Mirror Asymmetry Perfect Hashing: A Memory-Efficient and Load-Intensive-Optimized Hashing Index on Hybrid DRAM-PMem Architecture

Jingcheng Ju*‡§, Zirui Liu*†‖, Feng Liu*‡§, Kaicheng Yang†, Yikai Zhao†,
Tong Yang†, Guodong Yang¶, Xingchun Wang¶, Duohe Ma‡§,
†Peking University, China ‡Institute of Information Engineering, Chinese Academy of Sciences, China
§School of Cyber Security, University of Chinese Academy of Sciences, China ¶Huawei, China
‖:Project lead

2026 IEEE 42nd International Conference on Data Engineering (ICDE) | 979-8-3315-8365-1/26/$31.00 ©2026 IEEE | DOI: 10.1109/ICDE65706.2026.00212

***Abstract*—The increasing demand for high-performance key–value (KV) stores has driven the adoption of hierarchical memory architectures combining fast DRAM with persistent memory (PMem). Current solutions struggle to balance high load factors with high write performance and low DRAM overhead. This arises from the PMem–DRAM speed gap, which makes PMem-side collision resolution a performance bottleneck, and mitigating this often requires maintaining more complex DRAM-side mapping metadata. This paper proposes Mirror Asymmetry Perfect Hashing (MAPH), a cuckoo hashing-based hybrid DRAM–PMem KV store with a dynamic perfect hashing index, that achieves DRAM-only collision resolution through asymmetric position decoupling across memory tiers, enabling high load factors with minimal DRAM metadata overhead. Our approach simultaneously achieves high load factors, low latency, and high concurrent throughput. Experimental evaluation demonstrates 2.8×–13.6× lower latency and significantly higher throughput than state-of-the-art solutions, while incurring lower DRAM space overhead and competitive PMem efficiency.**

I. Introduction

Key-value (KV) stores are fundamental components of modern data infrastructure, supporting applications ranging from web services [1]–[4] to large-scale distributed systems [5]. To meet the performance and scalability requirements of large deployments, many KV storage systems adopt a hierarchical memory architecture that uses fast memory for indexing and slower, larger-capacity media for storing full KV pairs [6], [7].

The emergence of persistent memory (PMem) [8]–[10] (e.g., Intel Optane DC [11]) makes hybrid DRAM–PMem KV stores attractive, providing byte-addressable access and persistence across failures [12], [13], but also amplifies the cost of collision resolution due to expensive random PMem accesses and especially PMem writes. Perfect hashing is a special hashing index function that maps each key to a unique slot in the underlying hash table, eliminating collisions and probing [14]–[17]. In this context, this collision-free placement and the resulting constant-time lookups are particularly advantageous for hybrid DRAM–PMem KV stores and indexing structures that seek to minimize PMem accesses [18]–[20]. In a hybrid DRAM–PMem architecture, the index maintained in DRAM deterministically identifies a single target slot in the PMem-resident KV table for each existing key, so lookups do not require PMem-side collision handling such as chaining, multi-bucket probing, or record relocation.

Despite recent progress in dynamic perfect hashing for high load factors [21]–[24], implementing it efficiently on hybrid DRAM–PMem architectures remains challenging:

Challenge 1: High load factor without PMem write amplification. Under high load factors, many designs rely on relocations and conflict handling that trigger multiple PMem writes, causing tail latency and throughput degradation [25], [26].

Challenge 2: Efficient dynamic expansion. Dynamic expansion often requires rehashing and moving existing records in PMem [21], [22], which introduces high latency and severe write amplification as the table grows.

Challenge 3: Space-efficient design with fast recovery. Supporting crash consistency and recovery typically requires extra metadata in PMem [27]–[29], increasing space overhead and complicating recovery.

We propose Mirror Asymmetry Perfect Hashing (MAPH), a hybrid DRAM–PMem KV store that introduces an asymmetric, cuckoo-based perfect hashing design [30]. MAPH maintains two interlinked cuckoo hash tables: a DRAM-resident **MAPH-Index** that stores key fingerprints and PMem location information, and a PMem-resident **MAPH-Storage** that stores full KV pairs. Unlike conventional designs that enforce a strict one-to-one correspondence between index and storage positions, MAPH decouples fingerprint placement from KV placement, allowing all kick-out (relocation) operations to be performed entirely in DRAM. As a result, each successful insertion performs exactly one PMem write (writing the final KV pair), independent of the kick-out path length.

MAPH further employs (i) a copy-based expansion strategy

*Equal contribution.
Corresponding author: Duohe Ma (maduohe@iie.ac.cn)

2375-026X/26/$31.00 ©2026 IEEE
DOI 10.1109/ICDE65706.2026.00212

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

that avoids moving existing PMem KV pairs by modifying the DRAM index only, (ii) a load-balanced re-insertion recovery method that eliminates auxiliary PMem metadata, and (iii) a multi-threaded implementation that accelerates insertion, lookup, expansion, and recovery.

The main contributions of this work are:

- **Asymmetric DRAM–PMem perfect hashing with a single-write insertion guarantee.** We decouple the DRAM index from PMem storage so that collision resolution (kick-out) happens entirely in DRAM, and each successful insertion issues exactly one PMem write for the final placement.
- **PMem-optimized expansion and recovery without moving existing KV pairs**. We design a copy-based expansion method and a recovery algorithm that avoid PMem data movement and eliminate the need for auxiliary PMem metadata.
- **Extensive evaluation.** We present extensive theoretical analysis and experimental results showing that MAPH provides improved performance with lower DRAM space overhead compared to state-of-the-art approaches, without sacrificing load factor or concurrency support. All codes are open-sourced [31].

II. Background and Related Work

A. Persistent Memory

Persistent memory (PMem) provides byte-addressable, non-volatile storage with latency between DRAM and SSDs, enabling load/store access to persistent data structures without block-based I/O [32]–[35]. Intel Optane DC Persistent Memory demonstrates the practicality of this model in production systems. Consequently, hybrid DRAM–PMem architectures have become an attractive design point for KV stores: DRAM hosts hot metadata and index structures, while PMem provides large-capacity, durable storage for KV records.

From a data-structure perspective, three PMem properties are particularly relevant. First, PMem writes are significantly slower than DRAM writes, and small random writes often incur read–modify–write amplification due to PMem's coarse internal access granularity [8], [10]. Second, random PMem accesses are substantially more expensive than sequential accesses, making unpredictable probing patterns especially costly. Third, PMem systems transfer data through 64-byte CPU cachelines while the underlying media operates at a larger granularity (e.g., 256-byte XPLine) [36], so scattered updates can amplify both reads and writes. In addition, the asynchronous DRAM refresh (ADR) mechanism [37] ensures that PMem writes residing in the memory controller buffers are persisted after system power failures. However, applications must explicitly manage cache flush instructions (e.g., `clflush`, `clflushopt`, and `clwb`) and enforce appropriate memory ordering to guarantee crash consistency [38].

These characteristics motivate PMem-aware KV-store designs that minimize PMem writes, avoid random access patterns on PMem, and keep complex or iterative operations in fast volatile memory whenever possible.

B. Hashing-based Index Structures for PMem KV Stores

Hash-based indexing is widely adopted in PMem KV stores due to its efficient lookup and relatively simple, cache-friendly layouts. Compared with tree-based structures, hash tables avoid deep pointer chasing, which is particularly beneficial in PMem environments where random access latency is high.

A number of PMem-oriented hash indexes have been proposed. Level Hashing [23] uses a multi-level bucket organization to reduce resizing cost and improve space utilization. CCEH [12] adopts a cacheline-friendly extendible hashing layout with concurrent growth, while Clevel Hashing [24] further improves concurrency via lock-free operations and asynchronous resizing. These designs focus on reducing write amplification and ensuring crash consistency under PMem constraints.

Beyond PMem-only designs, recent systems explore hybrid DRAM–PMem architectures that explicitly decouple volatile index structures from persistent data. Halo [26] represents an early and influential example: it places hash metadata and structural information in DRAM, while storing KV records in PMem using a log-structured layout. By converting fine-grained random updates into sequential PMem writes, Halo significantly reduces write amplification and achieves fast recovery through periodic snapshots and logging. However, Halo's log-structured design introduces additional indirection during lookups and relies on background compaction, which can impact performance under high update rates.

Viper [18] further refines the hybrid design philosophy by carefully co-designing data layout and access patterns for PMem. Viper employs a DRAM-resident hash index (offset map) to locate PMem-resident KV records, and emphasizes PMem-optimized techniques such as segment-based allocation, sequential writes, and balanced thread-to-DIMM mapping. These optimizations improve bandwidth utilization and scalability, particularly for write-intensive workloads. Nevertheless, as load increases, insertions and updates may still trigger additional PMem accesses for probing, relocation, or metadata maintenance, limiting performance under high occupancy.

