom filters to launch threading only if DRAM memtable is a miss

Outline

Introduction

Background on LevelDB

Motivation

- High serialization, compaction, and logging cost
- Lack of parallelism

NoveLSM Design

- Persistent memtable, NVM mutability, In-place commits
- Read parallelism

Evaluation

Conclusion

Evaluation

Benchmarks and application traces

- Dbbench – Widely used LSM benchmark
- YCSB cloud benchmark (see paper)

Evaluation Goals

- Immutable memtable reduce (de)serialization cost?
- Mutable membtable reduce compaction cost?
- When read parallelism is effective?
- Reducing logging improves restart performance?

Evaluation Methodology

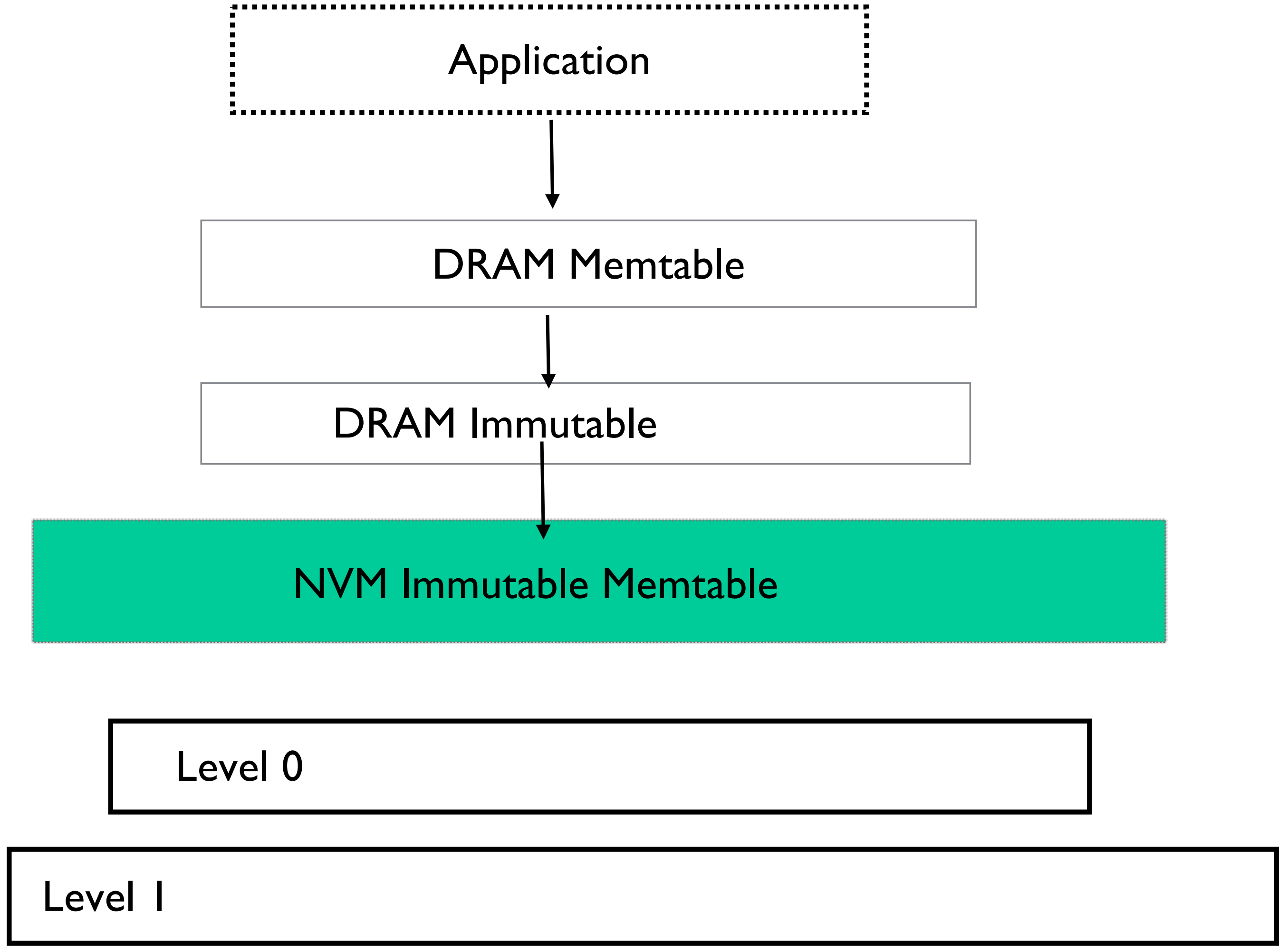
- 16 GB database size and vary values sizes
- SSTables always placed in NVM for all approaches

