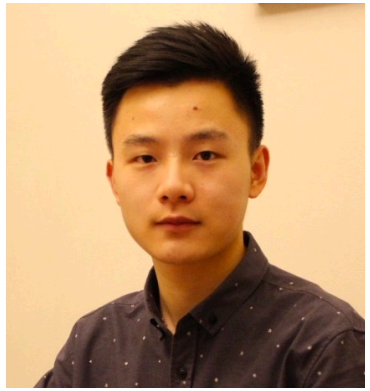
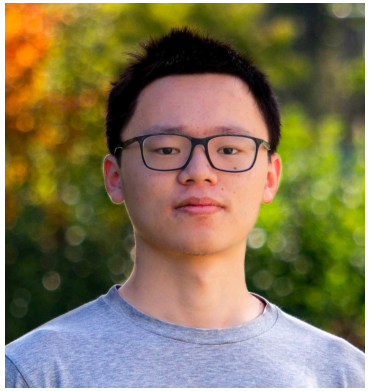
Building Scalable Dynamic Hash Tables on Persistent Memory



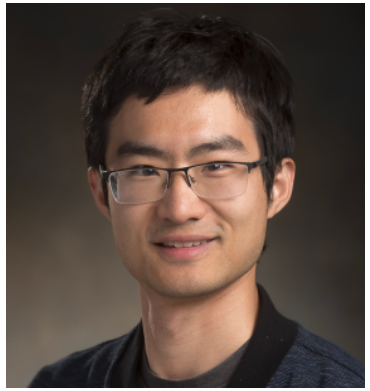
Baotong Lu

Chinese University of Hong Kong



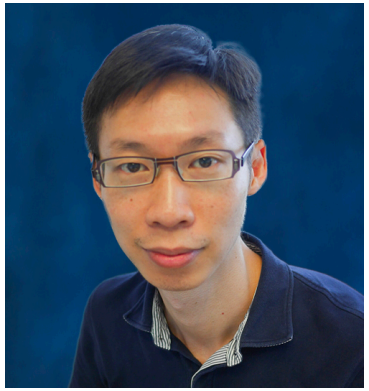
Xiangpeng Hao

Simon Fraser University



Tianzheng Wang

Simon Fraser University



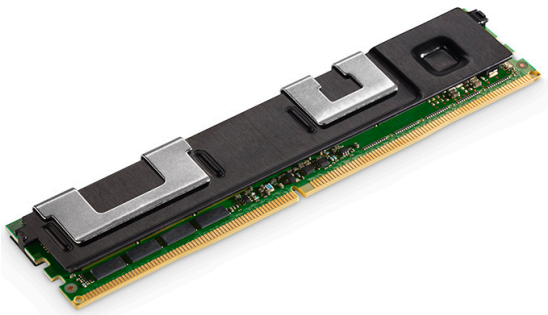
Eric Lo

Chinese University of Hong Kong

Persistent Memory Primer

- Persistent Memory (PM)
  - Bigger capacity/lower price (vs. DRAM)
  - Close-to-DRAM speed
  - Byte-addressable
  - Persistent

  } **Instant Recovery!**
- **Intel Optane DCPMM**
  - Distinct properties: **latency** and **bandwidth**



MEMCACHED redis

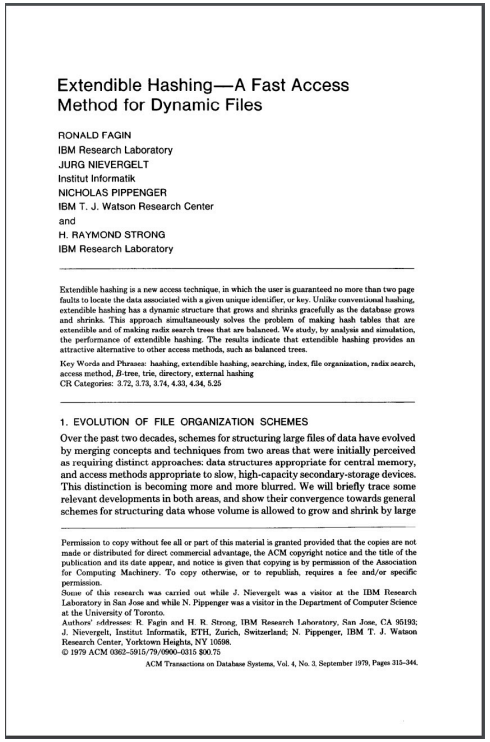
ORACLE BERKELEY DB Microsoft SQL Server

PM-based **hash table**: DRAM-based emulation vs. Optane DCPMM

Dynamic Hashing

- Extendible hashing and *Linear hashing*
  - Directory + buckets

See Paper



Extendible Hashing—A Fast Access Method for Dynamic Files

RONALD FAGIN
IBM Research Laboratory
JURG NIEVERGELT
Institut Informatik
NICHOLAS PIPPENGER
IBM T. J. Watson Research Center
and
H. RAYMOND STRONG
IBM Research Laboratory

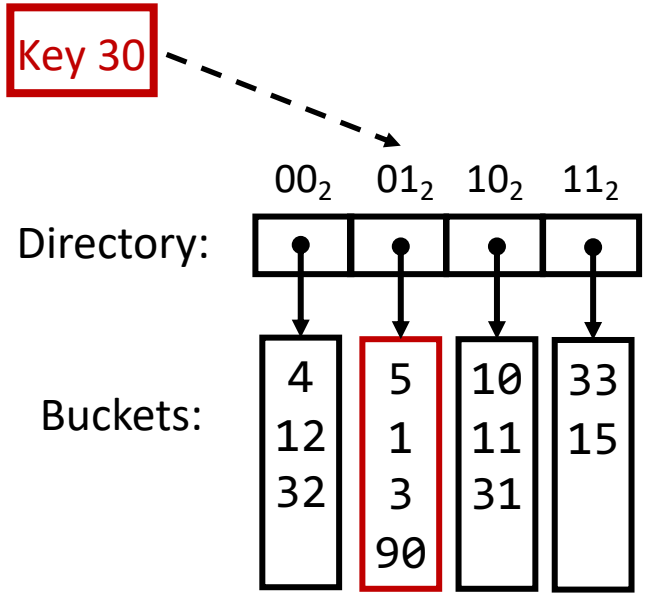
Extendible hashing is a new access technique, in which the user is guaranteed no more than two page faults to locate the data associated with a given unique identifier, or key. Unlike conventional hashing, extendible hashing has a dynamic structure that grows and shrinks gracefully as the database grows and shrinks. This approach simultaneously solves the problem of making hash tables that are extendible and of making radix search trees that are balanced. We study, by analysis and simulation, the performance of extendible hashing. The results indicate that extendible hashing provides an attractive alternative to other access methods, such as balanced trees.