Despite their effectiveness, both PMem-oriented and hybrid hashing designs share a common limitation: collision resolution and record relocation cannot be fully isolated from PMem. As load factors increase, the frequency and complexity of collision handling grow, leading to additional PMem reads and writes and exposing a fundamental tension between high space utilization and efficient insertion performance.

C. Perfect Hashing for Hybrid DRAM–PMem Architectures

Perfect hashing aims to eliminate collisions entirely by ensuring that each key maps to a unique location, providing bounded-time lookup without collision-resolution overhead. This property is particularly attractive in hybrid DRAM–PMem systems, where avoiding collision handling directly translates into fewer expensive PMem accesses.

Recent dynamic perfect hashing schemes achieve high load factors while maintaining collision-free guarantees. MapEm-

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

bed [21] targets fast–slow memory environments and uses movement-based insertion to resolve conflicts. While effective for reducing slow-memory lookups, such movements can translate into multiple slow-memory writes when full records reside in the slow tier. EEPH [22] specifically targets DRAM–PMem hybrid systems by placing index structures in DRAM and storing KV records in PMem. It introduces a complement-move algorithm to reduce PMem writes during insertion. However, under high load factors, EEPH still requires PMem-involved probing or relocation to identify available positions, making insertion performance sensitive to occupancy.

Cuckoo hashing [39]–[41] provides a promising foundation for addressing these challenges and can be viewed as a practical realization of dynamic perfect hashing. Under successful insertions, it enforces a one-to-one mapping between keys and storage locations, thereby achieving perfect hashing properties [42], [43].

Specifically, each key is associated with a small, fixed number of candidate buckets, which guarantees bounded lookup time. Insertions resolve conflicts through kick-out chains, where existing keys are relocated to their alternative locations. Importantly, this design cleanly separates lookup—requiring probes to only a constant number of locations—from insertion, which encapsulates all relocation complexity [44], [45].

This separation motivates our key insight: if collision resolution can be carried out entirely on compact metadata in DRAM, expensive PMem accesses during insertion can be eliminated regardless of load factor. Building on cuckoo hashing, our design decouples DRAM-resident fingerprints and placement decisions from PMem-resident KV storage. Kick-out operations and conflict resolution occur solely in DRAM, while PMem updates are minimized and localized. This approach enables high load factors without sacrificing insertion performance, addressing a core limitation of prior PMem hash indexes.

TABLE I
SYMBOLS FREQUENTLY USED IN THIS PAPER.

Symbols	Meaning
m	Number of buckets in each bucket array
D_d	Number of slots in each DRAM bucket
D_p	Number of slots in each PMem bucket
f	Fingerprint length (bits)
L	Maximum kick-out path length
B_i	The i-th bucket array in PMem (MAPH-Storage), $i \in \{1, 2\}$
I_i	The i-th bucket array in DRAM (MAPH-Index), $i \in \{1, 2\}$
$h_i(\cdot)$	Hash function mapping a key to a bucket index
$FP(\cdot)$	Function computing an f-bit fingerprint of a key

III. DESIGN AND IMPLEMENTATION

A. Algorithm Overview

We propose Mirror Asymmetry Perfect Hashing (MAPH), a hybrid DRAM–PMem indexing structure based on bucketized cuckoo hashing (Figure 1). We first define the data structure and the key design principles.

Data structure. MAPH consists of a DRAM-resident index (**MAPH-Index**) and a PMem-resident storage (**MAPH-Storage**). The index contains two bucket arrays I_1 and I_2, and the storage contains two bucket arrays B_1 and B_2. Each bucket array has m buckets. Each DRAM bucket has D_d slots; each slot stores $\langle FP(key), PMem_loc_info \rangle$, where $FP(key)$ is an f-bit fingerprint and $PMem_loc_info$ encodes the target PMem location (1 bit for selecting B_1 vs. B_2 and 7 bits for the slot index within the selected PMem bucket). Each PMem bucket has D_p slots ($D_p > D_d$), and each slot stores a full KV pair $\langle key, value \rangle$.

Candidate buckets. For a key key, we compute two candidate bucket indices: $h_1(key) = H(key) \bmod m$ and $h_2(key) = (h_1(key) \oplus H'(FP(key))) \bmod m$, where $H(\cdot)$ is the primary hash and $H'(\cdot)$ derives an additional hash value from the fingerprint. The key can be indexed in DRAM buckets $I_1[h_1(key)]$ and $I_2[h_2(key)]$, and stored in PMem buckets $B_1[h_1(key)]$ and $B_2[h_2(key)]$.

Design principles. Conventional hybrid designs often enforce that an entry occupies corresponding positions across the index and storage, which couples kick-out relocation with PMem updates. MAPH breaks this coupling via two principles:

Mirror (shared bucket indices): Both layers use the same bucket indices $h_1(key)$ and $h_2(key)$, so a lookup probes only the two candidate buckets computed from the key.

Asymmetry (decoupled array/slot choice): The DRAM index position of a fingerprint does not determine the PMem position of the full KV record. Instead, the DRAM slot explicitly records $PMem_loc_info$, allowing the system to choose bucket array and slot positions independently in DRAM and PMem.

Kick-out chains are executed on DRAM fingerprints and location metadata, while PMem records are not relocated during collision handling. Therefore, each *successful* insertion writes the KV pair to PMem once (to the final chosen PMem slot), regardless of the kick-out path length in DRAM. Lookups probe two DRAM buckets and verify the key only at the PMem locations pointed to by matching fingerprints.

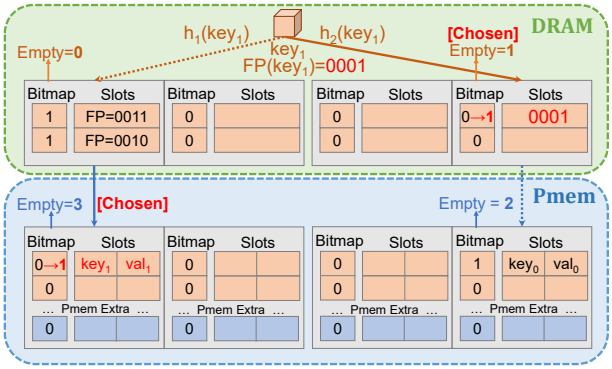


Fig. 1. Data structure and direct insertion.

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

B. Basic Operations

Insertion: To insert a KV pair $e = \langle key, value \rangle$, MAPH first computes the two candidate bucket indices $h_1(key)$ and $h_2(key)$.

PMem capacity check. MAPH treats PMem capacity as the hard constraint for insertion. Each PMem slot embeds a validity bit indicating whether it contains a valid key–value pair, and a complete one-to-one cached copy of these validity bits is maintained in DRAM for fast access. Using this DRAM-resident metadata, MAPH checks the availability of the two candidate PMem buckets $B_1[h_1(key)]$ and $B_2[h_2(key)]$ without accessing PMem. If neither bucket has a free slot, the insertion aborts and triggers expansion, regardless of available DRAM space.

Direct insertion. If at least one candidate PMem bucket has a free slot, MAPH next examines DRAM availability in the corresponding index buckets $I_1[h_1(key)]$ and $I_2[h_2(key)]$. When both layers have available capacity, MAPH performs independent load balancing in DRAM and PMem by selecting, for each layer, the bucket array with more free slots. Due to this asymmetric selection, the chosen DRAM and PMem bucket arrays may differ, as illustrated in Figure 1. Once a key–value pair is written to a PMem slot, the in-slot validity bit is updated atomically within the same PMem write, and the corresponding DRAM-cached copy is updated accordingly. This ensures that PMem remains the authoritative source of slot validity while allowing insertion decisions to be made entirely using DRAM-resident metadata.

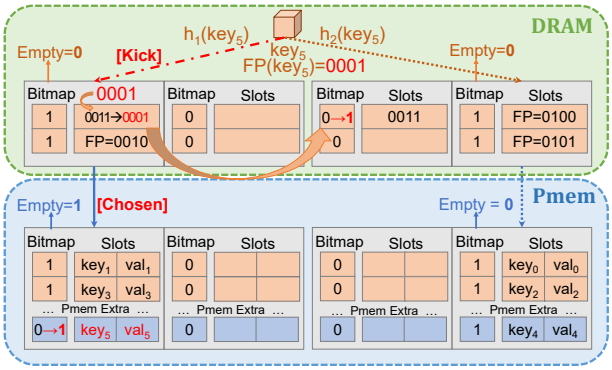


Fig. 2. Handling collision with kick-out.

DRAM-only kick-out: If PMem has a free slot but both DRAM candidate buckets are full, we initiate a kick-out operation in the DRAM index using a slot-by-slot DFS strategy [46] (Figure 2). We iteratively evict a fingerprint from one of the two candidate DRAM buckets and attempt to place it into its alternate bucket; this repeats until an empty DRAM slot is found or the maximum path length L is reached. Throughout the kick-out search, MAPH operates only on DRAM fingerprints and their location metadata, and performs no PMem reads. Upon success, the new KV pair is written once to the selected PMem slot, and the corresponding DRAM slot stores $\langle FP(key), PMem_loc_info \rangle$.

