

Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory

Pengfei Zuo, Yu Hua, Jie Wu

Huazhong University of Science and Technology, China

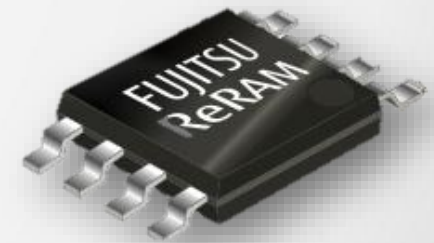
Persistent Memory (PM)

- Non-volatile memory as PM is expected to replace or complement DRAM as main memory
 - Non-volatility, low power, large capacity

	PCM	ReRAM	DRAM
Read (ns)	20-70	20-50	10
Write (ns)	150-220	70-140	10
Non-volatility	✓	✓	✗
Standby Power	~0	~0	High
Density (Gb/cm ²)	13.5	24.5	9.1



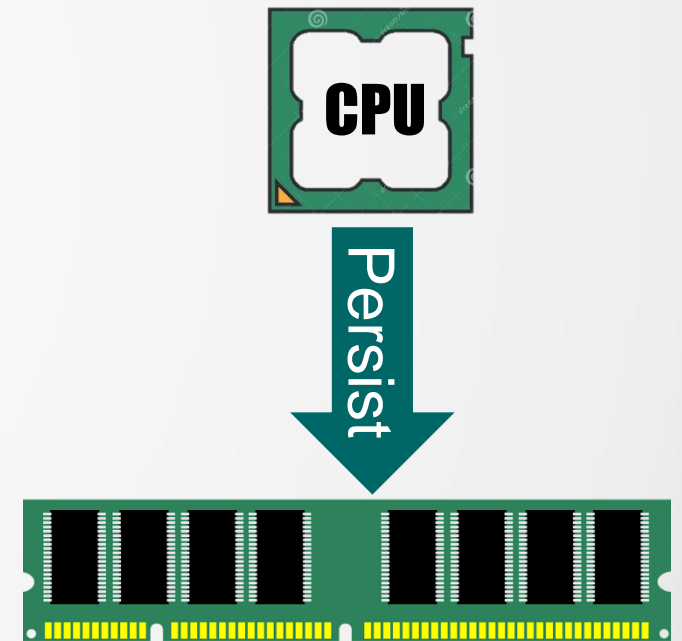
PCM



ReRAM

Index Structures in DRAM vs PM

- Index structures are critical for memory&storage systems
- Traditional indexing techniques originally designed for DRAM become inefficient in PM
 - **Hardware limitations of NVM**
 - Limited cell endurance
 - Asymmetric read/write latency and energy
 - Write optimization matters
 - **The requirement of data consistency**
 - Data are persistently stored in PM
 - Crash consistency on system failures



Tree-based vs Hashing Index Structures

➤ Tree-based index structures

- **Pros:** good for range query
- **Cons:** $O(\log(n))$ time complexity for point query
- Ones for PM have been widely studied
 - CDDS B-tree [FAST'11]
 - NV-Tree [FAST'15]
 - wB+-Tree [VLDB'15]
 - FP-Tree [SIGMOD'16]
 - WORT [FAST'17]
 - FAST&FAIR [FAST'18]

Tree-based vs Hashing Index Structures

➤ Tree-based index structures

- **Pros:** good for range query
- **Cons:** $O(\log(n))$ time complexity for point query
- Ones for PM have been widely studied
 - CDDS B-tree [FAST'11]
 - NV-Tree [FAST'15]
 - wB+-Tree [VLDB'15]
 - FP-Tree [SIGMOD'16]
 - WORT [FAST'17]
 - FAST&FAIR [FAST'18]

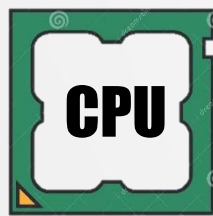
➤ Hashing index structures

- **Pros:** constant time complexity for point query
- **Cons:** do not support range query
- Widely used in main memory
 - Main memory databases
 - In-memory key-value stores, e.g., Memcached and Redis
- **When maintained in PM, multiple non-trivial challenges exist**
 - Rarely touched by existing work

Challenges of Hashing Indexes for PM

① High overhead for consistency guarantee

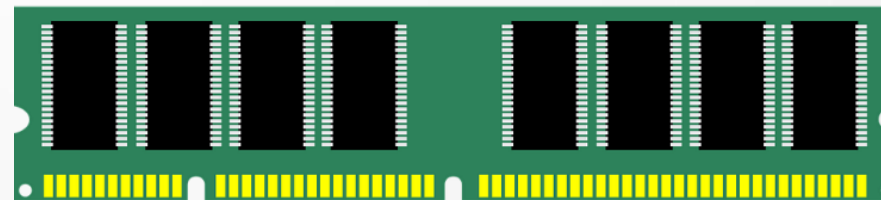
- Ordering memory writes
 - Cache line flush and memory fence instructions
- Avoiding partial updates for non-atomic writes
 - Logging or copy-on-write (CoW) mechanisms



Volatile caches



8-byte width



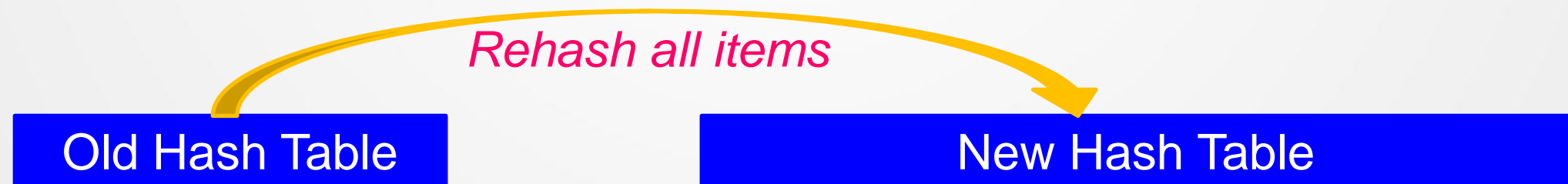
Non-volatile memory

Challenges of Hashing Indexes for PM

- ① **High overhead for consistency guarantee**
- ② **Performance degradation for reducing writes**
 - Hashing schemes for DRAM usually cause many extra writes for dealing with hash collisions [INFLOW'15, MSST'17]
 - Write-friendly hashing schemes reduce writes but at the cost of decreasing access performance
 - PCM-friendly hash table (PFHT) [INFLOW'15]
 - Path hashing [MSST'17]

Challenges of Hashing Indexes for PM

- ① High overhead for **consistency guarantee**
- ② Performance degradation for **reducing writes**
- ③ **Cost inefficiency for resizing hash table**
 - Double the table size and iteratively rehash all items
 - Take $O(N)$ time to complete
 - N insertions with cache line flushes & memory fences



Existing Hashing Index Schemes for PM

(“**X**”: bad, “**✓**”: good, “-”: moderate)

	Bucketized Cuckoo (BCH)	PFHT ¹	Path Hashing ²
Memory efficiency	✓	✓	✓
Search	✓	-	-
Deletion	✓	-	-
Insertion	X	-	-
NVM writes	X	✓	✓
Resizing	X	X	X
Consistency	X	X	X

[1] B. Debnath et al. “Revisiting hash table design for phase change memory”, INFLOW, 2015.

[2] P. Zuo and Y. Hua. “A write-friendly hashing scheme for non-volatile memory systems”, MSST, 2017.

Existing Hashing Index Schemes for PM

(“**X**”: bad, “**✓**”: good, “-”: moderate)

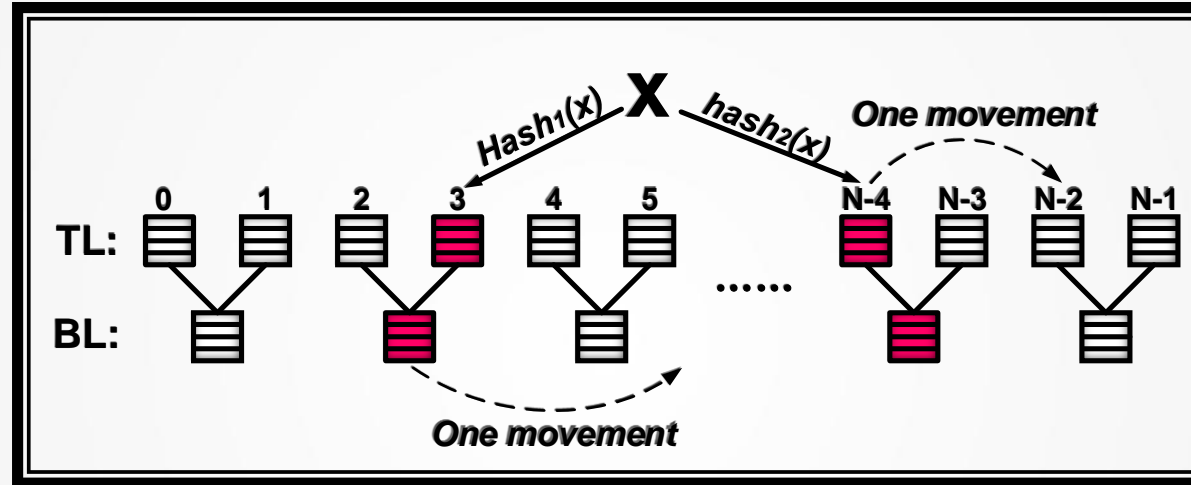
	Bucketized Cuckoo (BCH)	PFHT ¹	Path Hashing ²	Level Hashing
Memory efficiency	✓	✓	✓	✓
Search	✓	-	-	✓
Deletion	✓	-	-	✓
Insertion	X	-	-	✓
NVM writes	X	✓	✓	✓
Resizing	X	X	X	✓
Consistency	X	X	X	✓

[1] B. Debnath et al. “Revisiting hash table design for phase change memory”, INFLOW, 2015.