Key Words and Phrases: hashing, extendible hashing, searching, index, file organization, radix search, access method, *B*-tree, trie, directory, external hashing
CR Categories: 3.72, 3.73, 3.74, 4.33, 4.34, 5.25

1. EVOLUTION OF FILE ORGANIZATION SCHEMES

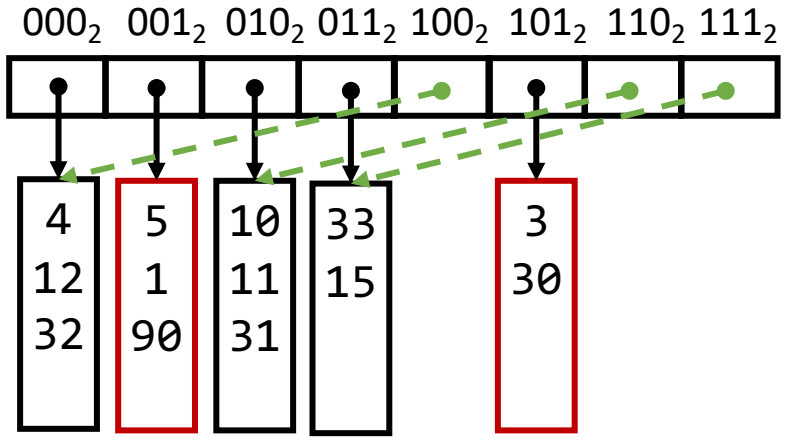
Over the past two decades, schemes for structuring large files of data have evolved by merging concepts and techniques from two areas that were initially perceived as requiring distinct approaches: data structures appropriate for central memory, and access methods appropriate to slow, high-capacity secondary-storage devices. This distinction is becoming more and more blurred. We will briefly trace some relevant developments in both areas, and show their convergence towards general schemes for structuring data whose volume is allowed to grow and shrink by large

Some of this research was carried out while J. Nievergelt was a visitor at the IBM Research Laboratory in San Jose and while N. Pippenger was a visitor in the Department of Computer Science at the University of Toronto.
Authors' addresses: R. Fagin and H. R. Strong, IBM Research Laboratory, San Jose, CA 95193; J. Nievergelt, Institut Informatik, ETH, Zurich, Switzerland; N. Pippenger, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.



Partial Expansion

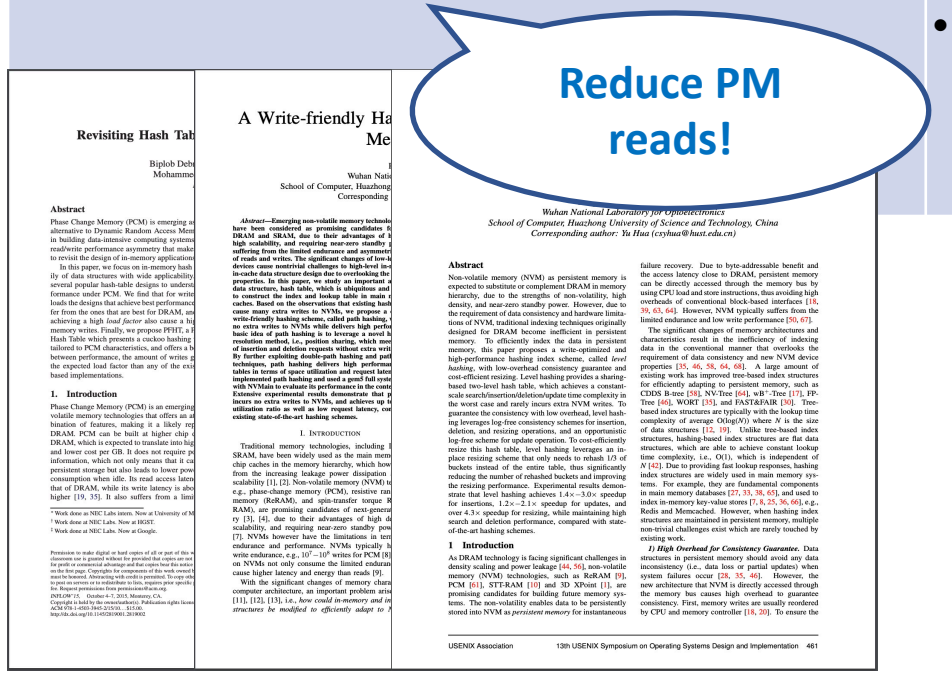
Scalable Structure!



Goals - High Throughput

Goals

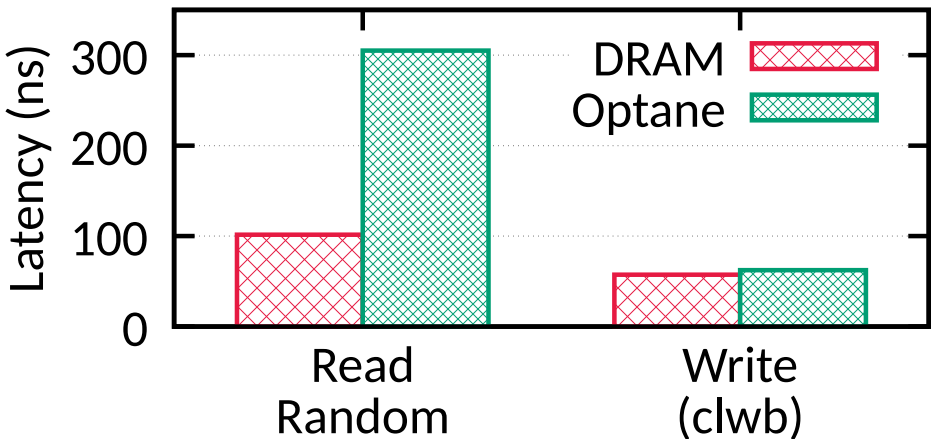
High (single-thread) throughput



Reduce PM reads!

What's wrong on emulation-based PM hashing?