Fingerprint Analysis: Fingerprints are widely used in KV stores and hash-based indexes to reduce fast-memory space consumption by avoiding caching full keys. In hybrid DRAM–PMem architectures, they primarily serve as a space-efficient filter to reduce unnecessary accesses to slow memory.

Viper [18] follows this design by storing 64-bit fingerprint in DRAM to identify candidate records in PMem, checking for equality only if the fingerprint matches. In such designs, fingerprints must be sufficiently long to keep collision probability low, since frequent collisions translate directly into extra PMem accesses.

In cuckoo hashing–based indexes, fingerprints also participate in collision resolution: they are used to derive alternate bucket locations during kick-out operations. Fingerprint collisions within a bucket therefore introduce ambiguity and may require auxiliary mechanisms (e.g., rely on a stash) or more complex fingerprint schemes [6], [40], [47], [48], incurring additional space or time overhead.

MAPH adopts a different trade-off. Its DRAM-based cuckoo index allows duplicate fingerprints within the same bucket. Fingerprint matching in DRAM eliminates non-existent keys, while collisions where several fingerprints match the same candidate are resolved by full key matching in PMem. This trades extra DRAM consumption for a small number of PMem reads; because PMem reads are served at a coarser granularity than CPU cachelines, verifying only a few candidate slots within a bucket can be amortized, enabling more aggressive fingerprint compression.

As a result, MAPH significantly reduces DRAM footprint while maintaining high performance. Our evaluation shows that MAPH with 16-bit fingerprints outperforms Viper using its default 64-bit fingerprints, and that further reduction in fingerprint length has limited impact on performance.

Key Lookup: The lookup process computes $FP(key)$ and checks the two candidate DRAM buckets $I_1[h_1(key)]$ and $I_2[h_2(key)]$. If no matching fingerprint exists, the lookup returns negative without accessing PMem. Otherwise, for each matching DRAM slot, we follow its $PMem_loc_info$ to read the corresponding PMem slot and perform exact key matching.

Perfect Hashing Property: MAPH maintains a perfect-hashing-style guarantee at the PMem tier: for each stored key, the DRAM index deterministically identifies a unique target slot in MAPH-Storage. This guarantee is preserved because collision resolution is handled in the DRAM index (via kick-out), while PMem records are written only to a selected free slot and are never relocated.

Update and Delete Operations: Both operations begin with the lookup procedure above. For updates, we overwrite the value at the located PMem slot and persist the change. For deletions, we mark the slot as deleted by updating the corresponding PMem validity flag as needed.

Space Overhead Analysis: MAPH introduces redundancy in both memory layers. In DRAM, each MAPH-Index slot stores a fingerprint plus location metadata to identify the corresponding record in MAPH-Storage (8 bits: 1-bit table index + 7-bit slot index). In PMem, MAPH-Storage provisions

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

a larger bucket depth than the DRAM index ($D_p > D_d$) because PMem records are never kicked out/relocated; all collision resolution is handled in DRAM, and PMem absorbs load imbalance via static slack space. Empirically, allocating only 1–2 extra slots per PMem bucket is sufficient to achieve high load factors (typically $> 95\%$, and $> 97\%$ with $D_d = 8$ in Figure 4(a)). Given that PMem costs significantly less than DRAM and is typically deployed at TB-scale while DRAM is configured at GB-scale in modern data centers, this minimal PMem redundancy represents an economically efficient optimization strategy.

C. Dynamic Expansion

Existing solutions expand hash tables by moving items in slow memory, introducing significant overhead. Our algorithm employs a memory copy-based expansion approach supporting arbitrary integer scaling ratios r. When expansion is triggered, we allocate new memory spaces in both DRAM and PMem that are r times the original size, then copy the original data r times with appropriate labeling.

We modify hash functions to accommodate the new table size. The original hash functions $h_1(key) = H(key) \bmod m$ and $h_2(key) = (h_1(key) + hash(FP(key))) \bmod m$ are updated to $h'_1(key) = H(key) \bmod (r \cdot m)$ and $h'_2(key) = (h_1(key) + hash(FP(key))) \bmod (r \cdot m)$.

Due to our decoupled design, we separately check and remove redundant items in both memory layers. For each KV pair $\langle key_i, value_i \rangle$ in PMem bucket B_j, we compute $bucket_{new} = h'_1(key_i) = H(key_i) \bmod (r \cdot m)$. If $bucket_{new} \neq j$, we mark the slot as empty in the PMem bitmap. For DRAM buckets, we first retrieve the corresponding key using location information, then perform the same computation and mark the DRAM bitmap accordingly.

We provide both active check (immediate examination) and lazy check (deferred until bucket access) modes.

Example(Figure 3): We illustrate expanding our hash table by $r = 3$ times with $m = 2$ buckets in both DRAM and PMem. The expansion allocates new memory spaces of size $r \cdot m = 6$ buckets, copying the original 2 buckets 3 times. To clean redundant items in PMem bucket B_1, we examine each KV pair directly. For $\langle key_1, value_1 \rangle$ with $H(key_1) = 9$, we compute $bucket_{new} = 9 \bmod 6 = 3$. Since $bucket_{new} = 3 \neq 1$, we mark the slot as empty in the PMem bitmap.We do not delete the actual item in MAPH-Storage, which will eventually be overwritten by subsequent inserted items. For DRAM fingerprints, we retrieve keys through location information, then perform the same checking.

Redundancy removal is implemented entirely via bitmap marking; it does not perform any actual KV deletions in PMem, thereby keeping the DRAM–PMem state consistent and avoiding expensive PMem writes during expansion.

D. Recovery

Our algorithm supports recovery from both normal shutdowns and unexpected system crashes. Since our PMem layer is relocation-free, once a KV record is placed into a PMem slot, it is never migrated to a different PMem slot during collision resolution or resizing. This enables uniform handling of both recovery scenarios by reconstructing the DRAM index based on KV pairs stored in PMem.

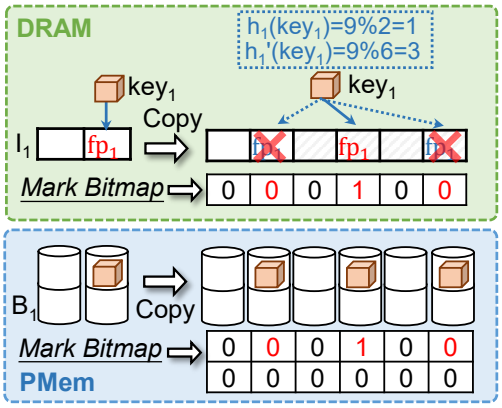


Fig. 3. Example of dynamic expansion.

Our decoupled design enables DRAM index construction through straightforward re-insertion methodology. We traverse each key in PMem and perform DRAM-only re-insertion, locating available slots within corresponding candidate buckets to store fingerprints and PMem location information. Regardless of insertion order, each fingerprint will be stored in one of the candidate buckets within the two Bucket Arrays computed by h_1 and h_2, ensuring that the system's normal functionality remains unaffected.

We can obtain the hash distribution of all keys' Bucket indices based on the PMem KV storage results. Building upon this foundation, we can further perform load-balanced insertion, such as sequentially selecting the first key that has not completed re-insertion from each bucket for insertion. This approach can significantly reduce the occurrence of collisions.

We establish the correctness of re-insertion through the following theorem: Given an initial valid insertion configuration, subsequent re-insertion of identical keys in arbitrary order will never result in insertion failure. This can be proven by contradiction: if a key insertion fails due to circular kick-out, it implies more keys than available slots, contradicting the initial valid configuration. Furthermore, our fingerprint collision handling strategy allows identical fingerprints to coexist within the same bucket, ensuring that regardless of insertion order during recovery, subsequent query operations and results maintain perfect consistency. This permissive fingerprint approach eliminates order-dependent recovery issues while preserving the correctness of all index operations.

Since the recovery process involves exclusively PMem read operations and DRAM manipulations, our algorithm maintains both algorithmic simplicity and computational efficiency.

E. Crash Consistency

This subsection specifies the persistence invariants and restart rules under unexpected crashes. MAPH follows a simple principle: PMem contents define the durable state, while

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

all DRAM-resident metadata can be discarded and rebuilt after restart.

PMem bitmap semantics. MAPH represents PMem slot occupancy using validity bits embedded within each PMem slot, which logically form a PMem-resident bitmap serving as the sole authority for record visibility. For fast insertion decisions, MAPH maintains in DRAM a complete, contiguous, one-to-one cached copy of this bitmap, enabling empty-slot detection without accessing PMem.