Immutable Memtable: Serialization Impact

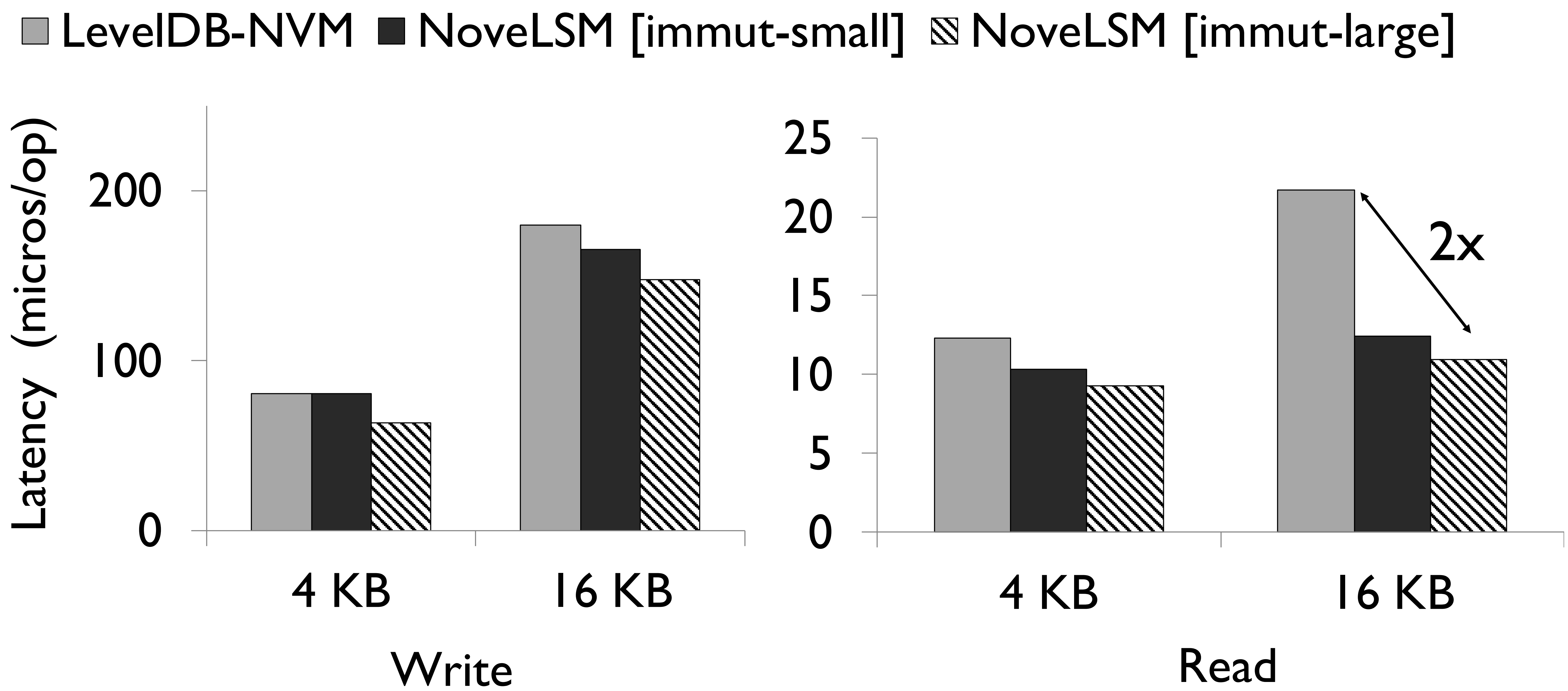
LevelDB-NVM – Vanilla LevelDB using NVM for SSTables