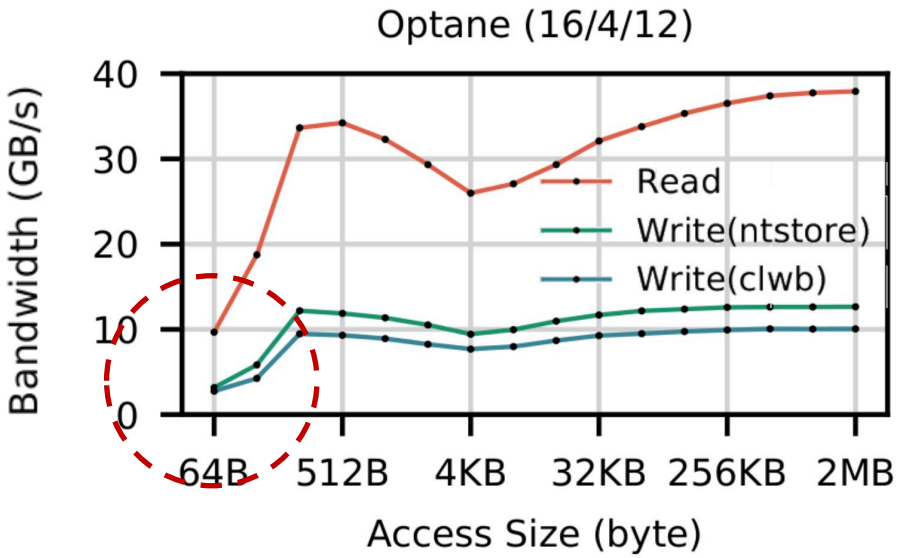
- **Optane**: End-to-end random write latency **<** read latency [1]
- Wrong assumption before:
  - Write latency **>** read latency in emulation
  - Previous write-limited hash [2, 3, 4] ==> excessive PM reads



[1] An empirical guide to the behavior and use of scalable persistent memory ***FAST'20***
[2] Revisit hash table design or phase change memory. ***INFLOW'15***
[3] A write-friendly hashing scheme or non-volatile memory***. MSST'17***
[4] Write-optimized and high-performance hashing index for persistent memory ***OSDI'18***

Goals - Scalable on Multi-core

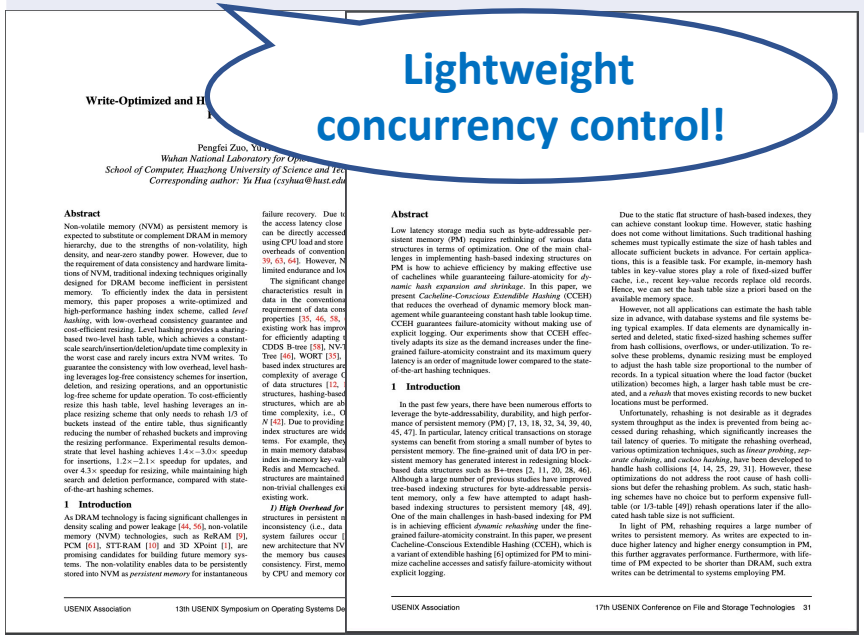
Goals	What's wrong on emulation-based PM hashing?
High (single-thread) throughput	• Excessive PM reads
Scalable on multi-core	• **Optane**: PM bandwidth is much lower than DRAM, especially for small random writes (~14X) [1]!

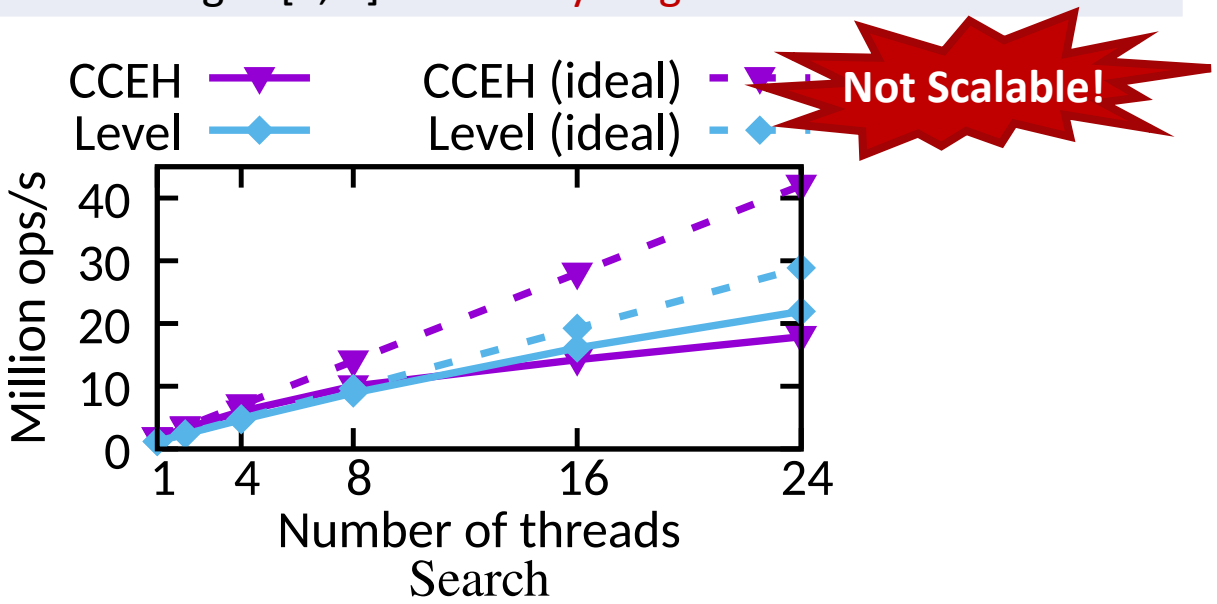


[1] An empirical guide to the behavior and use of scalable persistent memory ***FAST'20***
[2] Write-optimized and high-performance hashing index for persistent memory ***OSDI'18***
[3] Write-optimized dynamic hashing for persistent memory ***FAST'19***

Goals - Scalable on Multi-core

Goals	What's wrong on emulation-based PM hashing?
High (single-thread) throughput	• Excessive PM reads
Scalable on multi-core	• **Optane**: PM bandwidth is much lower than DRAM, especially for small random writes (~14X) [1]! • Previous designs [2, 3] ==> heavyweight read-write lock!