[2] P. Zuo and Y. Hua. “A write-friendly hashing scheme for non-volatile memory systems”, MSST, 2017.

Level Hashing

Write-optimized & High-performance Hash Table Structure

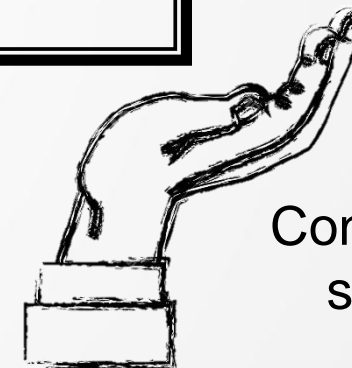


Resizing support



**Cost-efficient
In-place Resizing Scheme**

Consistency support



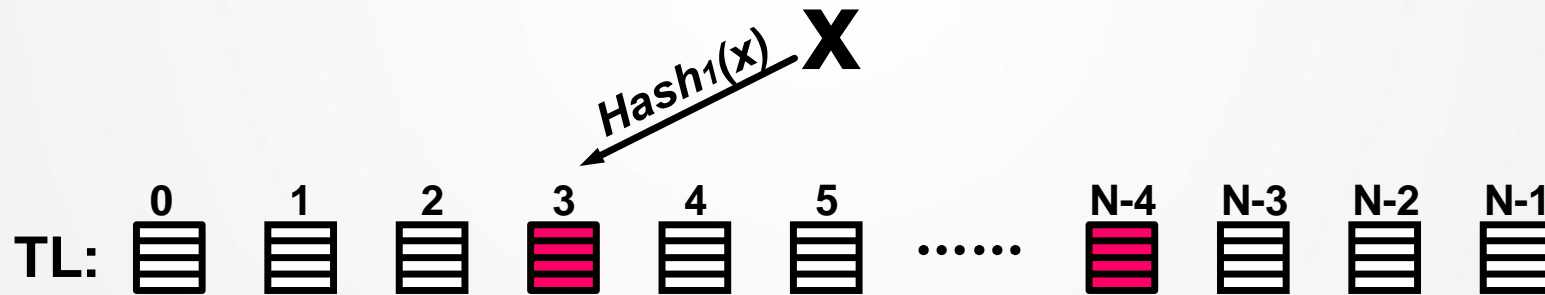
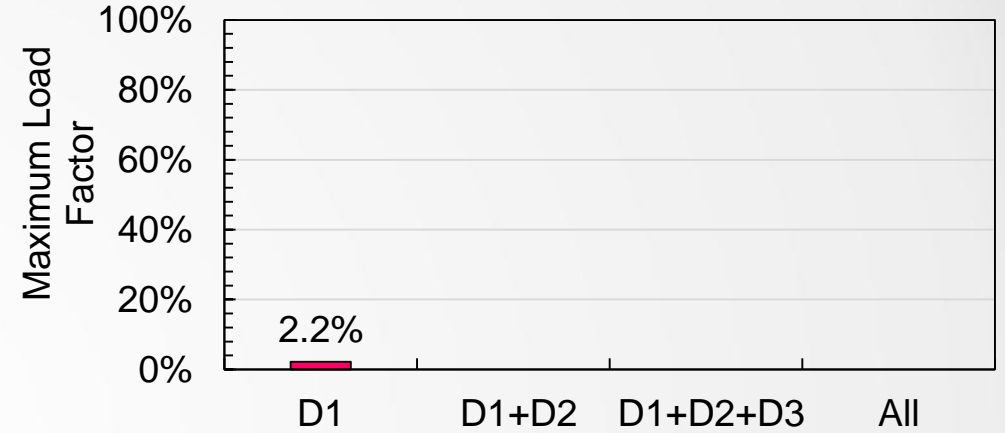
**Low-overhead Consistency
Guarantee Scheme**

Write-optimized Hash Table Structure

- ① Multiple slots per bucket
- ② Two hash locations for each key
- ③ Sharing-based two-level structure
- ④ At most one movement for each successful insertion

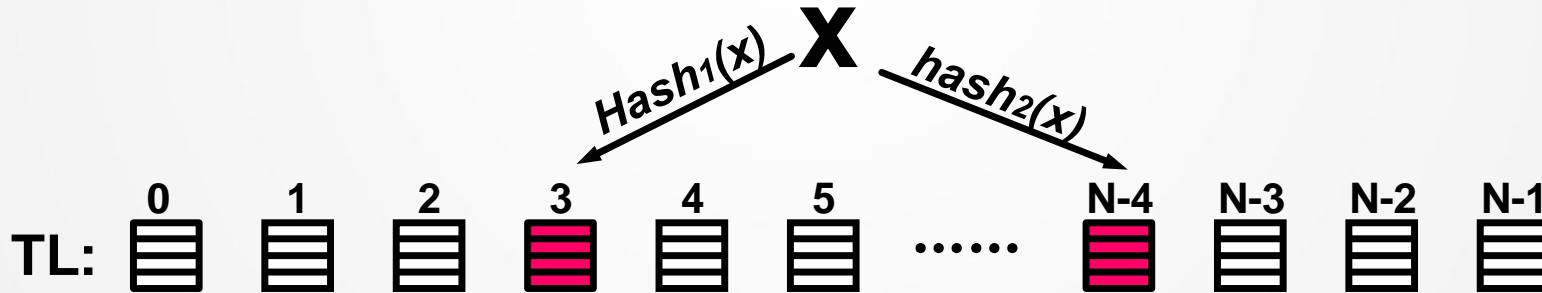
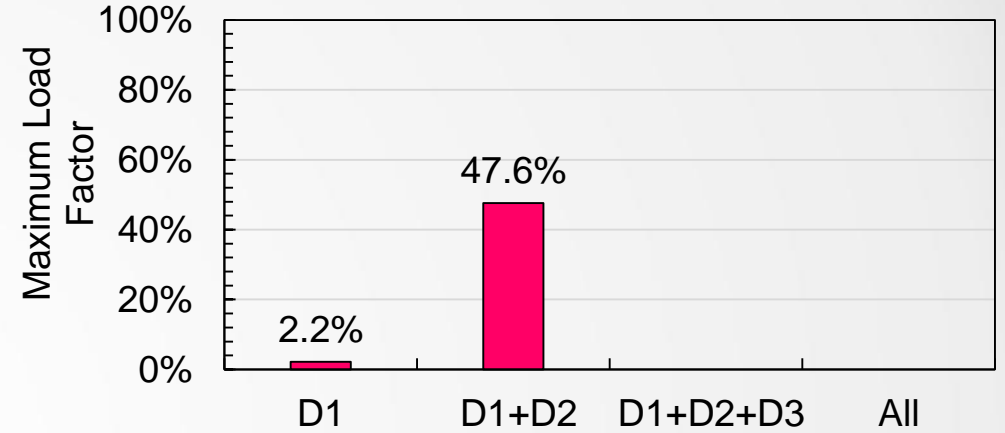
Write-optimized Hash Table Structure

- ① Multiple slots per bucket
- ② Two hash locations for each key
- ③ Sharing-based two-level structure
- ④ At most one movement for each successful insertion



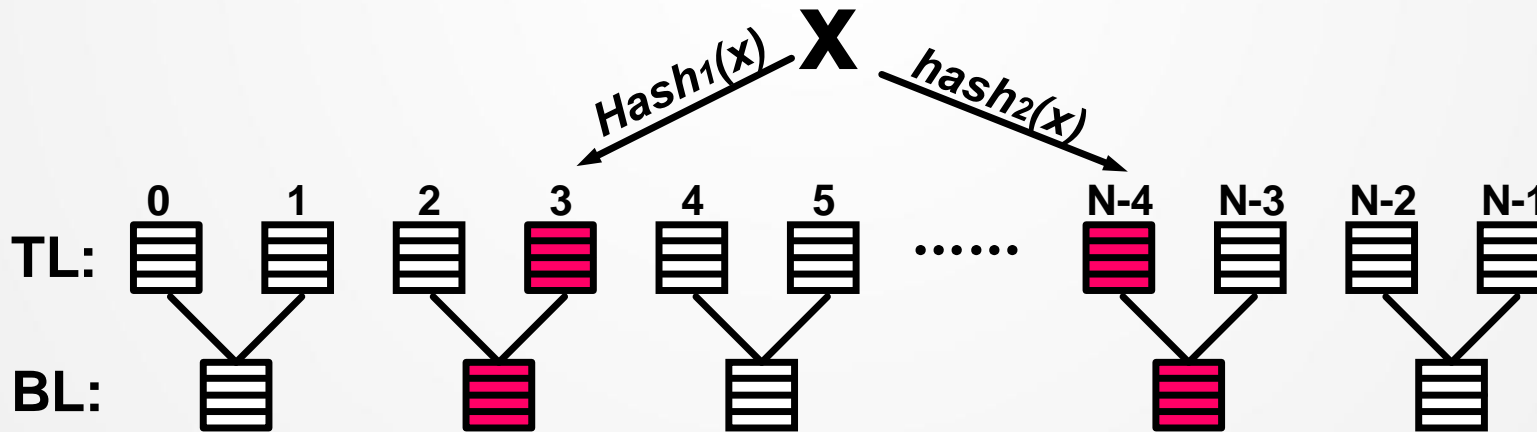
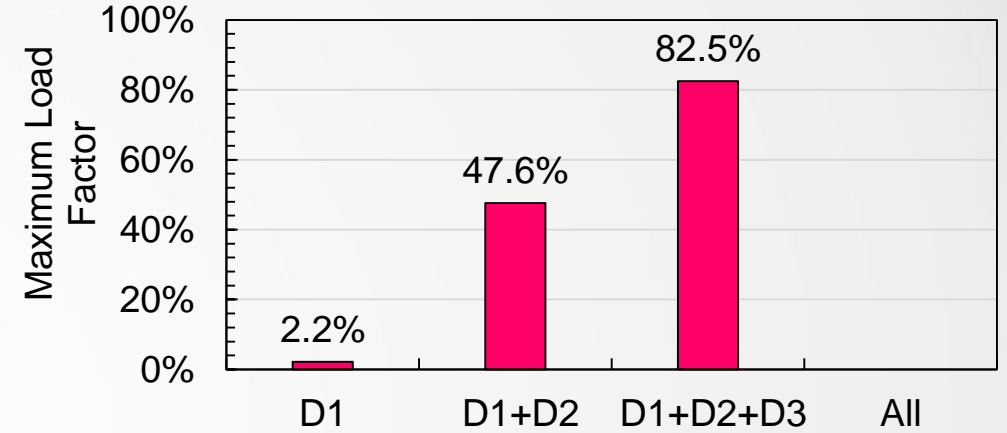
Write-optimized Hash Table Structure