Insertion into PMem performs a single persistent write of the entire slot, persistently updating both the key–value record and its validity bit in one operation (e.g., via `pmem_memcpy_persist`). If a crash occurs before this write completes, the slot remains invalid and is ignored during recovery; if it completes, both record and status are guaranteed durable. Thus, failures cannot produce partially visible records. After a crash, the DRAM bitmap is rebuilt by scanning PMem slots and reading their embedded validity bits.

Expansion across crashes. To avoid inconsistent interpretations of PMem during dynamic expansion, MAPH maintains a small persistent *expansion-state metadata* that identifies the active epoch and the corresponding PMem regions. Expansion proceeds in two phases: (i) persist the new parameters and PMem region addresses, and (ii) after the new regions are populated, persistently switch the active epoch. On restart, the system selects the active epoch accordingly: crashes before the switch retain the old epoch, while crashes after the switch use the new one.

Repairability of redundancy invalidation. Crashes during expansion may leave some redundant PMem slots temporarily marked as valid. Such states are repairable because redundancy is determined solely by the active epoch. For any record $\langle key_i, value_i \rangle$ in bucket B_j, the system recomputes $H(key_i) \bmod (r \cdot m)$; if the resulting bucket differs from j, the record is obsolete and its validity bit can be cleared during recovery or lazily upon access. Therefore, interrupted invalidation affects only bookkeeping cost, not correctness.

Restart procedure. After determining the active epoch, MAPH rebuilds the DRAM index by scanning PMem and re-inserting all records whose validity bits are set. Because record visibility is defined solely by the PMem bitmap and PMem is relocation-free, the same recovery procedure applies to both normal shutdowns and unexpected crashes.

F. Multi-threading Support

MAPH supports concurrent operations using DRAM-only synchronization. Because each DRAM index entry explicitly records the target PMem slot (via location metadata), consistency can be enforced by locking only the DRAM-side bucket state (fingerprints and bitmaps), without any PMem-side coordination.

Insertion critical section. An insertion consists of three steps: (1) Reserve a slot by selecting a target DRAM entry and its mapped PMem slot and atomically updating the corresponding bitmaps in DRAM. (2) Write the key–value record to the reserved PMem slot. (3) Write the fingerprint and location metadata to the reserved DRAM entry.

The critical section is limited to step (1). Once a slot is reserved, steps (2)–(3) are collision-free because no other thread can claim the same DRAM or PMem positions.

Locking and consistency. *1) Direct insertion.* We acquire the lock of the target DRAM bucket only while reserving a free entry (step (1)) and release it immediately after the bitmap update. The subsequent PMem and DRAM writes are performed outside the lock.

2) Kick-out (DFS) insertion. During the kick-out search, we use bucket-granularity locks and hold at most two bucket locks at any time: the current bucket and the next bucket to visit. Before moving forward, we acquire the next bucket lock and then release the previous one. This bounds the lock footprint and avoids long lock chains.

3) Mixed reads and writes. We use bucket-granularity read–write locks on the DRAM index. Lookups take shared locks on the relevant buckets to read fingerprints and metadata consistently, while updates (insert/delete) take exclusive locks to modify bitmaps and entries.

Expansion and recovery. Expansion and recovery reuse the same bucket-level locking rules. Any operation that updates DRAM bitmaps or entries follows the above locking discipline, while PMem records are never relocated.

IV. Discussion

Modern KV stores rely on a fast–slow memory hierarchy to achieve a balance between capacity and performance, keeping the primary KV table in slower memory and using fast memory to accelerate access via auxiliary structures. The form of this separation varies with the system's organization across existing designs. In client–server and disaggregated memory systems [49]–[52], KV data are stored in remote memory pools, and fast memory at the compute side is used to maintain hash tables or indexes optimized for low-latency access over the network. SRAM–DRAM architectures [7], [53], [54], commonly used in networking systems, employ a small amount of on-chip SRAM to cache hot entries or metadata, while placing the bulk KV data in external DRAM, making space efficiency in fast memory a primary constraint. In DRAM–persistent memory architectures, fast memory is typically used to store indexes or metadata that guide accesses to KV data in non-volatile memory; however, as the table scales, these indexes can grow substantially, leading to high fast memory consumption and reduced practicality.

MAPH is designed within this general fast–slow memory context and addresses a fundamental challenge shared by these architectures: achieving high space utilization in fast memory without introducing excessive operations on slow memory. To this end, MAPH leverages cuckoo hashing in fast memory to achieve a high load factor through kick-out-based collision resolution. While such collision handling is effective for improving space efficiency, its performance impact depends critically on where it is executed. If collision

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

resolution requires relocating entries or updating structures in slow memory, the resulting accesses can significantly degrade system performance.

The key insight of MAPH is to decouple fast-memory hash slots from the physical storage locations of KV pairs in slow memory. Instead of enforcing a strict one-to-one binding, MAPH maintains lightweight location metadata in fast memory and introduces redundancy in slow memory by allowing multiple candidate slots for KV placement. This design confines kick-out operations and other complex index manipulations entirely to fast memory, while slow memory is accessed only for KV retrieval or insertion. By doing so, MAPH preserves the high load factor benefits of cuckoo hashing and substantially reduces slow-memory accesses, resulting in both space-efficient indexing and high performance in tiered memory systems.

Importantly, the core design principles of MAPH are not tied to a specific hybrid memory implementation. More generally, they apply to systems that combine a fast memory tier for hash indexes and metadata with a slower memory tier for bulk KV storage. As long as fast memory can support efficient hash-based indexing and slow memory provides sufficient capacity for redundant KV placement, MAPH's approach remains applicable. In fact, as the access latency gap between fast and slow memory widens, reducing metadata and collision resolution operations on the slow tier becomes increasingly beneficial. From this perspective, MAPH illustrates a general design principle for fast–slow memory KV stores: high-complexity index management should be confined to fast memory, while slow memory should be accessed only for essential KV data operations through lightweight metadata indirection.

V. EVALUATION

Hardware Configuration: We conduct experiments on a server running Ubuntu 22.04.4 LTS with one Intel Xeon Gold 6240 processor (18 cores, 36 hyper-threads, 2.60GHz, 24.8MB L3 cache). The system provides 125GB DRAM configured with four 32GB DDR4-2666 DIMMs and 252GB Intel Optane DC PMem configured with two 126.422GB modules. All code is compiled using GCC 11.4.0 with all optimization enabled.

Workloads: We use the Yahoo! Cloud Serving Benchmark (YCSB) [55] to generate evaluation workloads. YCSB defines six standard workloads (A–F) that cover representative access patterns, including update-heavy (A, 50% read / 50% update), read-mostly (B, 95% read / 5% update), read-only (C, 100% read), read-latest (D, 95% read / 5% insert), scan-heavy (E), and read-modify-write (F). Each benchmark consists of a loading phase that pre-loads KV pairs into the system, followed by a running phase that issues operations on top of the pre-loaded dataset.

For the running phase, YCSB supports different request distributions over the key space. In this paper, we use: (i) `uniform`, where each key in the pre-loaded dataset is selected with equal probability; (ii) `zipfian`, where keys are selected according to a Zipf distribution (i.e., a small fraction of hot keys account for the majority of requests); and (iii) `latest`, where the selection probability is biased toward keys most recently inserted during the running phase.

Building on YCSB's standard workloads, we construct a more comprehensive workload suite for our evaluation(Table II). We categorize workloads into three classes according to their operation composition: single-operation workloads, insert–lookup mixed workloads, and update–lookup mixed workloads. Beyond directly instantiating YCSB-style mixed workloads, we further extend mixed workloads along two dimensions. First, we vary the operation mix ratio to cover different write intensities (e.g., 5/50/95% inserts or updates, with the remaining operations being lookups), thereby spanning read-dominant, balanced, and write-dominant scenarios. Second, we vary the request distribution to reflect different access skews: for update–lookup mixed workloads, we additionally use the `zipfian` distribution to model hot-key updates and reads; for insert–lookup mixed workloads, we additionally use the `latest` distribution to emphasize accesses biased toward newly inserted keys.

TABLE II
WORKLOADS USED IN OUR EVALUATION. SPECIFICALLY, YCSB-A CORRESPONDS TO U50L50, YCSB-B TO U5L95, YCSB-C TO L100, AND YCSB-D TO I5L95-L. SUFFIX "-Z" AND "-L" DENOTE ZIPFIAN AND LATEST DISTRIBUTIONS, RESPECTIVELY.