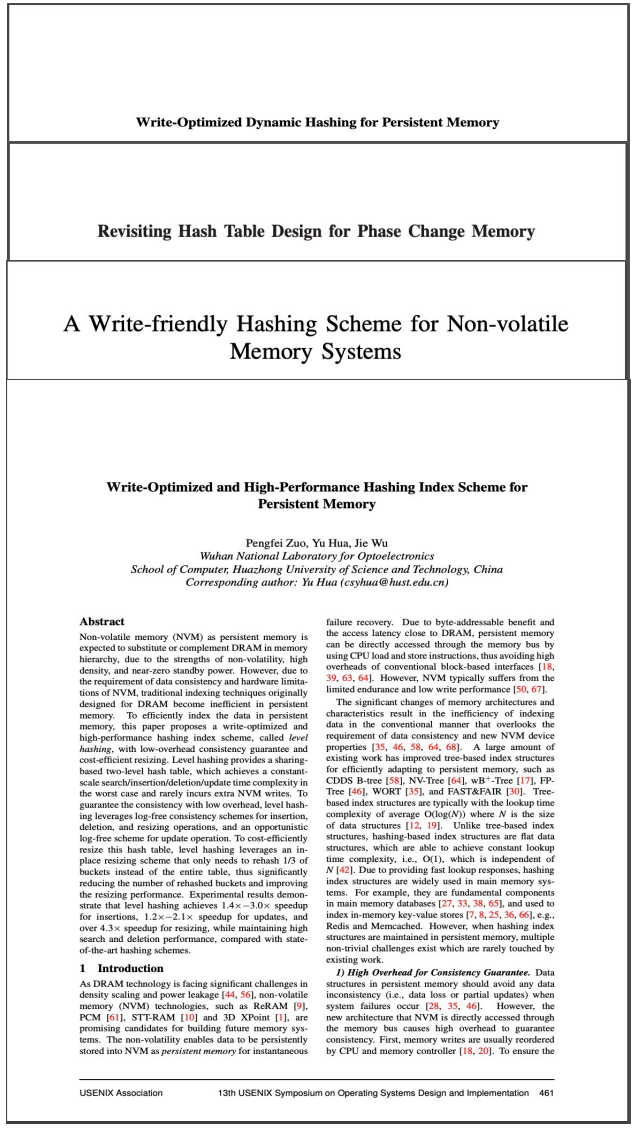


[1] An empirical guide to the behavior and use of scalable persistent memory ***FAST'20***
[2] Write-optimized and high-performance hashing index for persistent memory ***OSDI'18***
[3] Write-optimized dynamic hashing for persistent memory ***FAST'19***

Goals - Full Functionality

Goals	What's wrong on emulation-based PM hashing?
High throughput	• Excessive PM reads
Scalable on multi-core	• Heavyweight concurrency control
Instant recovery	• Recovery time = O(data size) [4]
High space utilization	• Low load factor [4]
Variable-length key	• Inefficient support [1, 2, 3, 4]

Without sacrificing any desirable properties!



Write-Optimized Dynamic Hashing for Persistent Memory

Revisiting Hash Table Design for Phase Change Memory

A Write-friendly Hashing Scheme for Non-volatile Memory Systems

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

Pengfei Zuo, Yu Hua, Jie Wu
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, China
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Non-volatile memory (NVM) as persistent memory is expected to substitute or complement DRAM in memory hierarchy, due to the strengths of non-volatility, high density, and near-zero standby power. However, due to the requirement of data consistency and hardware limitations of NVM, traditional indexing techniques originally designed for DRAM become inefficient in persistent memory. To efficiently index the data in persistent memory, this paper proposes a write-optimized and high-performance hashing index scheme, called *level hashing*, with low-overhead consistency guarantee and cost-efficient resizing. Level hashing provides a sharing-based two-level hash table, which achieves a constant-scale search/insertion/deletion/update time complexity in the worst case and rarely incurs extra NVM writes. To guarantee the consistency with low overhead, level hashing leverages log-free consistency schemes for insertion, deletion, and resizing operations, and an opportunistic log-free scheme for update operation. To cost-efficiently resize this hash table, level hashing leverages an in-place resizing scheme that only needs to rehash 1/3 of buckets instead of the entire table, thus significantly reducing the number of rehashed buckets and improving the resizing performance. Experimental results demonstrate that level hashing achieves 1.4×−3.0× speedup for insertions, 1.2×−2.1× speedup for updates, and over 4.3× speedup for resizing, while maintaining high search and deletion performance, compared with state-of-the-art hashing schemes.

1 Introduction

As DRAM technology is facing significant challenges in density scaling and power leakage [44, 56], non-volatile memory (NVM) technologies, such as ReRAM [9], PCM [61], STT-RAM [10] and 3D XPoint [1], are promising candidates for building future memory systems. The non-volatility enables data to be persistently stored into NVM as *persistent memory* for instantaneous failure recovery. Due to byte-addressable benefit and the access latency close to DRAM, persistent memory can be directly accessed through the memory bus by using CPU load and store instructions, thus avoiding high overheads of conventional block-based interfaces [18, 39, 63, 64]. However, NVM typically suffers from the limited endurance and low write performance [50, 67].

The significant changes of memory architectures and characteristics result in the inefficiency of indexing data in the conventional manner that overlooks the requirement of data consistency and new NVM device properties [35, 46, 58, 64, 68]. A large amount of existing work has improved tree-based index structures for efficiently adapting to persistent memory, such as CDDS B-tree [58], NV-Tree [64], wB$^+$-Tree [17], FP-Tree [46], WORT [35], and FAST&FAIR [30]. Tree-based index structures are typically with the lookup time complexity of average O(log(N)) where N is the size of data structures [12, 19]. Unlike tree-based index structures, hashing-based index structures are flat data structures, which are able to achieve constant lookup time complexity, i.e., O(1), which is independent of N [42]. Due to providing fast lookup responses, hashing index structures are widely used in main memory systems. For example, they are fundamental components in main memory databases [27, 33, 38, 65], and used to index in-memory key-value stores [7, 8, 25, 36, 66], e.g., Redis and Memcached. However, when hashing index structures are maintained in persistent memory, multiple non-trivial challenges exist which are rarely touched by existing work.