NoveLSM [immut-small] – 2GB NVM memtable

NoveLSM [immut-large] – 4GB NVM memtable



Serialization Impact: Immutable Memtable



Immutable memtable provides marginal gains for writes

- Compaction cost limits benefits

Reduces read deserialization reducing latency by 2x

Reducing Compaction: Mutable Memtable

RocksDB – Facebook's implementation, optimized for SSD

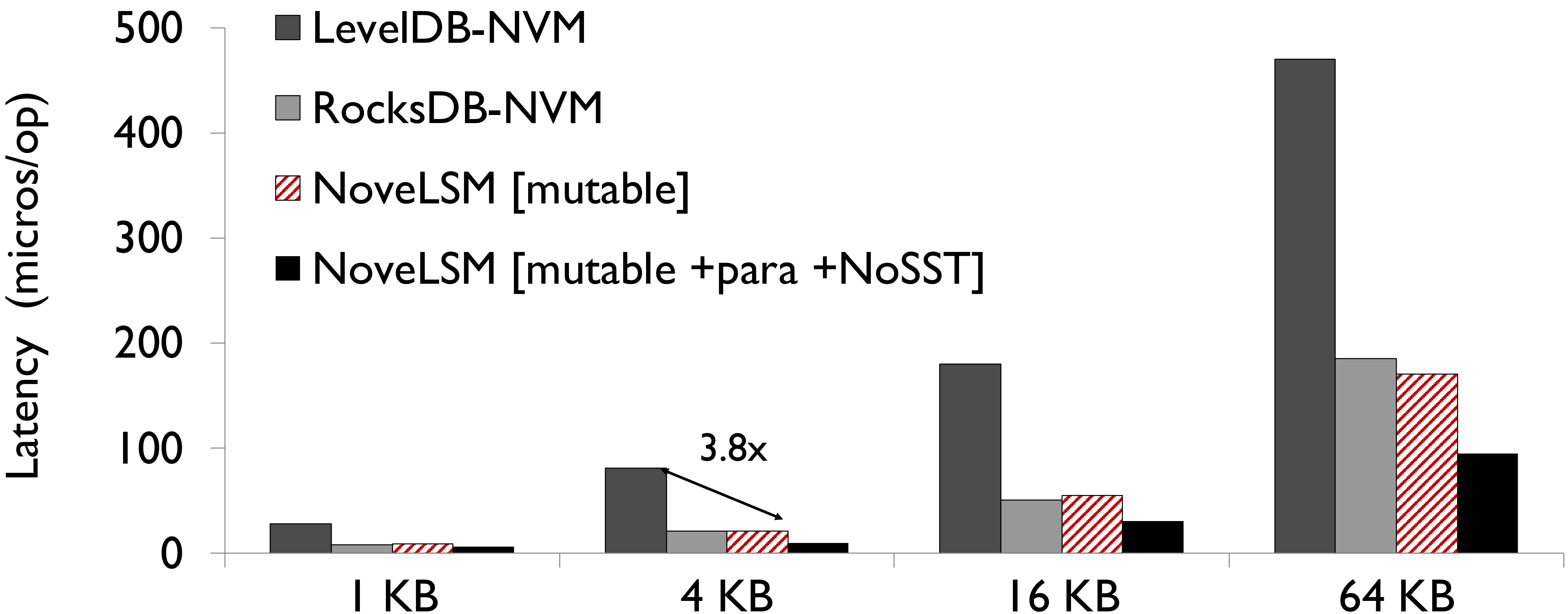
- Provides parallel compaction
- SSTable uses plain table (cuckoo hashmap) for random access