Workloads	Workload Configurations					
	a	**b**	**c**	**d**	**e**	**f**
Single-Operation	I100	L100	L100-Z	U100	U100-Z	D100
Insert-Lookup	I5L95	I5L95-L	I50L50	I50L50-L	I95L5	I95L5-L
Update-Lookup	U5L95	U5L95-Z	U50L50	U50L50-Z	U95L5	U95L5-Z

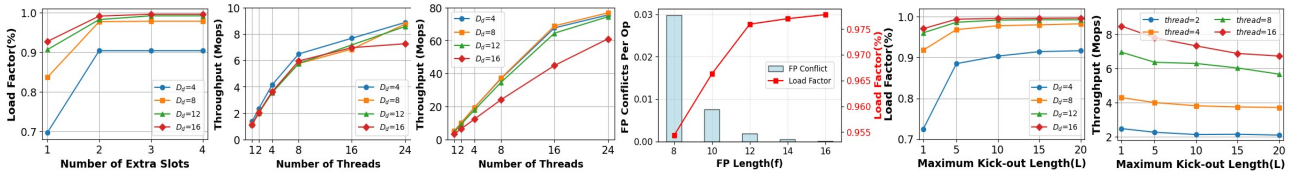
For insert-only workloads (I100), only the running phase inserts distinct keys. For all other workloads, the loading phase pre-populates the database with a fixed set of keys, and the running phase executes the corresponding operations. By default, we use 8-byte keys and 8-byte values.

A. Parameter Analysis and Ablation Studies

This section conducts a parameter sensitivity analysis and ablation study to quantify how key parameters and design choices affect both space efficiency (e.g., maximum load factor) and performance (throughput and tail latency). We run the experiments using single-operation workloads from Table II, including I100, L100, and L100 (zipfian), with a total of 30M KV pairs. We vary one parameter at a time (e.g., bucket size, fingerprint length, and maximum kick-out length) while keeping the remaining settings at their default values to isolate each factor's impact.

Impact of Bucket Size: Our algorithm's maximum load factor is determined by the number of slots in each DRAM bucket (D_d) and PMem bucket (D_p). The x-axis in Figure 4(a) represents the number of extra slots in D_p relative to D_d. Increasing extra slots from 1 to 2 significantly improves load factor, while further increases yield marginal gains as the system approaches its limit. Specifically, when $D_d = 4, 8, 12, 16$ and extra slots is 2, load factors are 90.3%, 97.6%, 98.2%, and

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.



(a) Impact of Bucket Size on Load Factor (b) Impact of Bucket Size on Insertion (c) Impact of Bucket Size on Lookup (d) Impact of Fingerprint Length on Insertion (e) Impact of Kick-out Length on Load Factor (f) Impact of Kick-out Length on Insertion

Fig. 4. Impact of parameters and design choices.

99.0% respectively. With 4 extra slots, load factors are 90.3%, 97.7%, 99.1%, and 99.5% respectively.

Figures 4(b)-(c) illustrate that as bucket size increases (extra slots = 2), average throughput of multi-threaded read and write operations decreases somewhat. Considering the trade-off between load factor and throughput, we conclude that the optimal value for D_d is 8 with 2 extra slots, ensuring high load factor while maintaining better throughput.

Impact of Fingerprint Length: Figure 4(d) shows the bar chart representing average fingerprint conflicts per insertion operation under different fingerprint lengths. Collision probability decreases as fingerprint length increases, improving overall system reliability. The line chart illustrates that even when fingerprint length is compressed to 8 bits, our algorithm maintains $> 95\%$ load factor.

Our experiments show that with $D_d = 8$ and 16 threads, insertion throughput achieves 8.27 Mops with 8-bit fingerprints and 8.44 Mops with 16-bit fingerprints. This demonstrates that even with shorter fingerprints, the system maintains high performance. We recommend 16-bit fingerprints when DRAM space is sufficient for better balance between collision probability and memory efficiency. For constrained DRAM scenarios, 8-bit fingerprints still allow high load factors and efficient operation.

Impact of Maximum Kick-out Length: The maximum kick-out length L significantly influences system load factor and insertion throughput. Figure 4(e) shows increasing L enhances load factor. However, once L reaches 10, further increases yield minimal improvements as the system's maximum load becomes primarily constrained by bucket size D_d.

Figure 4(f) demonstrates L's impact on insertion throughput under different thread counts. For $D_d = 8$, with 2 or 4 threads, throughput remains stable as L increases. However, with 8 or 16 threads, increasing L leads to noticeable throughput decline. This indicates that while larger L improves load factor, it may introduce overhead in multi-threaded environments. We recommend $L = 10$ for fewer working threads to maximize load factor, and reducing L in higher thread counts to maintain optimal throughput.

Impact of Extra Slots on Space-time Trade-offs: Table III further explores the space–time trade-off of provisioning extra PMem slots per bucket under an extremely skewed workload (Zipfian(0.99)).

Increasing the number of extra slots from 1 to 2 substantially improves the achievable load factor, at the cost of higher tail write latency when the table approaches full occupancy. The increase in insert p90 and p95 latencies is modest, rising from 1234 ns to 1279 ns and from 1348 ns to 1474 ns, respectively. In contrast, p99 insert latency increases more noticeably, from 1719 ns to approximately 2970 ns once two or more extra slots are provisioned. This increase is primarily caused by more frequent DRAM-side collision resolution, i.e., longer or more numerous kick-out chains, as the table nears its load-factor limit. Importantly, even this worst-case write latency remains far lower than the orders-of-magnitude overhead incurred when collision resolution must be performed directly in PMem without extra slots.

Read latency, by comparison, is largely insensitive to the number of extra slots, even under skewed Zipfian(0.99) workloads. The p99 read latency increases only slightly, from 970 ns to 1007 ns, while p90 and p95 latencies remain essentially unchanged. Overall, provisioning a small number of extra PMem slots provides a favorable space–time trade-off: it increases the achievable load factor while preserving stable read performance and confining worst-case write overhead to DRAM.

TABLE III
TAIL LATENCIES (NS) UNDER ZIPFIAN READ DISTRIBUTION WITH DIFFERENT NUMBERS OF EXTRA PMEM SLOTS PER BUCKET.

Extra Slots	Insert latency (ns)			Read latency (ns)		
	p90	p95	p99	p90	p95	p99
1	1233	1348	1719	744	818	970
2	1279	1474	2969	770	839	997
3	1281	1477	2971	770	840	1000
4	1286	1475	2976	776	845	1007

B. Comparison with State-of-the-Art

We compare MAPH against four representative hybrid DRAM–PMem KV stores, namely EEPH [22], MapEmbed [21], Halo, and Viper, which adopt different hashing schemes, collision resolution strategies, and memory allocation policies. This selection covers a broad spectrum of state-of-the-art designs for hybrid memory systems and provides a comprehensive baseline for evaluation.

For MapEmbed, which was originally designed for generic fast–slow memory hierarchies, we explicitly mapped its KV-table to persistent memory using libpmemobj [56] to ensure correct and fair utilization of PMem. For EEPH, Halo, and Viper, we directly adopted their original hybrid DRAM–PMem implementations. Due to the inherent differences in hash collision handling mechanisms and space allocation strategies

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

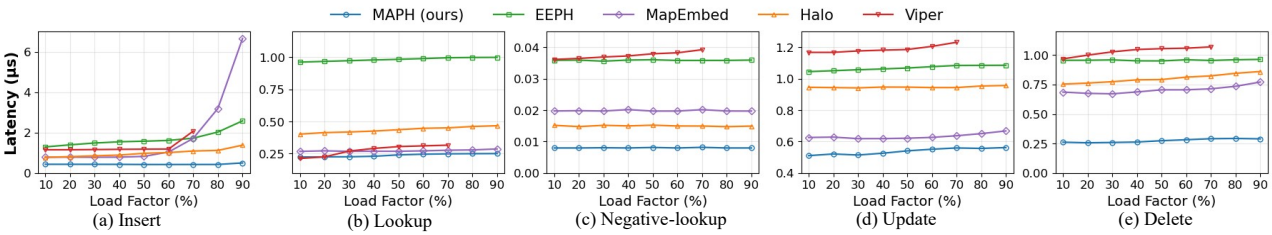


Fig. 5. Comparison of latency (LT) with prior solutions.

across these systems, we carefully controlled experimental settings to ensure fairness.

For latency evaluation under varying load factors, we pre-allocated a fixed capacity of 30 million KV pairs and disabled dynamic resizing, ensuring that both the loading and running phases issue 30 million operations without interference from expansion or reorganization. For throughput evaluation, we unified the initial DRAM budget to 1GB across all systems while enabling dynamic expansion, and we scaled the workload to 200 million operations to stress-test multi-threaded performance under realistic capacity growth. These contrasting setups isolate steady-state access costs in the former and fairly compare DRAM-efficiency under dynamic scaling in the latter.

Unless otherwise specified, MAPH uses the following default configuration: $D_d = 8,\ D_p = 10,\ f = 16,\ L = 10$. For all compared systems, we carefully tuned their parameters following best practices reported in the literature and conducted additional optimizations to achieve their best possible performance under our experimental environment.