1) High Overhead for Consistency Guarantee. Data structures in persistent memory should avoid any data inconsistency (i.e., data loss or partial updates) when system failures occur [28, 35, 46]. However, the new architecture that NVM is directly accessed through the memory bus causes high overhead to guarantee consistency. First, memory writes are usually reordered by CPU and memory controller [18, 20]. To ensure the

USENIX Association 13th USENIX Symposium on Operating Systems Design and Implementation 461

[1] Revisit hash table design or phase change memory. ***INFLOW'15***
[2] A write-friendly hashing scheme or non-volatile memory***. MSST'17***
[3] Write-optimized and high-performance hashing index for persistent memory ***OSDI'18***
[4] Write-optimized dynamic hashing for persistent memory ***FAST'19***

Dash = Fast + Scalable + Full Functionality

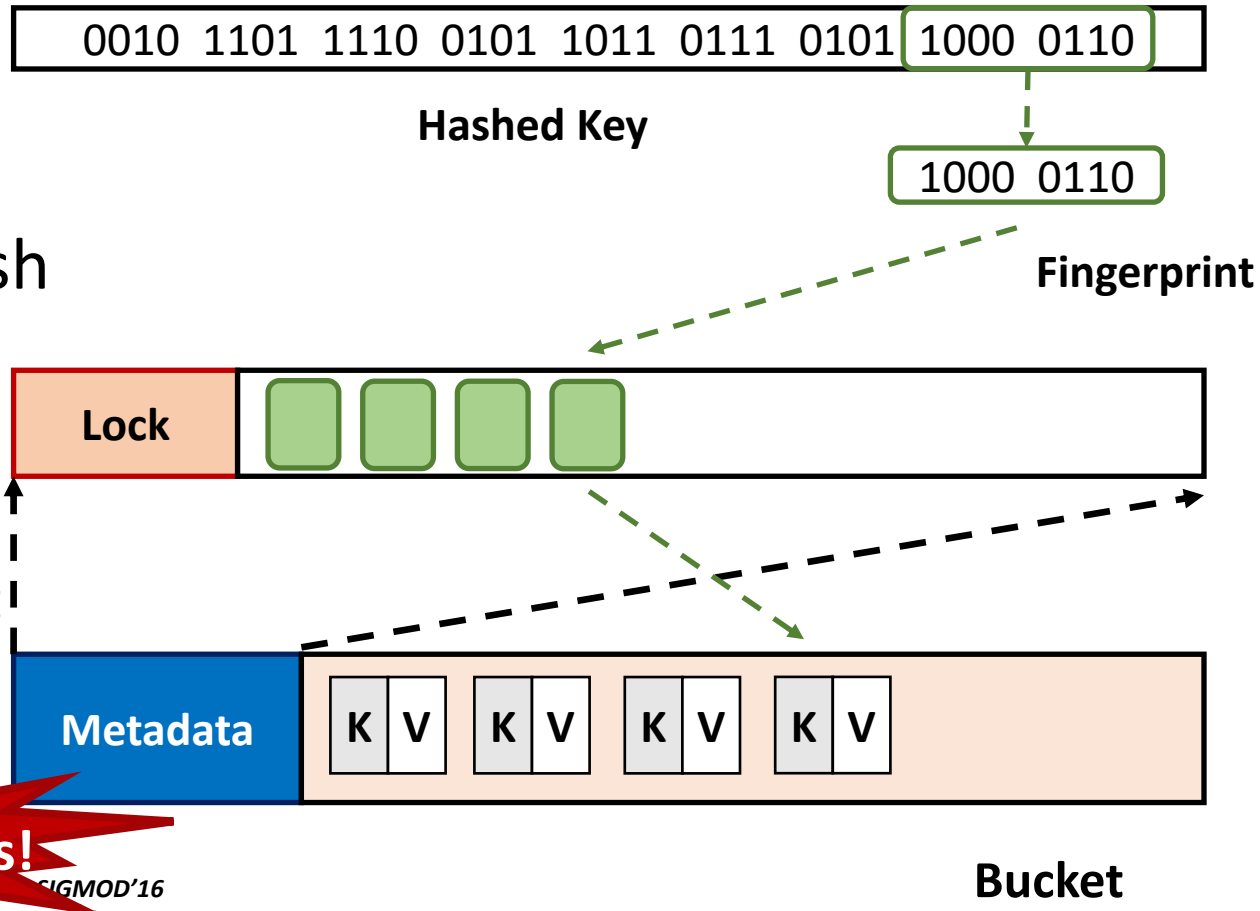
Goals	What's wrong on emulation-based PM hashing?
High (single-thread) throughput	• Excessive PM reads
Scalable on multi-core	• Heavyweight concurrency control
Instant recovery	• Recovery time = O(data size)
High space utilization	• Low load factor
Efficient support of variable-length key	• Inefficient support

Reduce both PM reads and writes + **With full functionality**

Extendible hashing and linear hashing

Problem 1: Excessive PM reads

- **Intel Optane DCPMM**
  - End-to-end read latency > write
- **Fingerprint** [1, 2]: one-byte hash
  - Filtering PM reads
- Goals achieved
  - **High (single-thread) throughput**
  - **Efficient variable-length key support**

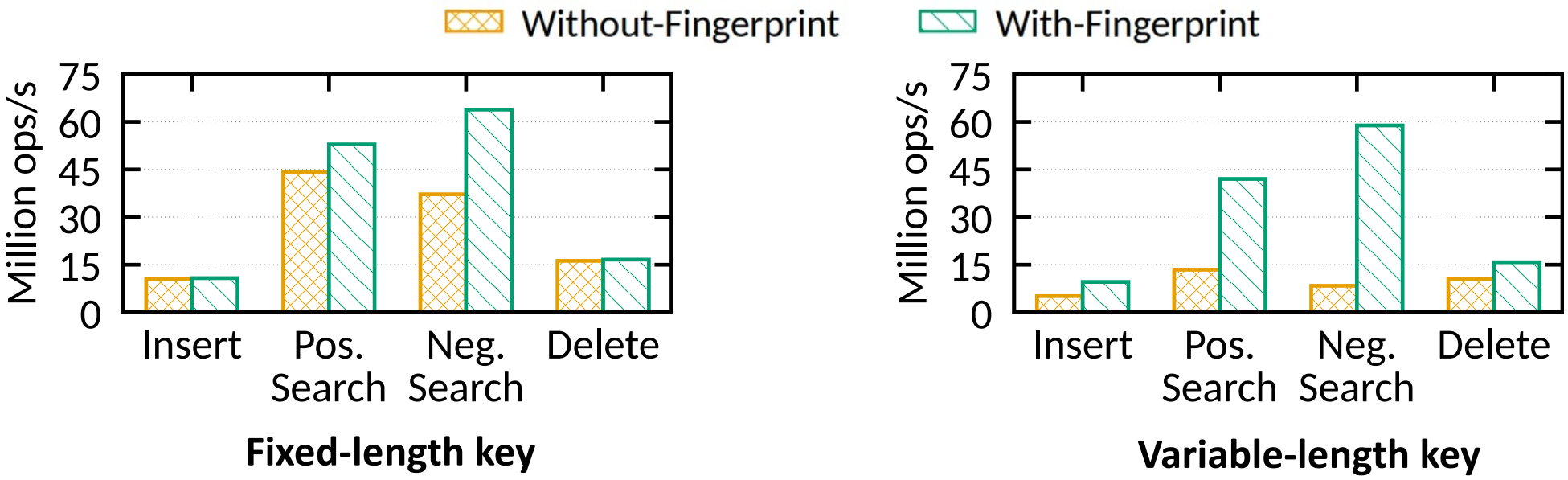


Skip most records!