- ① Multiple slots per bucket
- ② **Two hash locations for each key**
- ③ Sharing-based two-level structure
- ④ At most one movement for each successful insertion



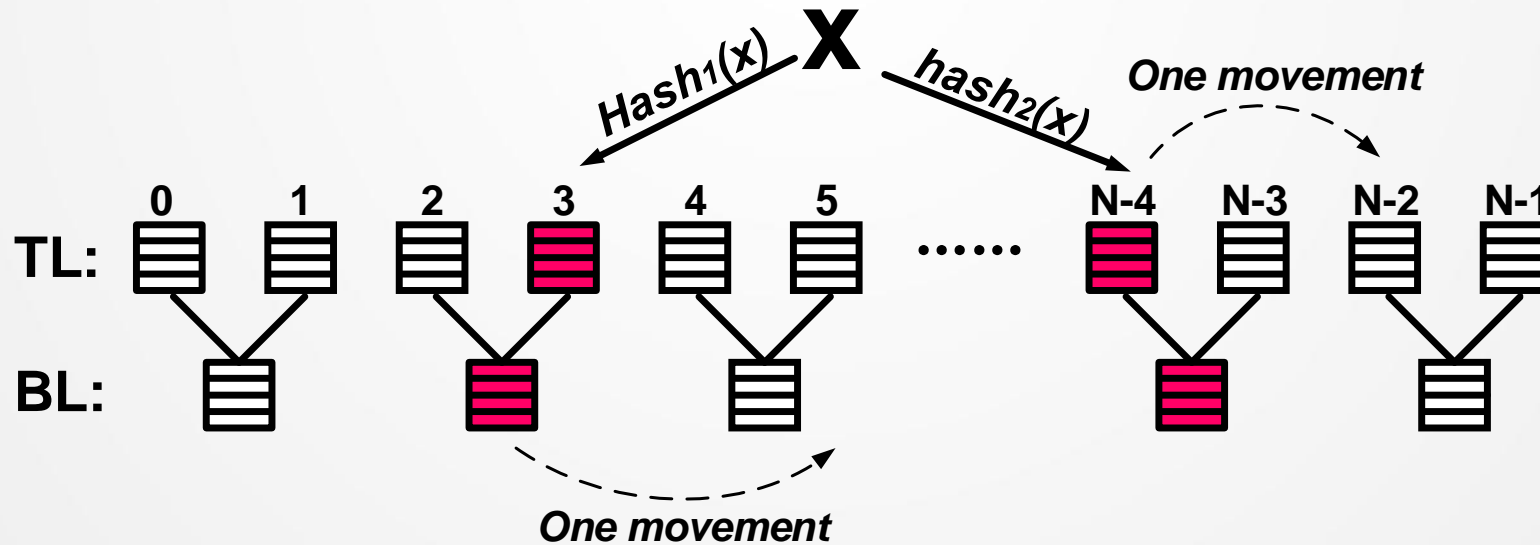
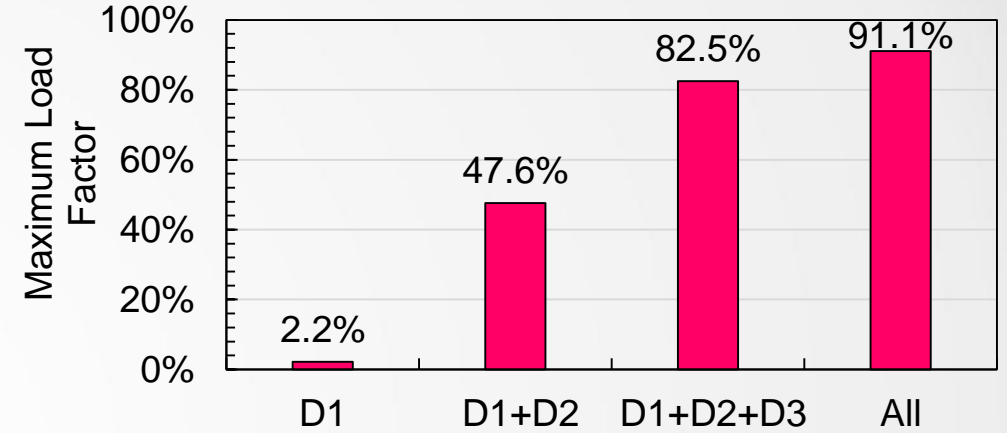
Write-optimized Hash Table Structure

- ① Multiple slots per bucket
- ② Two hash locations for each key
- ③ **Sharing-based two-level structure**
- ④ At most one movement for each successful insertion



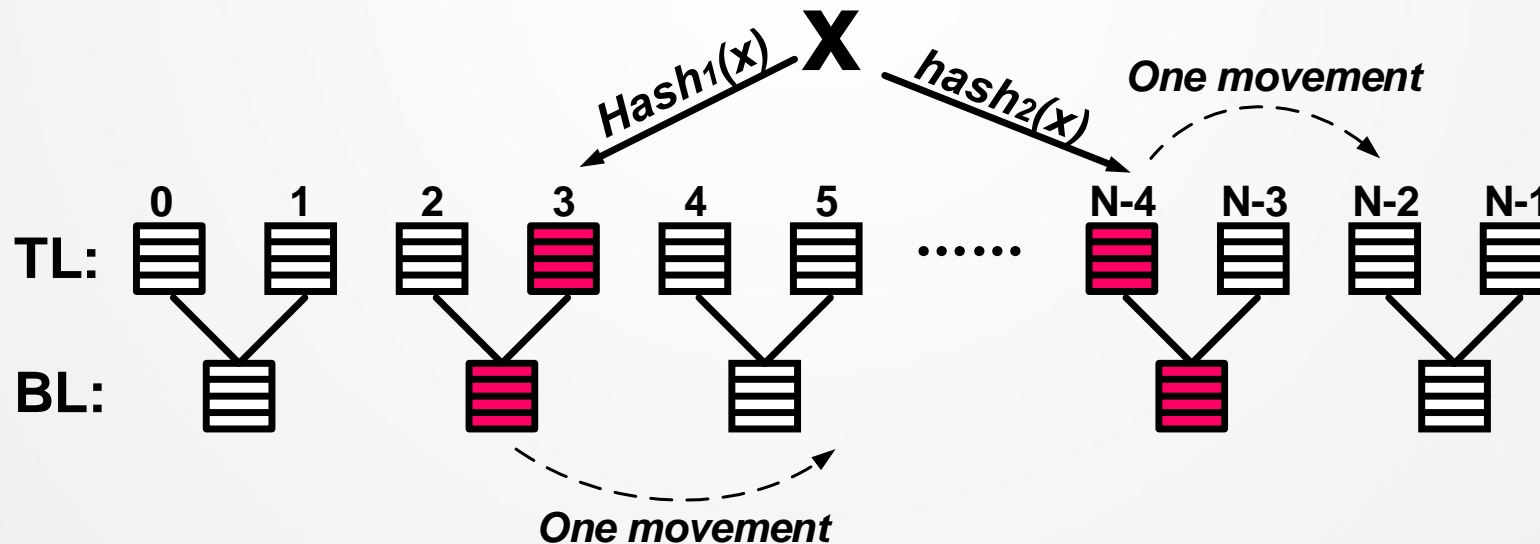
Write-optimized Hash Table Structure

- ① Multiple slots per bucket
- ② Two hash locations for each key
- ③ Sharing-based two-level structure
- ④ **At most one movement for each successful insertion**



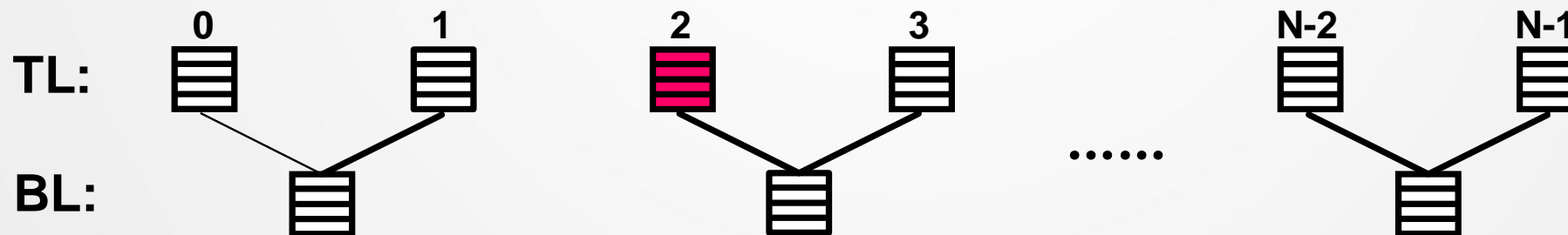
Write-optimized Hash Table Structure

- Write-optimized: only 1.2% of insertions incur one movement
- High-performance: constant-scale time complexity for all operations
- Memory-efficient: achieve high load factor by evenly distributing items