Design Trade-offs in Hybrid Perfect Hashing: MAPH diverges from prior perfect hashing designs (EEPH/MapEmbed) by eliminating DRAM-resident metadata (guide arrays, offsets or depth indicators) that scale with table size. Instead, it confines collision resolution to DRAM-only cuckoo operations, trading modest PMem redundancy (1–2 slots/bucket) for minimal DRAM footprint. This asymmetric design shifts space cost from DRAM to PMem, acceptable given PMem's larger capacity and lower cost per byte. To concretely illustrate the implications of this design choice, we analyze the insertion behavior of perfect hashing–based indexing mechanisms. This experiment focuses on MAPH, EEPH, and MapEmbed, all of which explicitly enforce bounded load factors through slot-based perfect hashing designs.

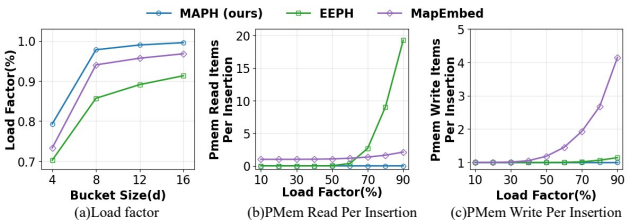


Fig. 6. Insertion-level analysis of hybrid perfect hashing designs.

Figure 6(a) shows that MAPH achieves a substantially higher achievable load factor, reaching up to 98.8% with $D_d = 8(D_p = 10)$, compared to 94.0% for MapEmbed and 85.7% for EEPH. At an identical bucket size, a higher achievable load factor implies that fewer buckets—and correspondingly less DRAM-resident indexing metadata—are required to store the same number of keys. This result demonstrates that MAPH can sustain denser key placement while maintaining insertion feasibility, directly translating into lower DRAM space consumption.

Figures 6(b)–(c) further reveal fundamental differences in PMem access behavior during insertions, reflecting where collision handling logic resides. MAPH completely eliminates PMem reads in collision-free insertions by leveraging DRAM-resident bucket bitmaps to identify valid slots, requiring only a single PMem write upon successful insertion. More importantly, even under high load, all collision resolution operations are confined to bounded kick-out procedures in DRAM, avoiding additional PMem accesses.

In contrast, EEPH incurs substantial PMem read overhead once the load factor exceeds 70%, as its complement move algorithm performs multi-path searches across PMem-resident entries to locate feasible insertion positions. MapEmbed exhibits significantly higher PMem write amplification, peaking at 4.2 writes per insertion, due to its circular movement mechanism that redistributes keys across virtual buckets during collisions.

Together, these results empirically confirm that MAPH's reduced DRAM footprint does not stem from weaker collision guarantees, but from a fundamentally different placement of collision handling logic. By avoiding DRAM-resident perfect hashing metadata and isolating collision resolution within DRAM, MAPH achieves higher space efficiency, stable PMem access costs, and the stable latency observed under high load, while significantly reducing DRAM consumption compared to prior systems.

Latency Performance: We evaluate operation latency under load factors ranging from 10% to 90% (Figure 5), covering all five basic operations. We use the single-operation workloads (a), (b), (c), and (f) listed in Table II. For negative lookup, we use the L100 workload with the queried keys modified to be absent from the KV store.

1) *Insertion latency (Figure 5(a)).* MAPH maintains low and stable insertion latency as the load factor increases. At 70% load factor, MAPH achieves an insertion latency of 0.42μs, compared to 1.72μs (EEPH), 1.71μs (MapEmbed), 1.08μs (Halo), and 2.07μs (Viper), corresponding to performance improvements of 2.57×–4.92×. As the load factor increases to 90%, MAPH's insertion latency rises only slightly to 0.49μs, while the other systems experience substantially higher

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

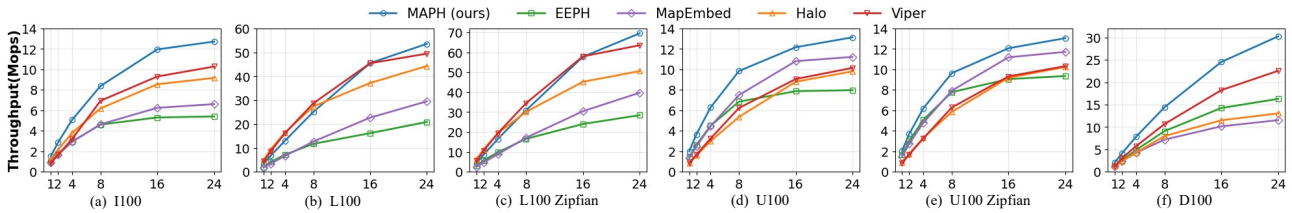


Fig. 7. Comparison of throughput under different threads of single-operation workloads.

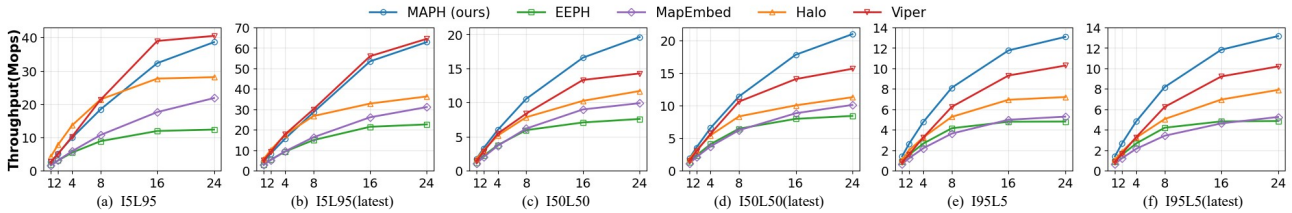


Fig. 8. Comparison of throughput under different threads of insert-lookup mixed workloads.

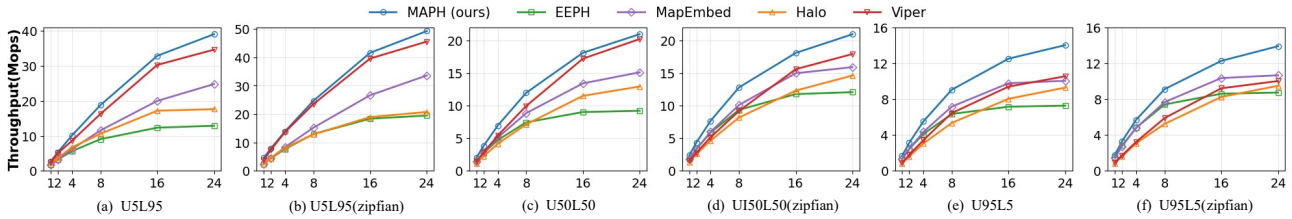


Fig. 9. Comparison of throughput under different threads of update-lookup mixed workloads.

latency, reaching 2.57μs (EEPH), 6.65μs (MapEmbed), and 1.38μs (Halo). Overall, MAPH outperforms other designs by 2.8×–13.6× in insertion latency under high load, highlighting the effectiveness of its DRAM-only collision handling in avoiding PMem accesses on the insertion critical path.

2) *Lookup, non-existent lookup, and update latency (Figure 5(b)–(d)).* For lookup, non-existent lookup, and update operations, latency remains largely stable for all systems across the evaluated load range, as these operations do not trigger structural modifications. MAPH consistently achieves the lowest latency among all algorithms. At 70% load factor, MAPH reduces latency by 1.1×–4.0× for lookup, 1.82×–4.8× for non-existent lookup, and 1.14×–2.2× for update, compared to the other systems. This advantage stems from MAPH's one-to-one DRAM-to-PMem mapping enabled by cuckoo hashing, which reduces DRAM-side computation overhead and requires significantly less DRAM-resident metadata compared to other algorithms.

3) *Deletion latency (Figure 5(e)).* Delete operations further highlight MAPH's efficiency. Deletion in MAPH only requires updating DRAM-resident validity bitmaps without modifying PMem data, resulting in consistently lower latency. At 70%load factors, MAPH achieves deletion latency of approximately 0.29μs, outperforming other systems by 2.45×–3.68×. In contrast, the other designs typically involve PMem writes or more complex metadata updates, leading to higher deletion overhead.

Throughput Analysis: We evaluate throughput under the workloads defined in Table II, covering single-operation, insert–lookup mixed, and update–lookup mixed scenarios.

Single-operation workloads.((Figure 7)) *Insert-only workloads.* MAPH consistently outperforms other designs, achieving throughput improvements ranging from 1.24× to 2.35×. This advantage arises from MAPH's streamlined insertion path, where collision resolution is handled entirely in DRAM, avoiding expensive PMem-side coordination, and each insertion requires only a single PMem write to atomically persist the key–value record and its validity bit. Moreover, MAPH incurs fewer expansion events during index construction, further reducing disruptive resizing overheads. In contrast, other designs involve additional PMem accesses, metadata updates, or coordination during insertion and resizing, which limits their scalability under multi-threaded execution.