[1] FPTree: A Hybrid SCM-DRAM Persistent and C… *SIGMOD'16*
[2] MemC3: Compact and Concurrent MemCache with D… Caching and Smarter Hashing. *NSDI'13*

Effectiveness of Fingerprint

- With-Fingerprint vs. Without-Fingerprint
  - The effect of reducing PM accesses
  - The effect of variable-length key support



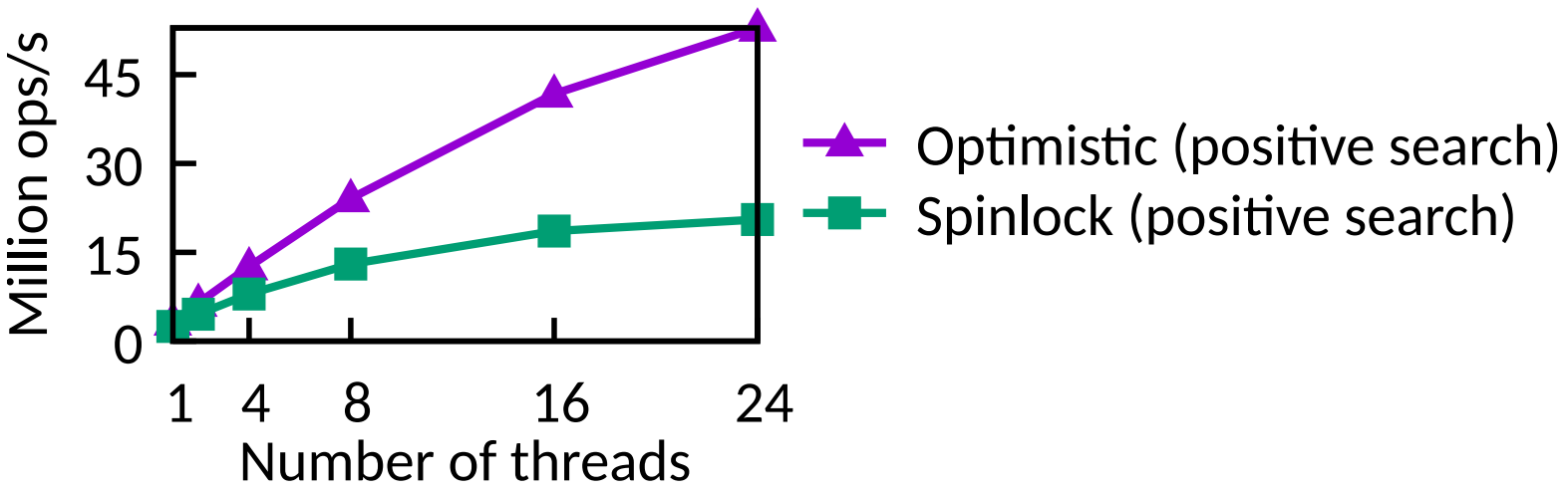
Problem 2: Heavyweight Concurrency Control

Read-write lock on PM

- RW-lock incurs writes for search
- Even the **search operations** could easily exhaust the PM bandwidth!

Dash: optimistic lock

- No writes for search operations
- Goals achieved: **scalable on multi-core** for search operations

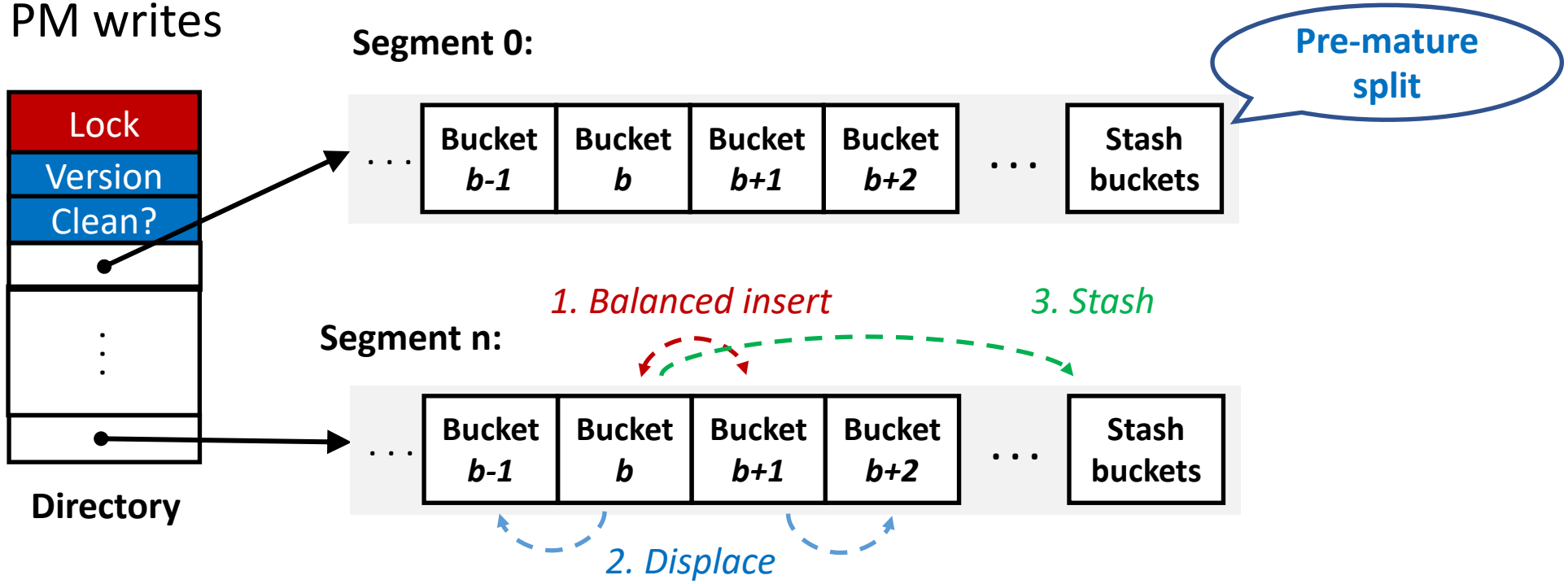


Problem 3: Low Space Utilization

Pre-mature splits

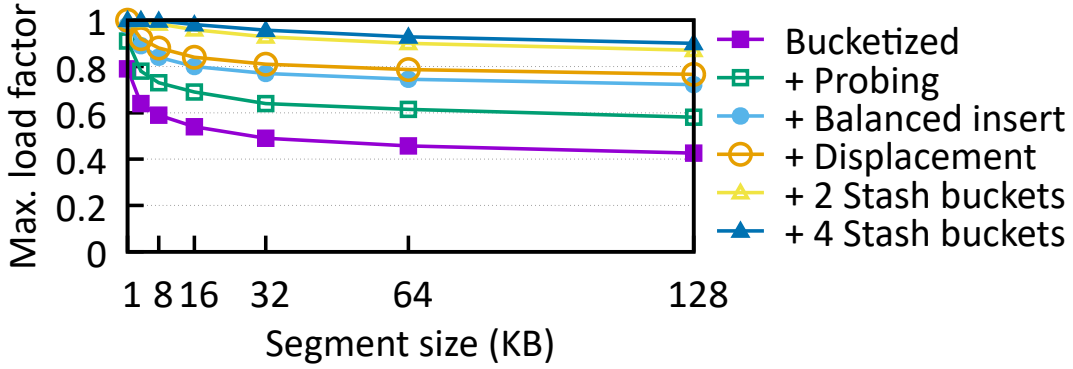
- Low space efficiency
- Excessive PM writes