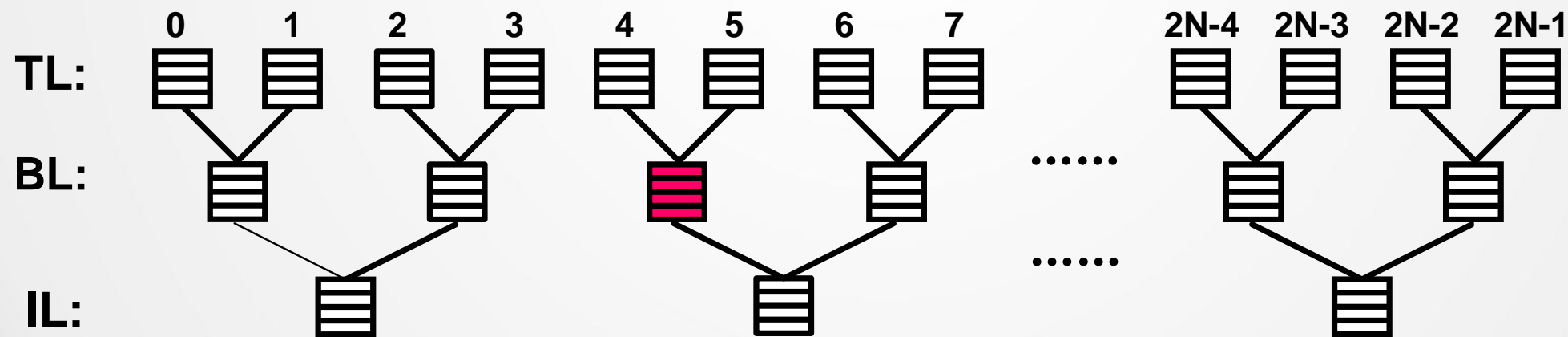
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level



Cost-efficient In-place Resizing

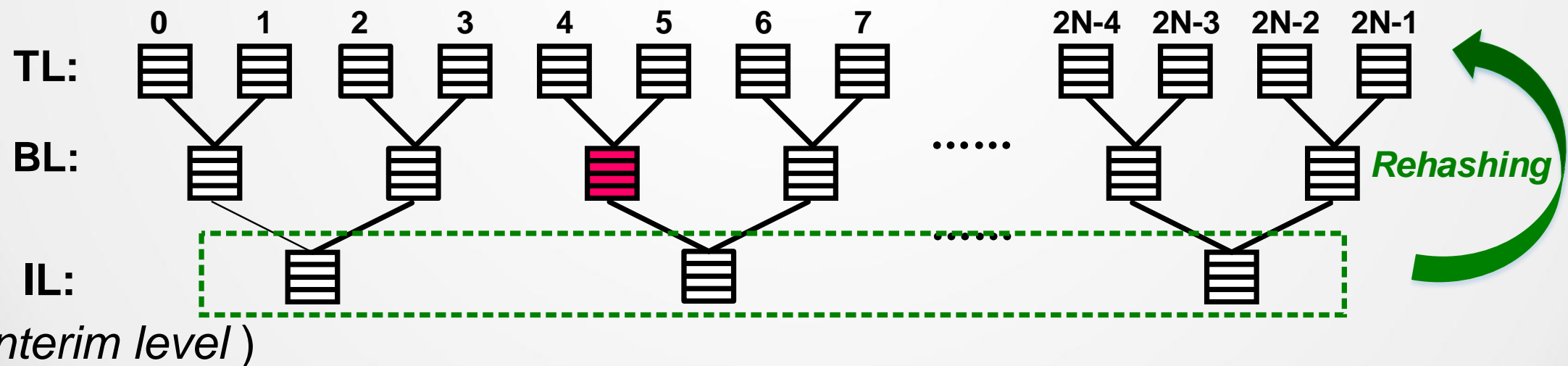
- Put a new level on top of the old hash table and only rehash items in the old bottom level



(the interim level)

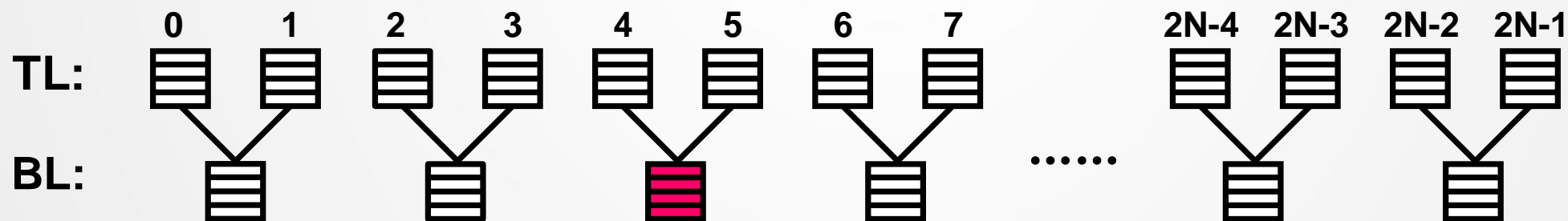
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level



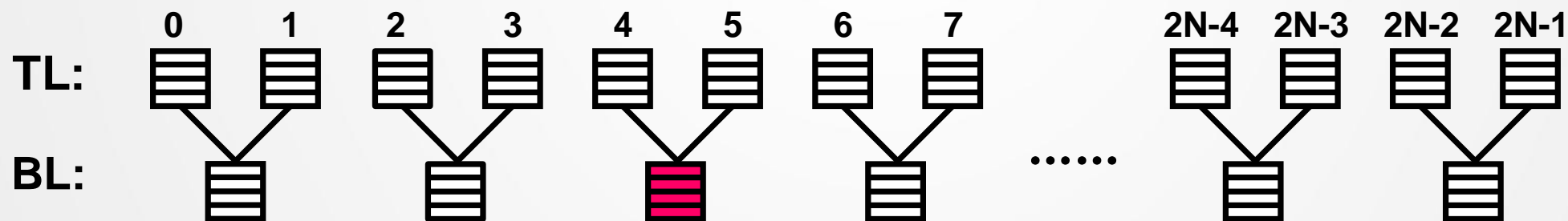
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level



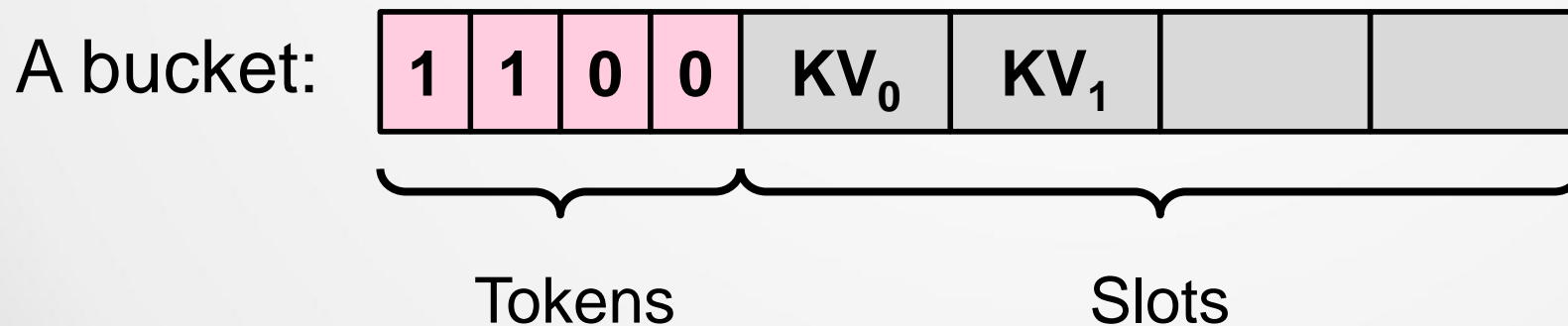
Cost-efficient In-place Resizing

- Put a new level on top of the old hash table and only rehash items in the old bottom level
 - The new hash table is exactly double size of the old one
 - Only 1/3 buckets (i.e., the old bottom level) are rehashed



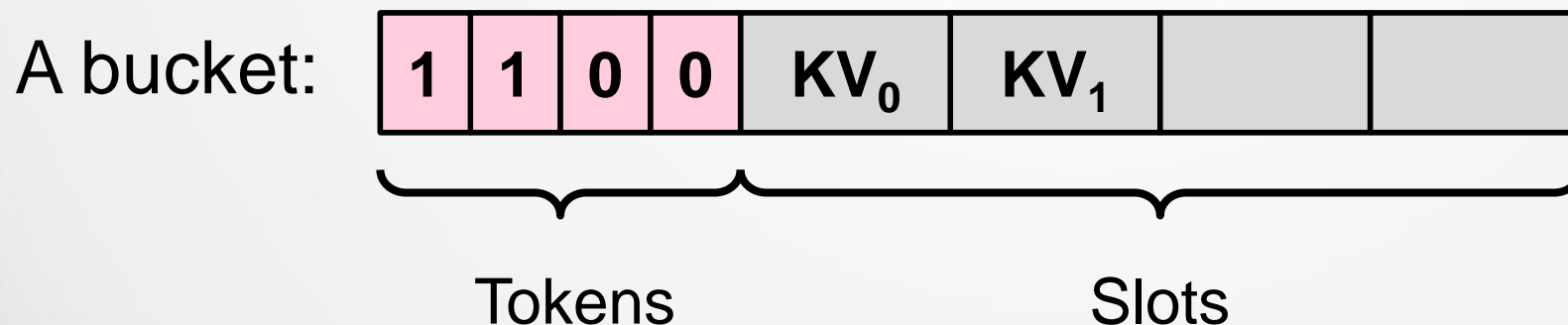
Low-overhead Consistency Guarantee

- A token associated with each slot in the open-addressing hash tables
 - Indicate whether the slot is empty
 - A token is 1 bit, e.g., “1” for non-empty, “0” for empty