Read-only workloads. Under 24 threads, MAPH achieves 1.08×–2.57× higher throughput under uniform access and 1.09×–2.44× under zipfian access. This improvement is primarily due to the query path design: both MAPH and Viper resolve lookups via DRAM-side computation followed by a single PMem read, whereas other schemes often require multiple PMem accesses to locate or validate records. As a result, MAPH reduces PMem read amplification and sustains higher read throughput, even under skewed access patterns.

Update-only workloads. For update workloads, MAPH achieves throughput gains of 1.17×–1.40× under uniform distributions and 1.11×–1.39× under zipfian distributions. MAPH and MapEmbed update only the key–value record

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

itself, while other designs must additionally modify PMem-resident metadata, introducing extra write traffic and persistence overhead. This difference becomes more pronounced as contention increases under skewed workloads.

Delete-only workloads. MAPH also shows clear advantages for delete operations, achieving 1.34×–2.62× higher throughput than the baselines. Deletion in MAPH only requires clearing the validity flag embedded in the PMem slot, whereas alternative approaches perform more complex metadata updates or bucket-level modifications. By minimizing PMem writes and confining deletion logic to a single slot, MAPH achieves higher throughput and better scalability.

Insert-lookup mixed workloads.(Figure 8) For read-dominant workloads (I5L95), Viper slightly outperforms MAPH. This is because Viper employs lighter-weight read-side concurrency control, whereas MAPH may experience bucket-level read–write contention when inserts and lookups target the same bucket concurrently. As a result, MAPH's throughput is marginally lower in scenarios where read operations dominate overall execution.

In the balanced I50L50 workload, MAPH achieves throughput improvements of 1.37×–2.58× under uniform distributions and 1.34×–2.49× under latest distributions. In the write-heavy I95L5 workload, the improvements persist, reaching 1.27×–2.71× and 1.29×–2.70× under uniform and latest distributions, respectively.

These gains arise from MAPH's more efficient write-side concurrency control and its ability to confine collision resolution to DRAM. As the insert ratio increases, other designs incur higher coordination and PMem access overheads, particularly under the latest distribution, where temporal locality intensifies contention on recently inserted keys. MAPH mitigates these effects and sustains higher throughput as write intensity grows.

Update-lookup mixed workloads.(Figure 9) Across all update–lookup configurations, MAPH consistently outperforms the baselines under both uniform and zipfian distributions. For U5L95, U50L50, and U95L5 workloads, MAPH achieves throughput improvements of 1.13×–3.03× and 1.08×–2.53×, 1.03×–2.27× and 1.17×–1.73×, and 1.32×–1.92× and 1.39×–1.60× under uniform and zipfian distributions, respectively.

The advantage stems from MAPH's lightweight update path, which modifies only the key–value record and avoids additional PMem metadata updates. This design reduces write traffic and persistence overhead, especially under higher update ratios and skewed access patterns.

C. Space Overhead Analysis

We evaluate the space overhead of MAPH and representative hybrid hashing index designs, with the results summarized in Table IV. Starting from an initial allocation of 1 GB DRAM, we insert 200 million key–value pairs with each design's native expansion mechanism enabled. Whenever insertions fail due to capacity limits, the index dynamically expands its DRAM and PMem space according to its original design. After all key–value pairs are successfully inserted, we measure the final DRAM and PMem space consumption of each approach.

MAPH exhibits substantially lower DRAM space consumption than the other hybrid hashing indexes. This advantage primarily stems from the high load factor supported by MAPH's Cuckoo hashing index, where expansion is triggered only when the load factor exceeds 95%. In contrast, other hash indexes typically initiate expansion at much lower load factors, ranging from 40% to 70%. Consequently, under the same initial DRAM allocation, MAPH requires fewer expansion events to accommodate the same number of key–value pairs, resulting in significantly lower overall DRAM overhead.

TABLE IV
SPACE CONSUMPTION FOR DRAM AND PMEM (IN GB)

	MAPH	MapEmbed	EEPH	Halo	Viper
DRAM	2.05	10.76	8.20	7.48	8.04
PMem	4.92	5.54	9.69	3.75	10.02

In addition, MAPH maintains more compact DRAM metadata. While other designs store auxiliary metadata to track DRAM–PMem mappings, MAPH enforces a one-to-one correspondence between its Cuckoo hashing index and PMem layout. Each KV pair in DRAM is therefore represented using only a 16-bit fingerprint and an 8-bit PMem location identifier, whereas other designs incur 1.4×–2.7× higher metadata overhead.

On the PMem side, Halo achieves the lowest space consumption by allocating contiguous space for complete key–value records with minimal recovery metadata. MAPH introduces moderate PMem overhead due to redundant slots in its PMem layout, but still consumes less PMem space than designs that maintain more complex PMem-resident indexing structures.

Overall, MAPH achieves a favorable balance between DRAM and PMem space efficiency, consistent with its design goal of minimizing memory overhead while preserving high performance.

TABLE V
INSERTION THROUGHPUT (MOPS) UNDER DIFFERENT DRAM:PMEM CAPACITY RATIOS

	1:10	1:100	1:1000
MAPH	9.26	3.20	0.32
Halo	6.22	1.72	0.17
Viper	6.81	1.83	0.19
EEPH	3.89	1.59	0.27
MapEmbed	5.11	1.65	0.29

To further examine system behavior under a large capacity gap between DRAM and PMem, we evaluate insertion performance when the two memory tiers differ by orders of magnitude in capacity. Specifically, we insert 10 million key–value pairs using 16 threads and adjust value sizes such that the resulting DRAM-to-PMem space consumption ratios are approximately 1:10, 1:100, and 1:1000. Table V reports the insertion throughput of different hash index designs under these configurations.

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

As the capacity gap between DRAM and PMem increases, insertion throughput degrades for all designs as PMem access cost becomes increasingly dominant. Nevertheless, MAPH consistently achieves the highest throughput across all three configurations. At a DRAM:PMem ratio of 1:10, MAPH outperforms other systems by 35%–50%, benefiting from its high DRAM load factor and DRAM-only collision resolution. As the gap widens to 1:100 and 1:1000, MAPH maintains this advantage over designs that incur additional PMem operations during insertion.

Designs with more compact PMem layouts, such as Halo, scale better as PMem capacity dominates, but still underperform MAPH due to lower DRAM utilization and more frequent DRAM expansions. Overall, these results demonstrate that MAPH remains effective even when PMem is provisioned at orders-of-magnitude larger scale than DRAM.

D. Dynamic Expansion and Recovery Performance

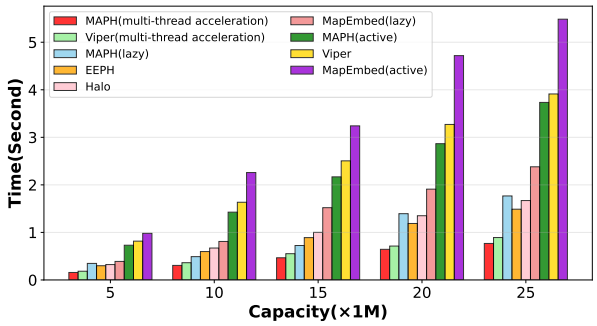


Fig. 10. Comparison of dynamic expansion time.

Dynamic Expansion: We evaluate the time cost of dynamic expansion when doubling table capacity under high utilization (Figure 10). All systems are evaluated at a 90% load factor, except Viper, whose load factor is capped at 70% due to design constraints.

MAPH and MapEmbed perform expansion through bucket copying followed by redundant checks. MAPH only reads PMem during redundant checks, whereas MapEmbed relocates key–value pairs within PMem, resulting in higher data movement overhead. Both designs support active-mode and lazy-mode expansion, depending on whether redundant checks are executed eagerly or deferred to post-expansion accesses.

EEPH adopts a passive lazy expansion strategy, allocating larger space upon insertion failure and amortizing expansion overhead into subsequent collision handling. In contrast, Halo and Viper allocate a new hash table and rehash all entries from the old table while updating their index structures. Halo performs asynchronous migration, and we measure the time to complete the entire process, while Viper enables multi-threaded acceleration for rehashing, which is also used in our evaluation.

Results show that lazy-mode expansion in MAPH achieves the lowest expansion latency among all evaluated designs. Overall, MAPH provides more efficient expansion behavior, offering practical advantages under capacity growth.

Recovery Performance: We evaluate the recovery time overhead of MAPH and compare it with EEPH, Halo, and Viper, all of which provide native recovery implementations (Table VI). We restart the service under different table capacities and measure the time required to reconstruct the DRAM hash index from PMem-resident data. All systems are evaluated under the same capacity, with a 90% load factor for MAPH, EEPH, and Halo, while Viper is evaluated at a 70% load factor due to its design constraints.