Dash: bucket load balancing

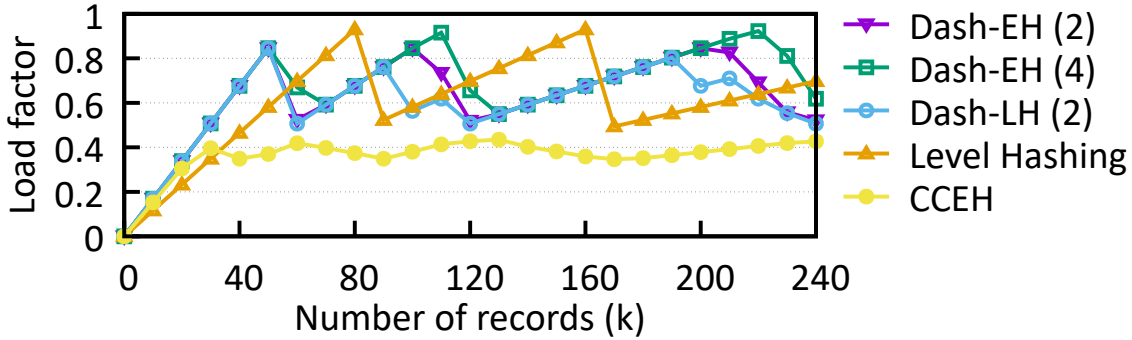


Load Factor

- Goals achieved: high load factor (> 90%).



Max. Load factor after adding different techniques



Overall Load factor comparison

Problem 4: Crash Consistency

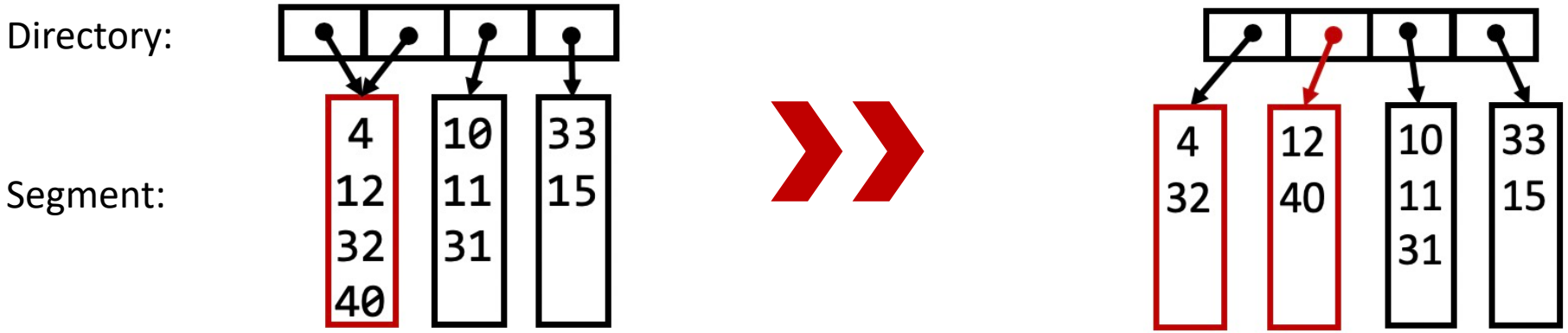
Persistency on PM

- Volatile CPU cache
  - Cacheline flush + fence
- Failure atomicity of PM is 8B
  - Writes on hash table > 8B

Dash: crash consistency design

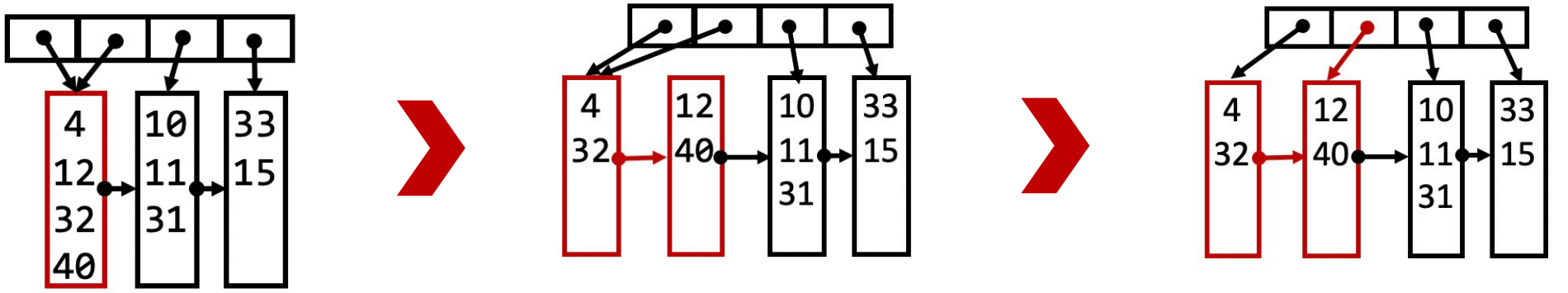
- ***SMO (segment split)***
- Basic operations (Insert/Delete)

Bitmap



Problem 4: Crash Consistency - SMO

- Fully relying on logging is heavyweight
- Recoverable segment split without logging
  - State variable: indicate which step the SMO is in
  - Side link: find the new segment



(a) Initial state.

(b) Allocate a new segment and do the rehashing.

(c) Update the directory entry and local depth.