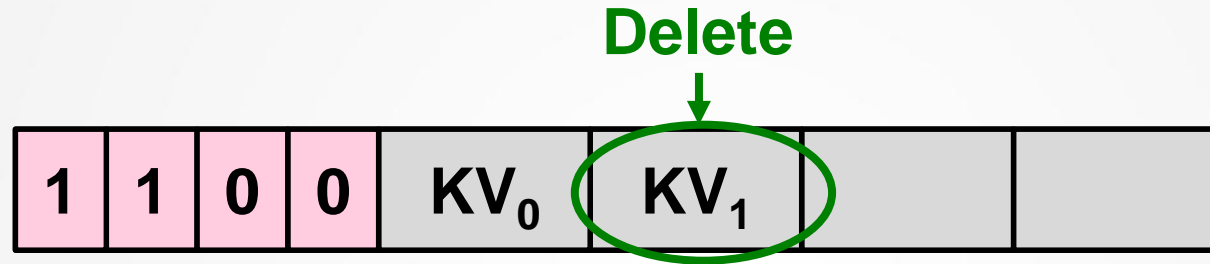
Low-overhead Consistency Guarantee

- A token associated with each slot in the open-addressing hash tables
 - Indicate whether the slot is empty
 - A token is 1 bit, e.g., “1” for non-empty, “0” for empty
- Modifying the token area only needs an atomic write
 - Leveraging the token to perform log-free operations



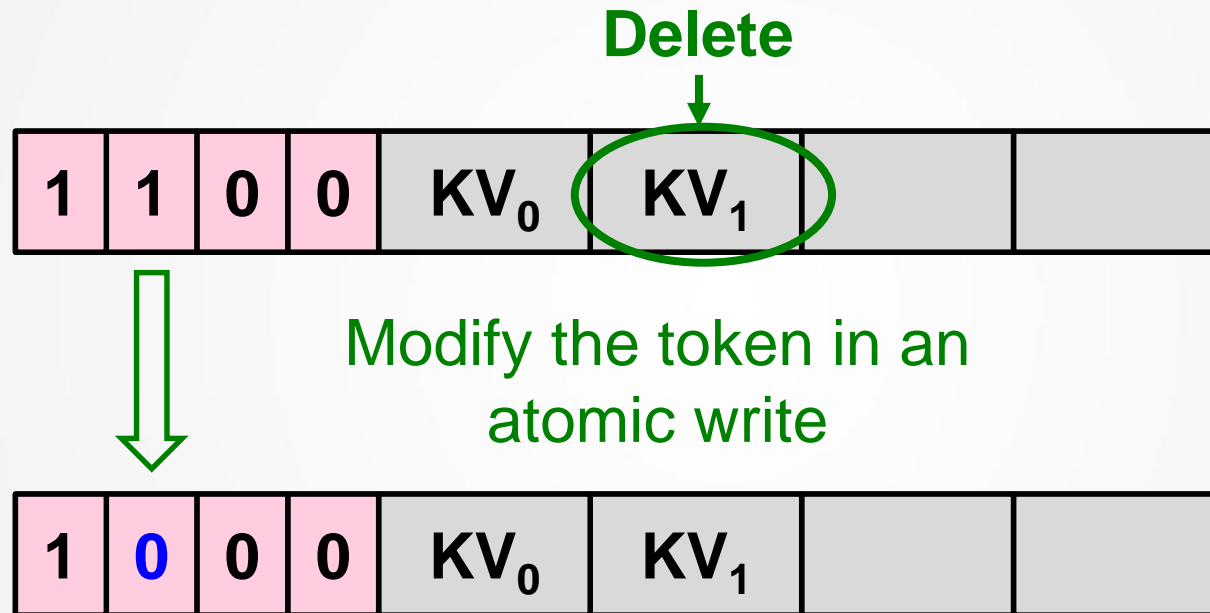
Log-free Deletion

- Delete an existing item



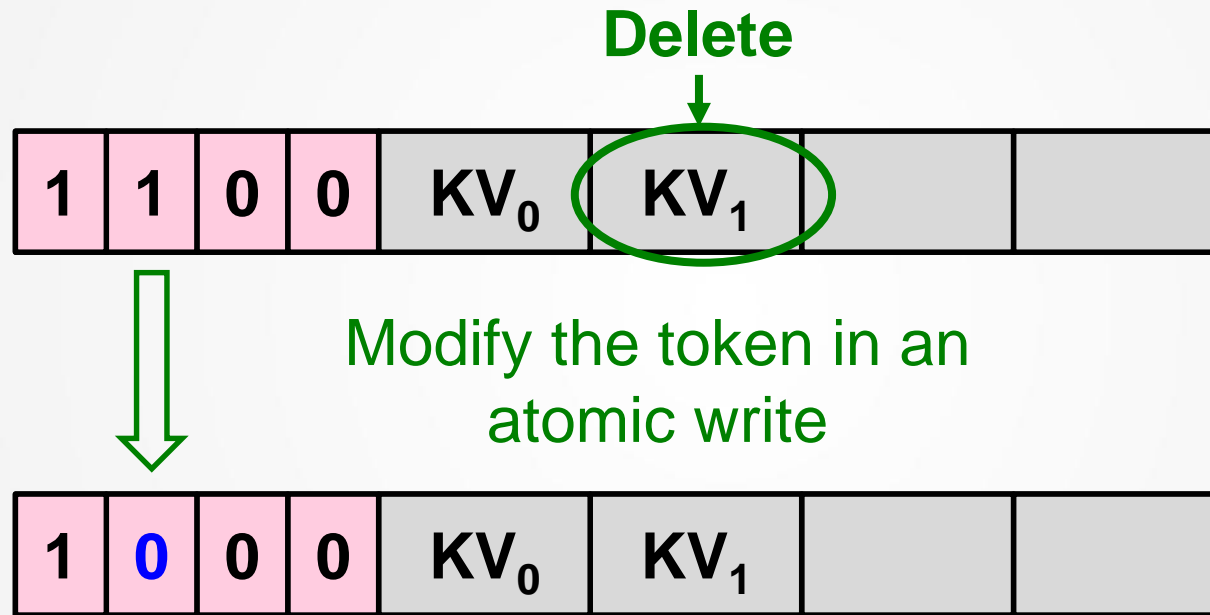
Log-free Deletion

- Delete an existing item



Log-free Deletion

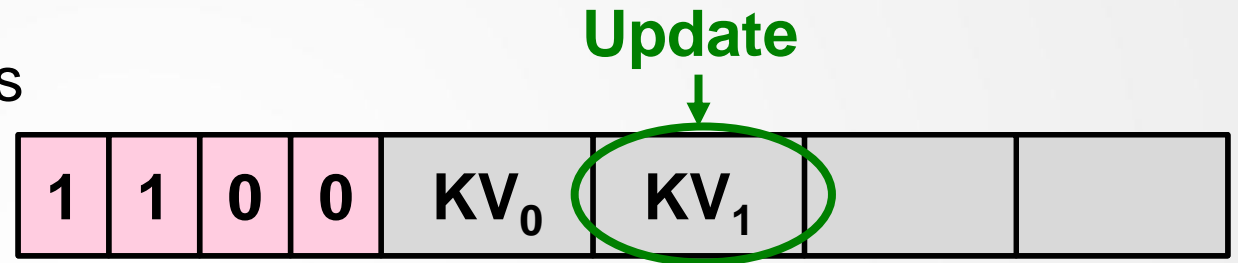
- Delete an existing item



- **Log-free insertion and log-free resizing**
 - Please find them in our paper

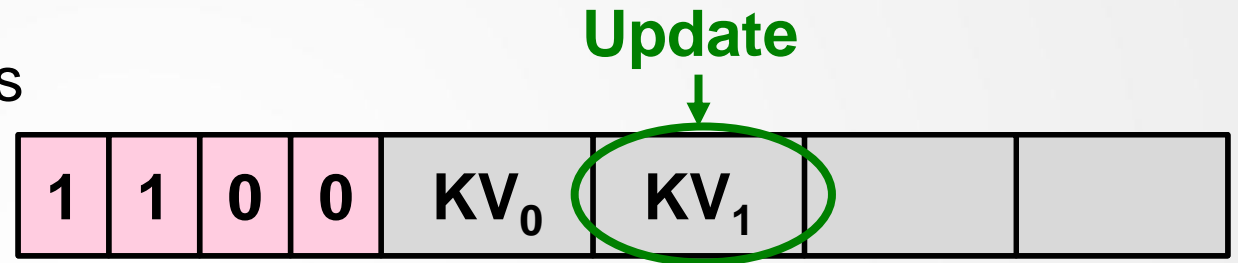
Consistency Guarantee for Update

- If directly update an existing key-value item in place
 - Inconsistency on system failures