NoveLSM [mutable] – Direct mutable 4 GB NVM memtable

NoveLSM [mutable +para] – Mutable NVM + read parallelism

NoveLSM [mutable+para +NoSST] – All mutable memtable without SST

Reducing Compaction: Mutable Memtable

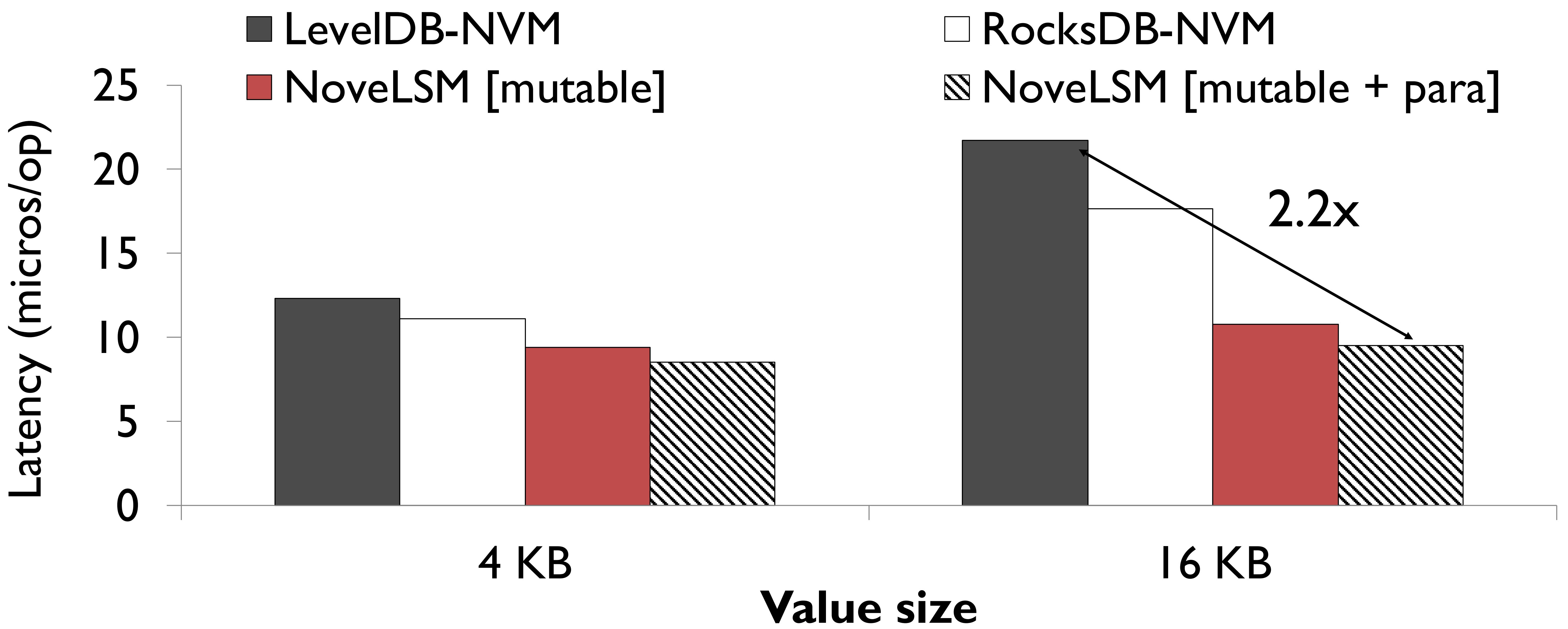


Mutable NVM memtable provides up to 3.8x gains over LevelDB

RocksDB parallel compaction and plain table storage effective

NoveLSM [mutable+para +NoSST] – upto 50% gain even over RocksDB

Read Parallelism Impact

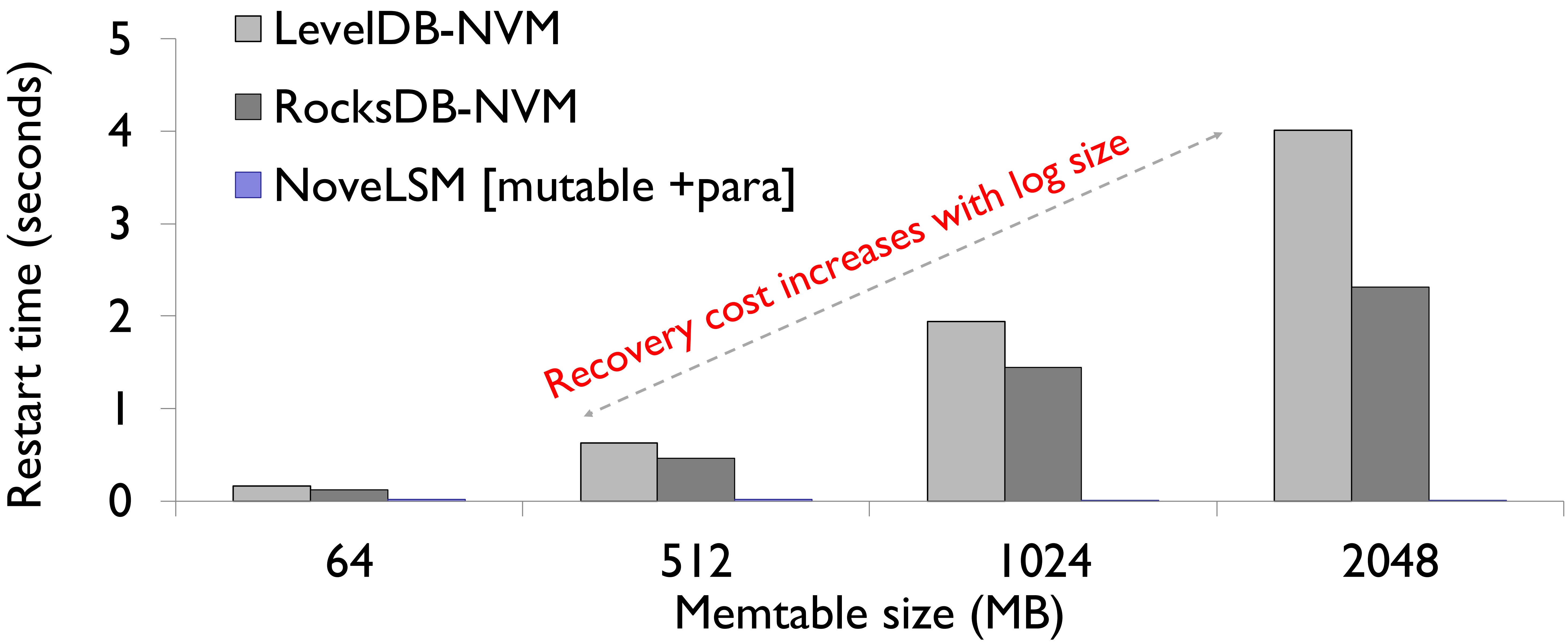


Mutable NVM memtable improves read performance even over RocksDB

Read parallelism (mutable+para) provides gains for larger value sizes

- NoveLSM provides 73% gains even over RocksDB

Restart Performance



For LevelDB and RocksDB, we increase DRAM memtable size

For NoveLSM, we increase persistent NVM memtable size

NoveLSM reduces log recovery cost by more than 99%

Outline

Introduction

Background on LevelDB

Motivation