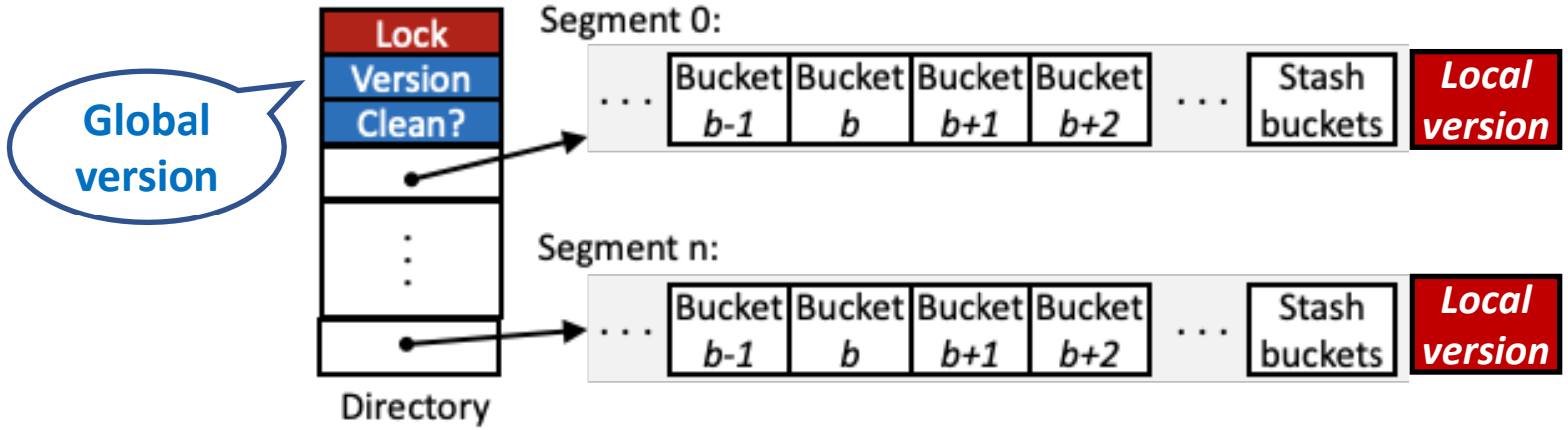
Problem 5: Non-instant Recovery

Slow recovery process

- Heavyweight recovery work
  - Locked locks
  - Ongoing SMO
  - …

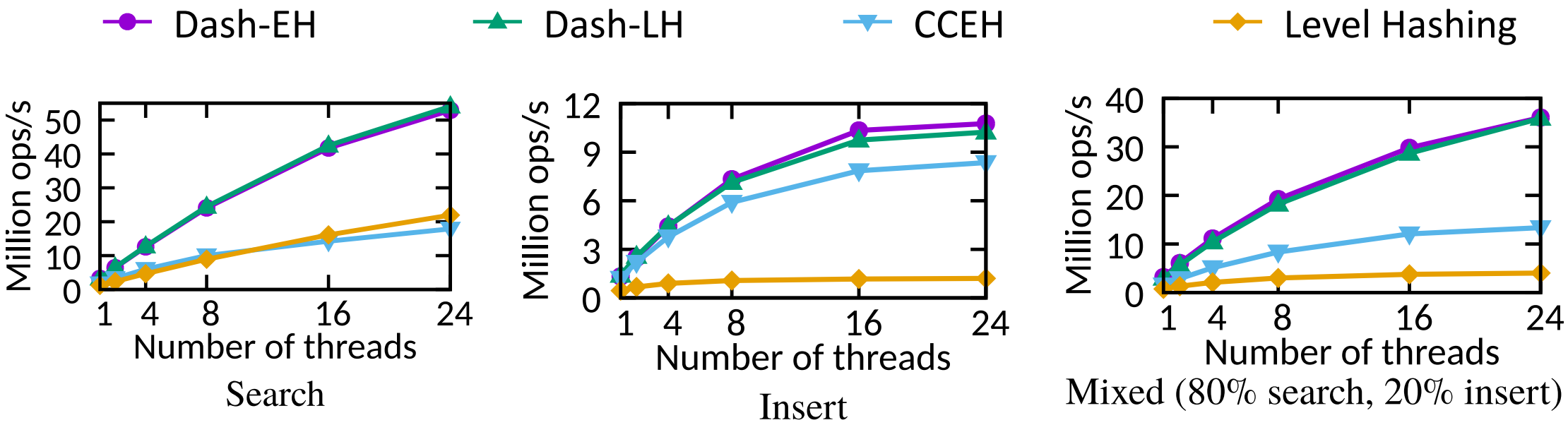
Dash: instant recovery

- Lazy recovery
  - Amortize real recovery work to runtime
- Accept requests instantly upon system restart
  - Global version and local version



Evaluation: Scalability

- Setting: 24-core CPU, 128GB X 6 Optane DCPMM (on all six channels)
- **Near-linear scalability** for **search** operation
- **Better insert performance** than state-of-the-arts

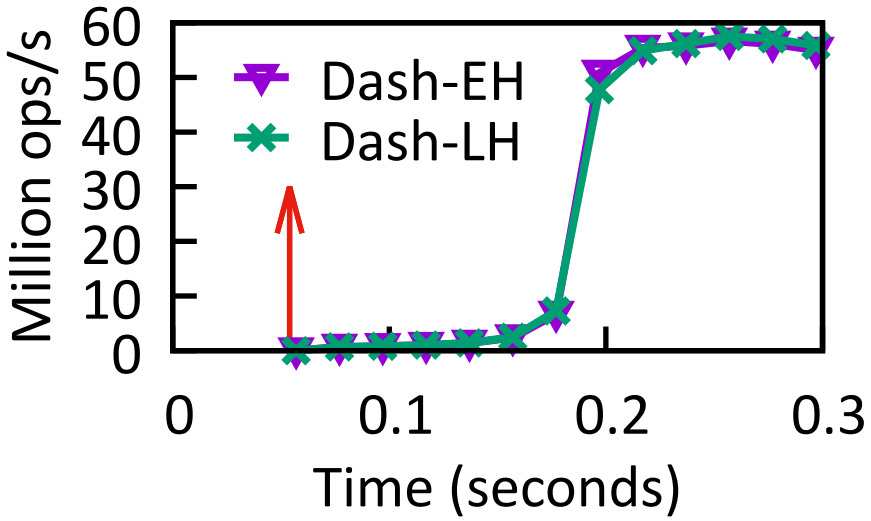


Evaluation: Recovery

- Dash: Instant recovery regardless of indexed data size
- Multi-threading helps throughput to return to normal very fast!

Recovery time (ms)

Hash Table	Number of indexed records (million)					
	40	80	160	320	640	1280
Dash-EH	57	57	57	57	57	57
Dash-LH	57	57	57	57	57	57
CCEH	113	165	262	463	870	1673
Level hashing	53	53	53	53	53	(53)



Conclusions

- Emulation-based PM hash tables considered harmful on real PM
- **Dash**: fast, scalable and full functionality hash index on real PM
- Applicability
  - **Extendible hashing/linear hashing**, also applicable to other hashing schemes
  - Ready to be integrated with KV store/DB systems
- Full-paper: *Dash: Scalable Hashing on Persistent Memory, PVLDB 2020*

Open source at: *https://github.com/baotonglu/dash*