Consistency Guarantee for Update

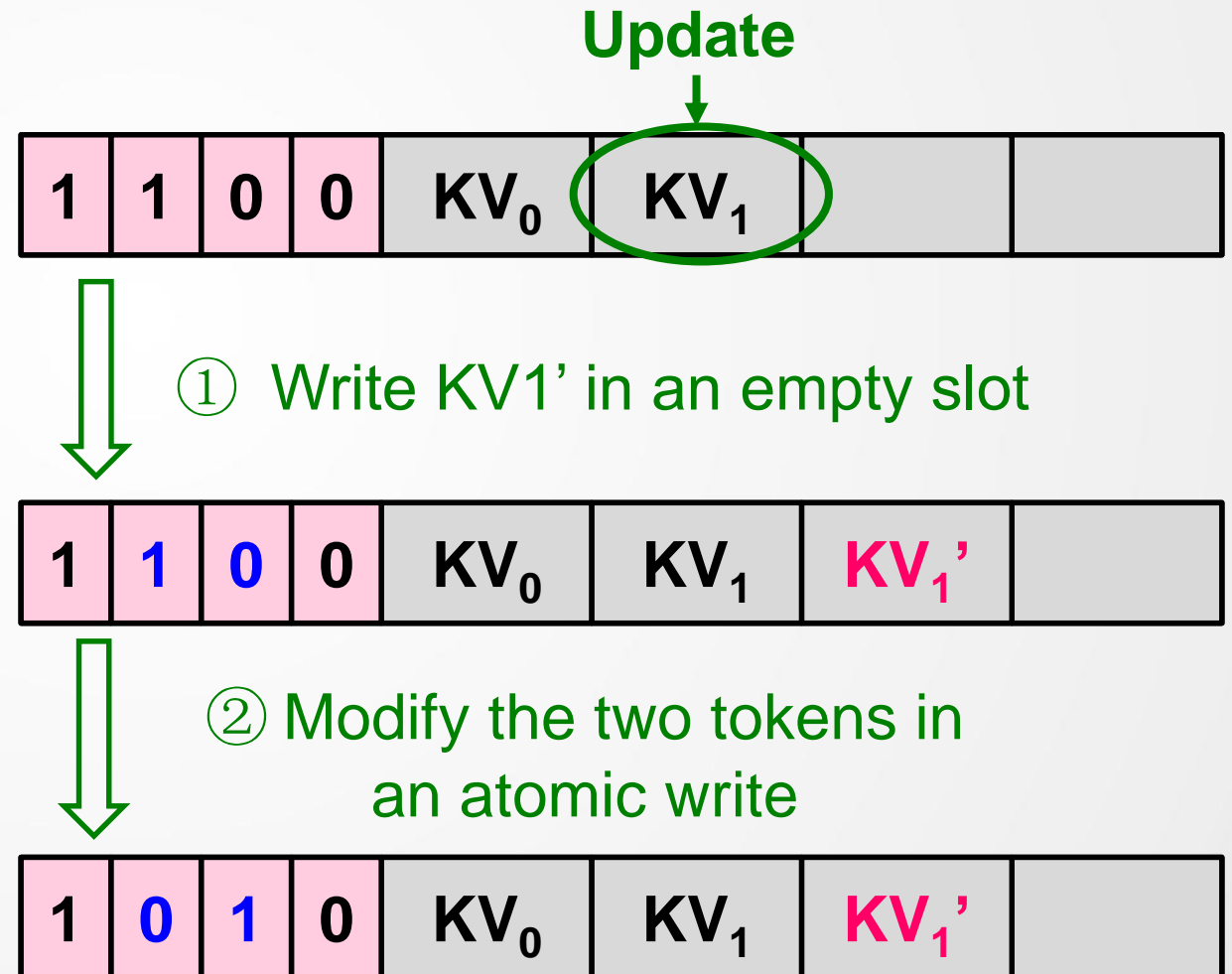
- If directly update an existing key-value item in place
 - Inconsistency on system failures
- A straightforward solution is to use logging



Expensive!

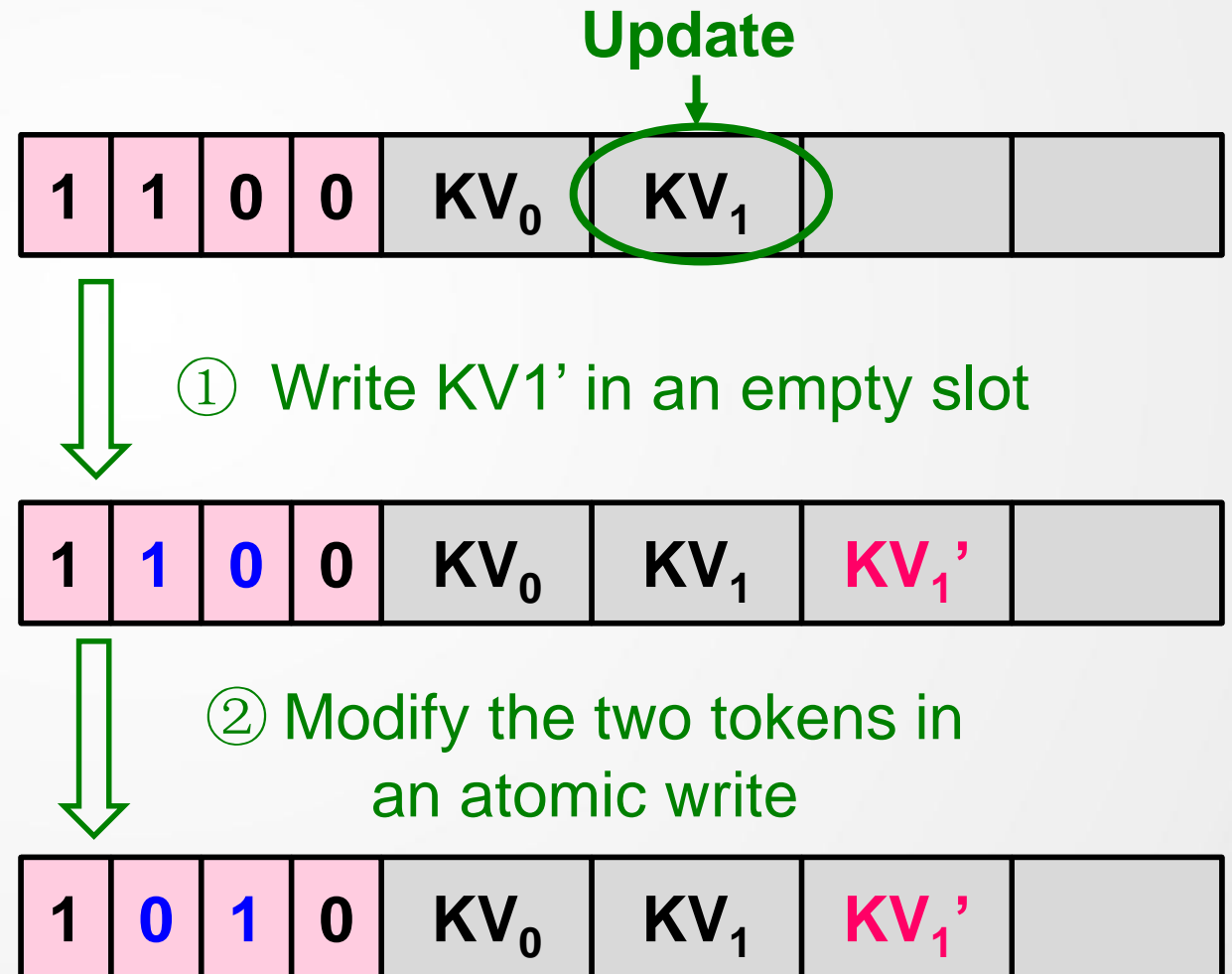
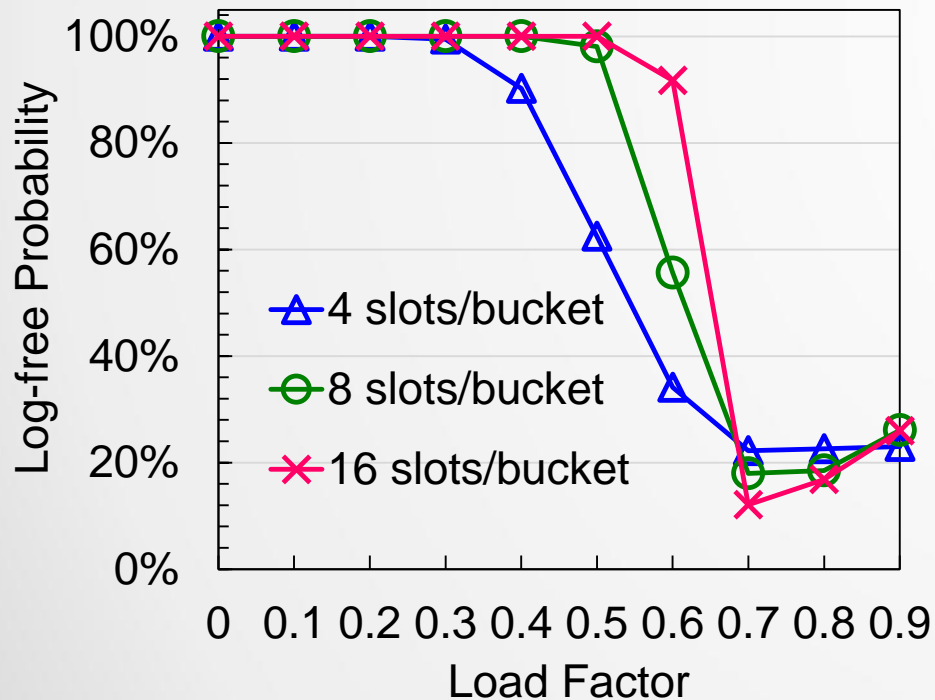
Opportunistic Log-free Update

- **Our scheme:** check whether there is an empty slot in the bucket storing the old item
 - Yes: log-free update
 - No: using logging



Opportunistic Log-free Update

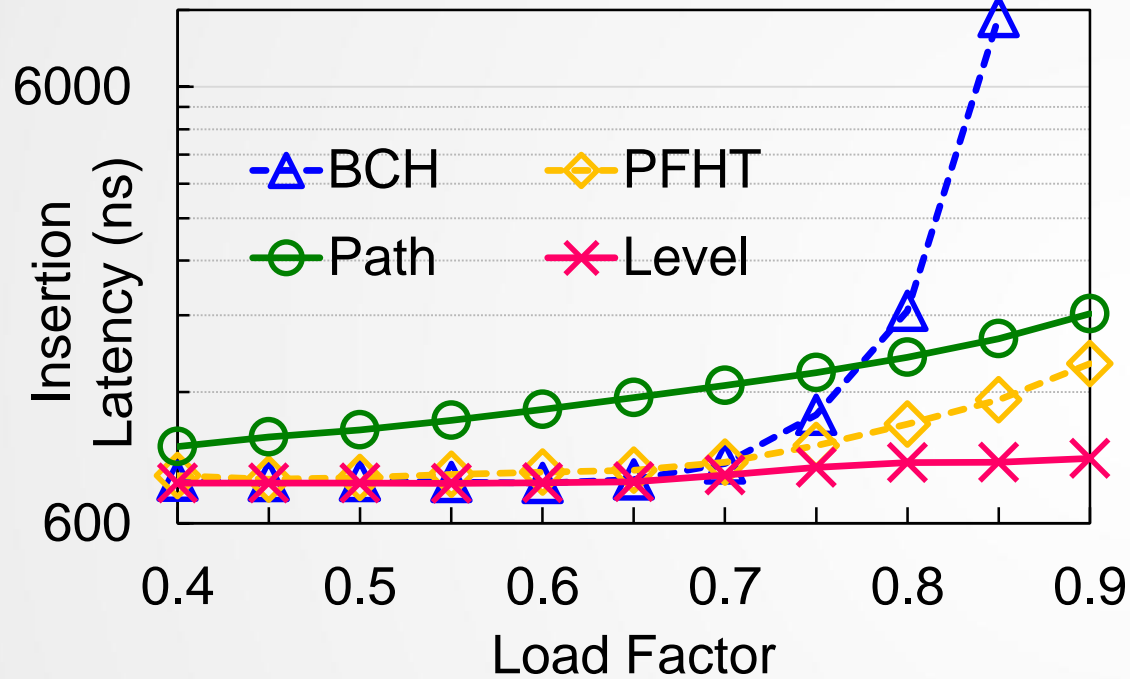
- **Our scheme:** check whether there is an empty slot in the bucket storing the old item
 - Yes: log-free update
 - No: using logging