- High serialization, compaction, and logging cost
- Lack of parallelism

NoveLSM Design

- Persistent memtable, NVM mutability, In-place commits
- Read parallelism

Evaluation

Conclusion

Summary

Motivation

- Simply adding NVMs to existing LSMs for storage not sufficient
- Eliminating S/W overhead (e.g., serialization, compaction) is critical

Solution

- NoveLSM - byte-addressable and persistent data structures
- Reduce serialization, compaction, and logging cost
- Improve read parallelism

Evaluation

- NoveLSM reduces write latency by up to 3.8x and read latency by 2x
- Makes restarts significantly fast

Conclusion

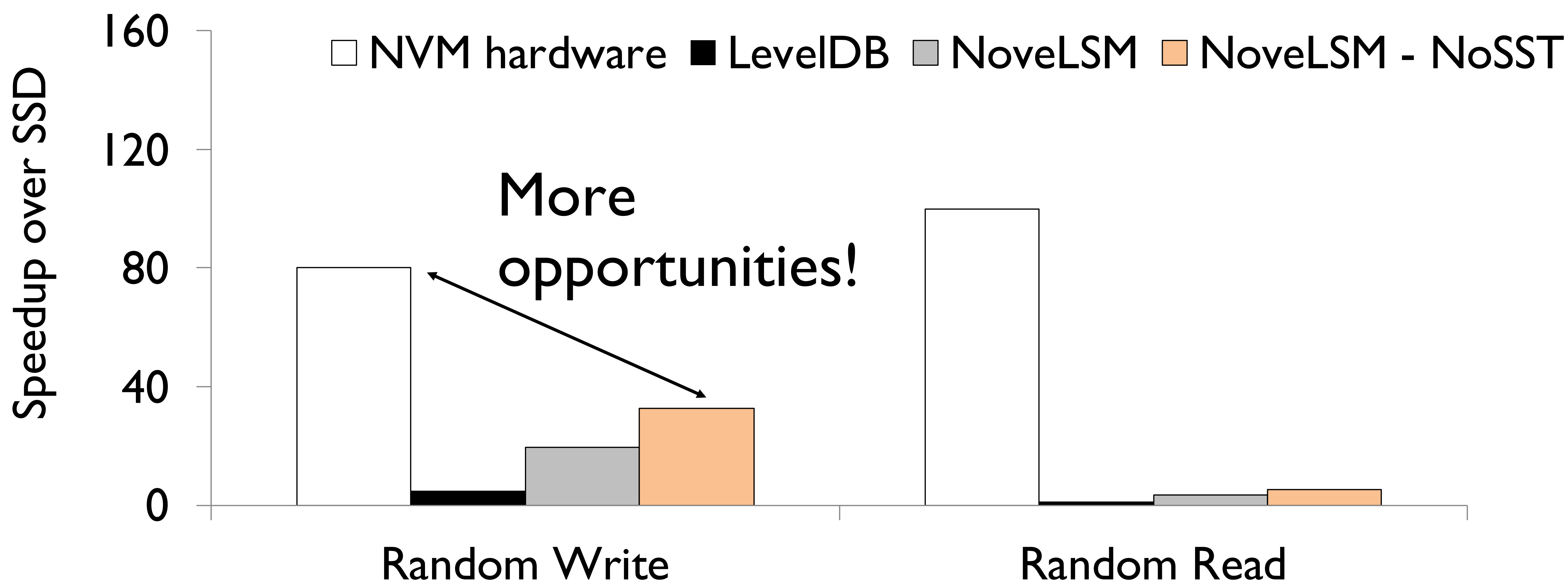
We are moving towards a storage era with microsecond latency

Eliminating software overhead is critical

- We take first step towards redesigning existing LSMs for NVM

Future work

- Rethink LSMs from scratch for NVM hardware-level performance



Thanks!

Questions?