TABLE VI
RECOVERY TIME COMPARISON UNDER DIFFERENT CAPACITY

	5M	10M	15M	20M	25M
MAPH	1.04 s	2.18 s	3.23 s	4.90 s	5.41 s
EEPH	2.93 s	6.21 s	9.50 s	12.83 s	16.14 s
Viper	1.07 s	2.87 s	5.88 s	9.03 s	13.96 s
Halo	1.43 s	3.03 s	5.86 s	8.70 s	11.41 s

EEPH, Halo, and Viper incur higher recovery overhead due to the amount of PMem metadata accessed and the complexity of index reconstruction. EEPH processes both bucket contents and auxiliary metadata to restore its hash index, causing recovery cost to grow with metadata volume and table capacity. Halo relies on snapshot-based recovery, which requires scanning PMem metadata and copying hash tables and data pages back into DRAM, resulting in recovery time that scales with table size. Viper reconstructs its DRAM index by scanning PMem blocks and reinserting valid entries, making recovery overhead proportional to the number of stored entries.

In contrast, MAPH's cuckoo hashing-based design maintains a strict one-to-one correspondence between DRAM and PMem buckets with minimal metadata. This low-complexity index structure enables recovery by lightweight reconstruction with limited PMem metadata access. As a result, MAPH rebuilds its DRAM index with reduced memory traffic and consistently low recovery overhead across all evaluated capacities.

VI. CONCLUSION

MAPH is a memory-efficient hybrid DRAM–PMem KV store that decouples indexing from storage, confining all collision handling to DRAM and writing each KV pair to PMem only once—achieving high load factors and low insertion latency while minimizing DRAM metadata overhead. The design is guided by two core principles: Mirror (shared bucket indices across DRAM and PMem) and Asymmetry (independent placement decisions per tier). Experiments across diverse workloads confirm that MAPH delivers lower latency, higher throughput, and a smaller DRAM footprint than representative hybrid baselines, while maintaining compact PMem overhead. More broadly, MAPH demonstrates that a refined fast-memory index with simplified cross-tier mapping—combined with slow-memory operations restricted to final writes—yields scalability and crash resilience, offering a practical blueprint for future hybrid DRAM–PMem KV stores that jointly optimize load factor, performance, and memory efficiency.

Acknowledgment. This work was supported by National Natural Science Foundation of China (No.62472418).

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

AI-Generated Content Acknowledgement

We used GPT-5 (OpenAI) to assist in preparing certain portions of this article. Specifically, GPT-5 was used to polish and refine the text in the Introduction and Conclusion sections for improved clarity and conciseness; to condense the description of the multi-threading support subsection within the Design and Implementation section; and to assist in generating Matplotlib code used to visualize the experimental results.

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[2] Redis, "Redis, an in-memory data structure store," https://redis.io, 2020.

[3] Y. Wang, L. Yuan, W. Zhang, X. Li, Z. Chen, and Q. Liu, "Simpler is more: Efficient top-k nearest neighbors search on large road networks," *Proc. VLDB Endow.*, vol. 17, no. 13, pp. 4683–4695, 2024.

[4] Y. Wang, L. Yuan, Z. Chen, W. Zhang, X. Lin, and Q. Liu, "Towards efficient shortest path counting on billion-scale graphs," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2579–2592.

[5] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 215–226.

[6] Z. Liu, X. Niu, W. Zhou, Y. Hong, Z. Shi, T. Yang, Y. Zhang, Y. Wu, Y. Zhao, Z. Fan *et al.*, "Extendible rdma-based remote memory kv store with dynamic perfect hashing index," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2025, pp. 1745–1758.

[7] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 90–106.

[8] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent memory i/o primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–7.

[9] P. Götze, A. K. Tharanatha, and K.-U. Sattler, "Data structure primitives on persistent memory: an evaluation," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020, pp. 1–3.

[10] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 169–182.

[11] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[12] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "{Write-Optimized} dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 31–44.

[13] A. Van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing non-volatile memory in database systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1541–1555.

[14] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf Der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 738–761, 1994.

[15] F. C. Botelho, D. M. Gomes, and N. Ziviani, "A new algorithm for constructing minimal perfect hash functions," *differences*, vol. 100, no. 2, p. 09, 2004.

[16] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," in *Workshop on Algorithms and Data Structures*. Springer, 2007, pp. 139–150.

[17] G. E. Pibiri and R. Trani, "Pthash: Revisiting fch minimal perfect hashing," in *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, 2021, pp. 1339–1348.

[18] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid pmem-dram key-value store," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[19] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[20] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[21] Y. Wu, Z. Liu, X. Yu, J. Gui, H. Gan, Y. Han, T. Li, O. Rottenstreich, and T. Yang, "Mapembed: Perfect hashing with high load factor and fast update," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 1863–1872.

[22] Q. Chen, H. Hu, C. Deng, D. Liu, S. Li, B. Tang, T. Yao, and W. Xia, "Eeph: An efficient extendible perfect hashing for hybrid pmem-dram," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1366–1378.

[23] P. Zuo, Y. Hua, and J. Wu, "Level hashing: A high-performance and flexible-resizing persistent hashing index structure," *ACM Transactions on Storage (TOS)*, vol. 15, no. 2, pp. 1–30, 2019.

[24] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 799–812.

[25] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *arXiv preprint arXiv:2003.07302*, 2020.

[26] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, "Halo: A hybrid pmem-dram persistent hash index with fast recovery," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1049–1063.

[27] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "{SLM-DB}:{Single-Level}{Key-Value} store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 191–205.

[28] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 677–694, 2016.

[29] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic, "Aerospike: Architecture of a real-time operational dbms," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1389–1400, 2016.

[30] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[31] "Source codes of maph," https://github.com/MAPH2025/MAPH2025, gitHub repository.

[32] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[33] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong *et al.*, "Spin-transfer torque magnetic random access memory (stt-mram)," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, pp. 1–35, 2013.

[34] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 9, no. 2, pp. 1–20, 2013.

[35] R. Crooke and M. Durcan, "A revolutionary breakthrough in memory technology," *Intel 3D XPoint launch keynote*, 2015.

[36] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for olap workloads," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 339–351.

[37] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[38] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: system programming guide, Part*, vol. 2, no. 11, pp. 0–40, 2011.

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.

[39] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, "{SmartCuckoo}: a fast and {Cost-Efficient} hashing index scheme for cloud storage systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 553–565.

[40] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, and S. Cao, "Mincounter: An efficient cuckoo hashing scheme for cloud storage systems," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–7.

[41] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *International Workshop on Approximation Algorithms for Combinatorial Optimization*. Springer, 2009, pp. 490–503.

[42] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[43] H.-P. Lehmann, P. Sanders, and S. Walzer, "Sichash-small irregular cuckoo tables for perfect hashing," in *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2023, pp. 176–189.

[44] B. Fan, D. G. Andersen, and M. Kaminsky, "{MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.

[45] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 97–108.

[46] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[47] M. Aumüller, M. Dietzfelbinger, and P. Woelfel, "Explicit and efficient hash families suffice for cuckoo hashing with a stash," *Algorithmica*, vol. 70, no. 3, pp. 428–456, 2014.

[48] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2010.

[49] P. Zuo, Q. Zhou, J. Sun, L. Yang, S. Zhang, Y. Hua, J. Cheng, R. He, and H. Yan, "Race: one-sided rdma-conscious extendible hashing," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–29, 2022.

[50] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 249–264.

[51] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "{ROLEX}: A scalable {RDMA-oriented} learned {Key-Value} store for disaggregated memory systems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 99–114.

[52] T. Liu, T. Peng, P. Yang, X. Zhao, X. Lu, W. Huang, Z. Liu, X. Chen, Z. Liang, J. Xiong, D. Jin, M. Zhang, J. Guo, Y. Deng, X. Zhang, X. Dong, S. Wang, S. Wu, Y. Wu, Z. Tang, Y. Zeng, Y. Wang, J. Liu, M. Kang, M. Li, Y. Wang, Y. Liu, X. Ma, Y. Wang, Y. Zhang, J. Yin, K. Zheng, J. Yin, J. Zhang, Z. Wang, X. Lin, L. Liu, L. Lan, Y. Liu, C. Peng, H. Liu, S. Ren, X. Wang, Y. Shen, Y. Wang, G. Liu, H. Chen, T. Yang, H. Yang, J. Li, G. Ding, and K. Zhang, "xllm technical report," *arXiv preprint 2510.14686*, 2025.

[53] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A high-speed stateful packet processing approach for tbps programmable switches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1237–1255.

[54] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 523–535.

[55] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[56] Intel, "Persistent memory development kit (libpmemobj)," https://pmem.io/pmdk/libpmemobj/, 2018.

Authorized licensed use limited to: Peking University. Downloaded on November 07,2026 at 03:58:39 UTC from IEEE Xplore. Restrictions apply.