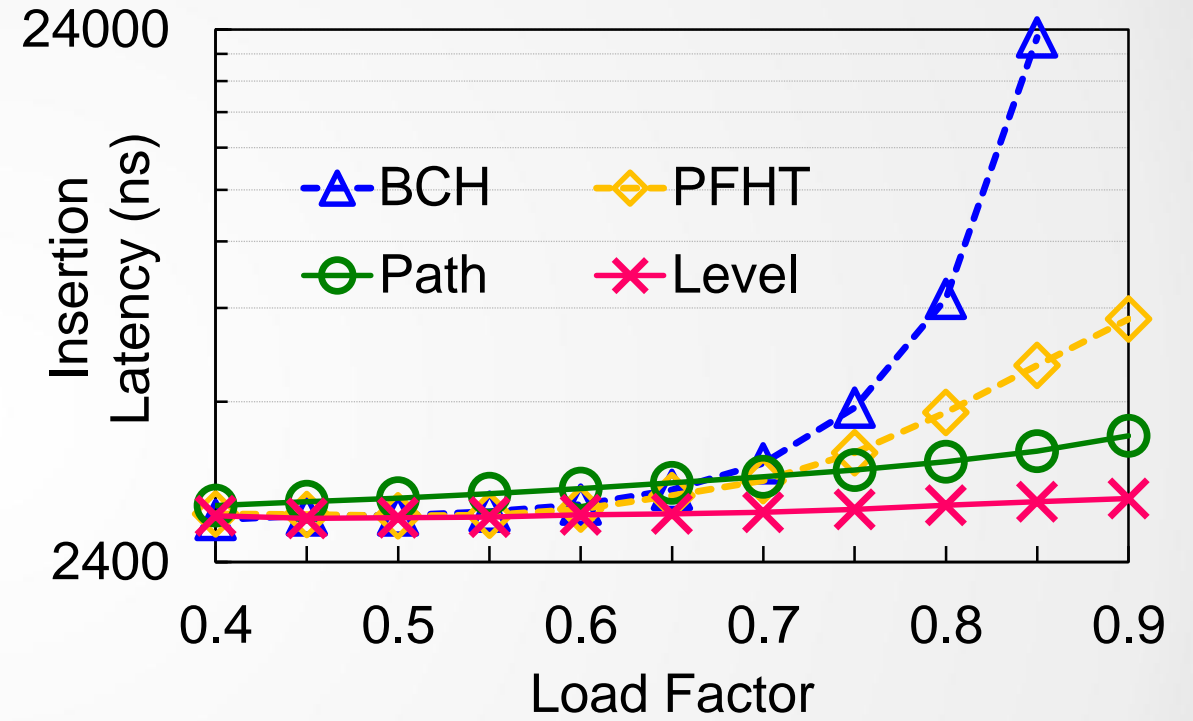
Performance Evaluation

- Both in DRAM and simulated PM platforms
 - Quartz (Hewlett Packard)
 - A DRAM-based performance emulator for PM
- Comparisons
 - Bucketized cuckoo hashing (BCH) [NSDI'13]
 - PCM-friendly hash table (PFHT) [INFLOW'15]
 - Path hashing [MSST'17]
 - In PM, implement their persistent versions using our proposed log-free consistency guarantee schemes

Insertion Latency



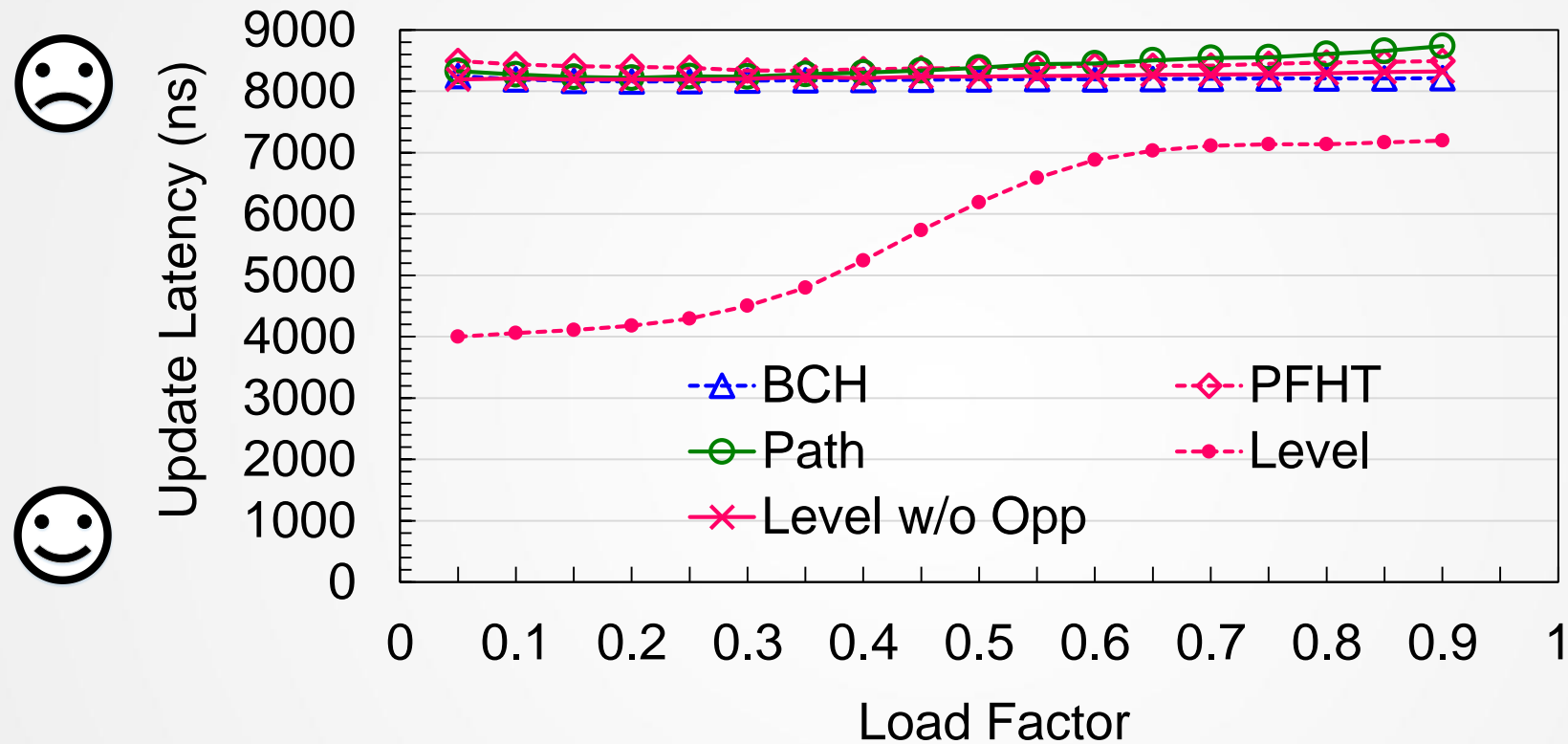
DRAM



NVM read/write latency: 200/600

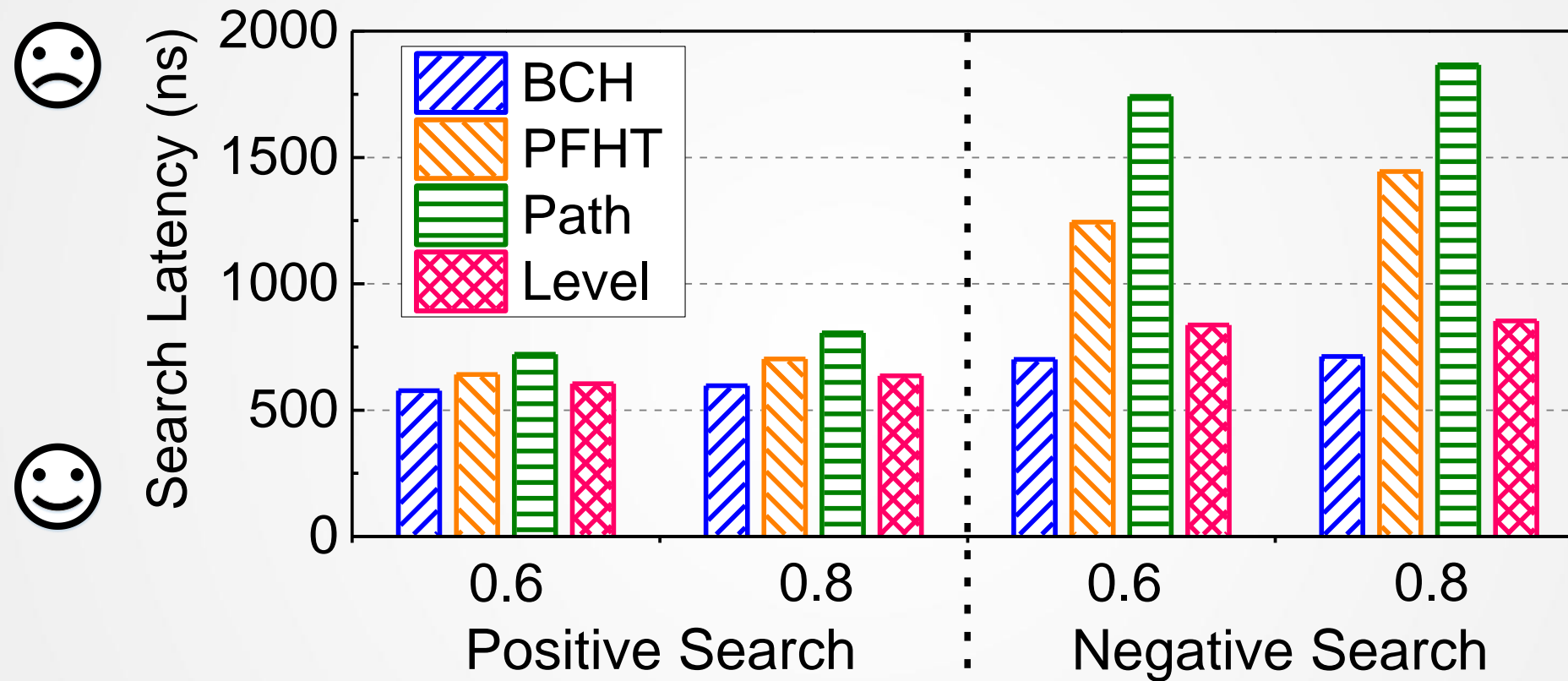
- Level hashing has the best insertion performance in both DRAM and NVM

Update Latency



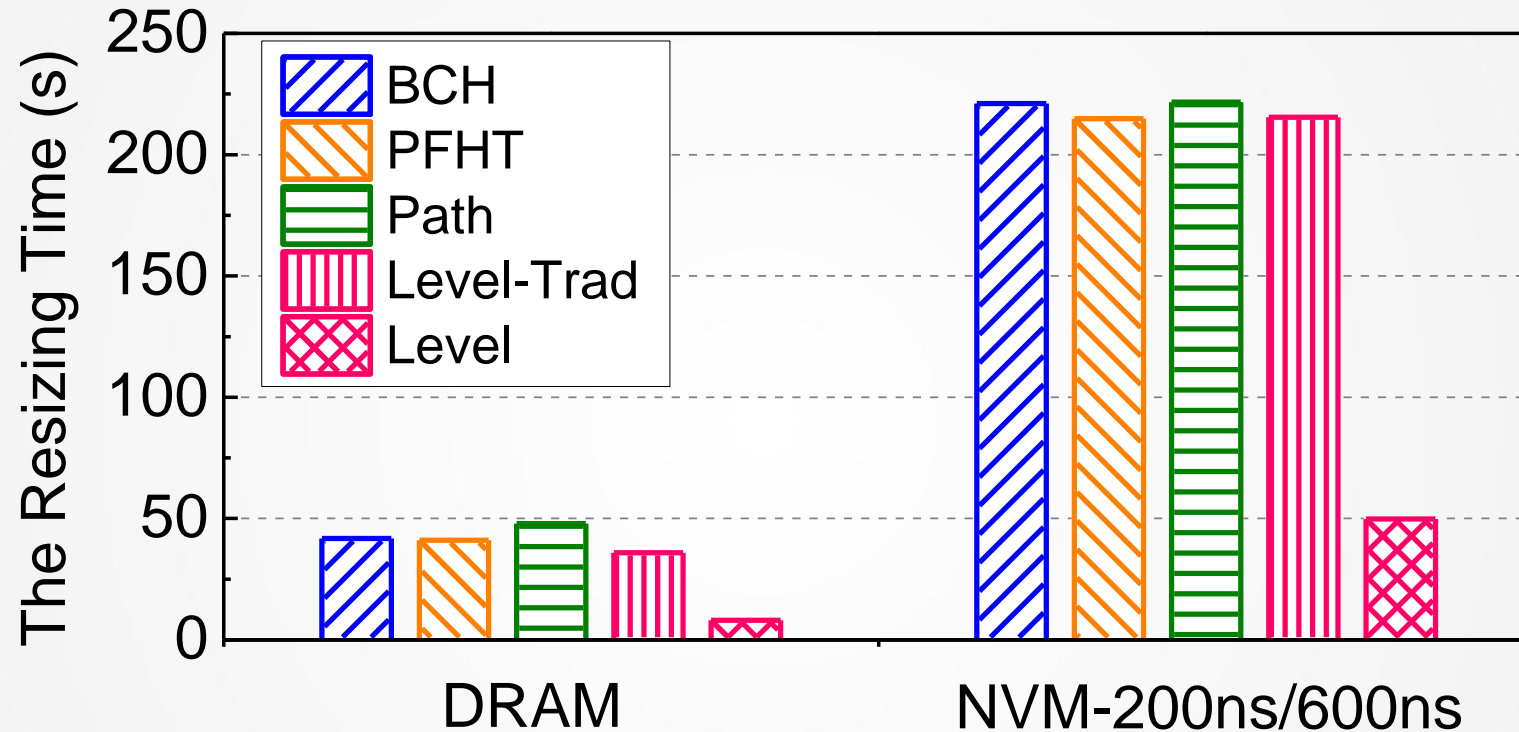
- Opportunistic log-free update scheme reduces the update latency by 15% ~ 52%, i.e., speeding up the updates by $1.2\times - 2.1\times$

Search Latency



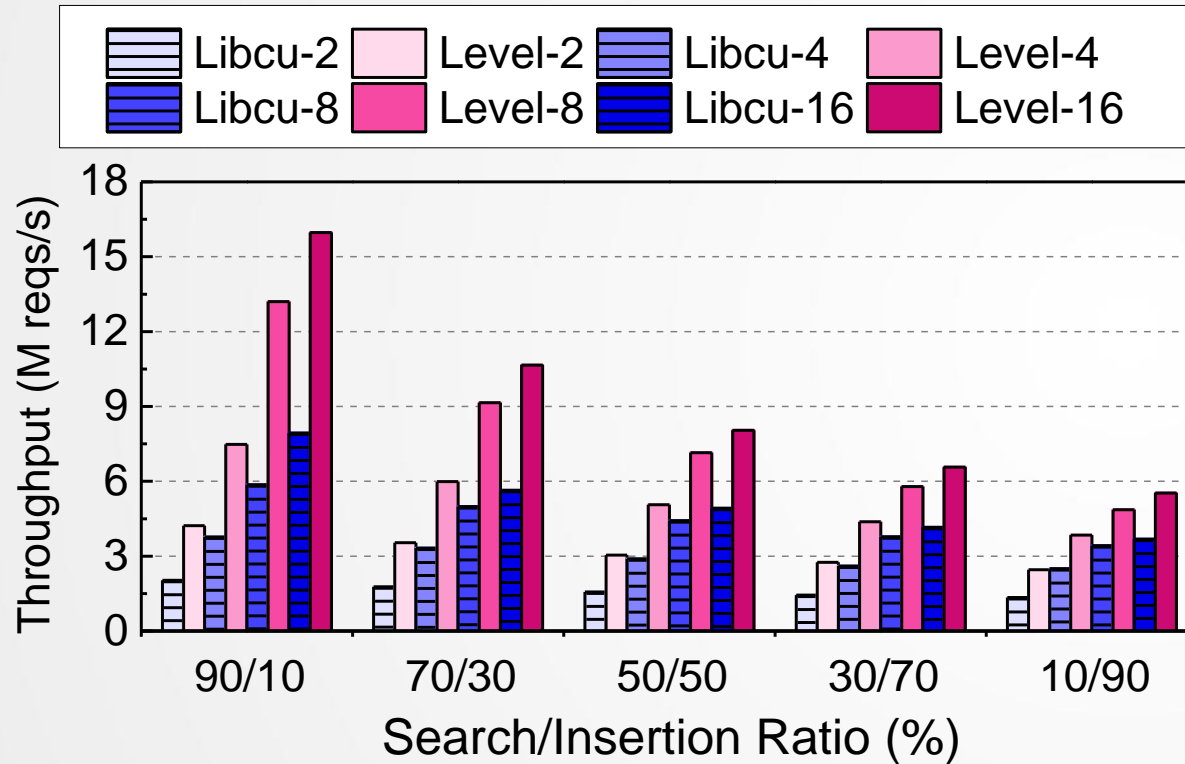
- The search latency of level hashing is close to that of BCH, which is much lower than PFHT and path hashing

Resizing Time



- Level hashing reduces the resizing time by about 76%, i.e., speeding up the resizing by $4.3\times$

Concurrent Throughput



- **Concurrent level hashing:** Support multiple-reader multiple-writer concurrency via simply using fine-grained locking
- Concurrent level hashing has $1.6\times - 2.1\times$ higher throughput than `libcuckoo`¹, due to locking fewer slots for insertions

[1] X. Li et al.. "Algorithmic improvements for fast concurrent cuckoo hashing", Eurosys, 2014.

Conclusion

- Traditional indexing techniques originally designed for DRAM become inefficient in PM
- We propose level hashing, a write-optimized and high-performance hashing index scheme for PM
 - Write-optimized hash table structure
 - Cost-efficient in-place resizing
 - Log-free consistency guarantee
- $1.4\times - 3.0\times$ speedup for insertion, $1.2\times - 2.1\times$ speedup for update, and over $4.3\times$ speedup for resizing

Thanks! Q&A

Open-source code: <https://github.com/Pfzuo/Level-Hashing>