Nap: Persistent Memory Indexes for NUMA Architectures

QING WANG, YOUYOU LU, JUNRU LI, MINHUI XIE, and JIWU SHU, Tsinghua University

2

We present Nap, a black-box approach that converts concurrent persistent memory (PM) indexes into non-uniform memory access (NUMA)-aware counterparts. Based on the observation that real-world workloads always feature skewed access patterns, Nap introduces a NUMA-aware layer (NAL) on the top of existing concurrent PM indexes, and steers accesses to hot items to this layer. The NAL maintains (1) *per-node partial views* in PM for serving insert/update/delete operations with failure atomicity and (2) *a global view* in DRAM for serving lookup operations. The NAL eliminates remote PM accesses to hot items without inducing extra local PM accesses. Moreover, to handle dynamic workloads, Nap adopts a fast NAL switch mechanism. We convert five state-of-the-art PM indexes using Nap. Evaluation on a four-node machine with Optane DC Persistent Memory shows that Nap can improve the throughput by up to 2.3× and 1.56× under write-intensive and read-intensive workloads, respectively.

CCS Concepts: • **Information systems** → **Data structures**; • **Hardware** → *Non-volatile memory;*

Additional Key Words and Phrases: Persistent memory, non-uniform memory access, indexes

ACM Reference format:
Qing Wang, Youyou Lu, Junru Li, Minhui Xie, and Jiwu Shu. 2022. Nap: Persistent Memory Indexes for NUMA Architectures. *ACM Trans. Storage* 18, 1, Article 2 (January 2022), 35 pages.
https://doi.org/10.1145/3507922

1 INTRODUCTION

We consider the problem of making **persistent memory** (**PM**) indexes NUMA-aware. Although there has been a wealth of prior research designing high-performance PM indexes [19, 20, 23, 24, 43, 54, 55, 58, 61, 65–67, 74, 80, 83, 84], the impacts of **non-uniform memory access** (**NUMA**) architecture to PM indexes have not been deeply explored. Due to limited DIMM slots and cores in a single CPU, NUMA architecture is a necessity for providing massive bandwidth and capacity of PM along with enormous computational power. In a NUMA machine, the CPU cores and DRAM/PM DIMMs are grouped into nodes, which connect each other via inter-node links, e.g., Intel **Ultra Path Interconnect** (**UPI**).

The NUMA problem on PM indexes is unique. First, PM suffers from more severe impacts of NUMA than DRAM. Specifically, for Intel Optane DC Persistent Memory (i.e., Optane DIMM), the

This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300), and Huawei (Grant No. YBN2019125112).
Authors' address: Q. Wang, Y. Lu, J. Li, M. Xie, and J. Shu (corresponding author), Department of Computer Science and Technology, Tsinghua University, 30 Shuangqing Road, 201 East Main Building, Beijing 100084, China; emails: q-wang18@mails.tsinghua.edu.cn, luyouyou@tsinghua.edu.cn, {lijr19, xmh19}@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
1553-3077/2022/01-ART2 $15.00
https://doi.org/10.1145/3507922

first PM product, compared with local PM write, the peak bandwidth of remote ones is decreased to 59%; worse, highly concurrent remote PM writes (i.e., more than eight threads) experience a bandwidth cliff (Section 2.2). Second, to guarantee failure atomicity (i.e., the system can recover to a correct state upon system crashes), a PM index should issue flush instructions for explicitly evicting data from CPU caches to PM. For data that resides on remote nodes, these flush instructions expose remote PM writes on the critical path, degrading the performance. Third, PM has limited bandwidth (1/6 and 1/3 of DRAM in terms of writes and reads, respectively [79]), making replication-based approaches impractical. Existing NUMA-aware DRAM indexes always (partially) replicate indexes across NUMA nodes and synchronize these replicas via compact operation logs [15, 63]. Replication effectively reduces remote accesses; yet, since every update operation is executed at every node, *the number of local accesses is amplified significantly*. Although this amplification is not a problem for DRAM due to its extremely high local bandwidth, it is fatal for PM with low local bandwidth.

In this article, we propose **NUMA-Aware Persistent Memory Indexes** (**Nap**), a black-box approach that converts concurrent PM indexes into NUMA-aware counterparts. Nap is based on a common observation: Real-world workloads always feature skewed access patterns [13, 16, 18, 42, 81], where a small portion of hot items receive extremely frequent accesses. The key idea of Nap is *making hot accesses NUMA-aware*. Nap introduces a general **NUMA-aware layer** (**NAL**), which can be placed on the top of any existing concurrent PM index. The NAL absorbs accesses to *hot items*, while the underlying PM index handles accesses to other items. Specifically, NAL maintains per-node **partial and crash-consistent views** (**PC-views**) in PM, which serve insert/update/delete operations from local threads with failure atomicity. NAL does not synchronize states between PC-views, to *avoid remote PM accesses without inducing extra local PM accesses*. Such a synchronization-less approach brings two challenges: (1) serving lookup operations to hot items; (2) identifying the latest values from multiple PC-views upon recovery. For (1), NAL maintains an additional global view of hot items in DRAM. For (2), NAL adopts a version-based mechanism to order insert/update/delete operations to the same items, along with low-overhead methods of failure atomicity.

Upon workloads change, Nap can identify the new set of hot items and then switch to a new NAL quickly. The hot set identification is achieved by a combination of accurate and efficient streaming algorithms (e.g., count-min sketch [26]). To mitigate blocking of foreground index operations during NAL switch, Nap introduces a *three-phase switch*. This mechanism detects the states of access threads via a lightweight grace-period-based method. By leveraging these states, Nap divides the switch into three phases, and carefully splits tasks (e.g., initializing new NAL, flushing, and recycling old NAL) into different phases. As a result, only a small portion of index operations during a small interval are blocked.

Nap approach offers several advantages. First, it is general and efficient; we convert five state-of-the-art concurrent PM indexes using Nap, and the Nap-converted counterparts boost the throughput significantly on a four-node machine. Second, since the set of hot items is always small, the extra memory consumption and recovery time induced by Nap is bounded. Our evaluation on a four-node machine running 72 threads shows that, when maintaining 100 K hot items in the NAL, Nap uses less than 70 MB extra DRAM/PM space, and the recovery time is less than 1 second.

Nap has some limitations. First, it targets skewed workloads but not uniform workloads, which appear relatively rarely in the real world. Second, Nap-converted PM indexes may be outperformed by a crafted NUMA-aware PM index. However, when designing and evaluating Nap, we conclude some guidelines that may benefit future specialized NUMA-aware PM indexes, among which the most remarkable is that *a NUMA-aware PM index should reduce remote PM accesses without consuming extra local PM bandwidth*.

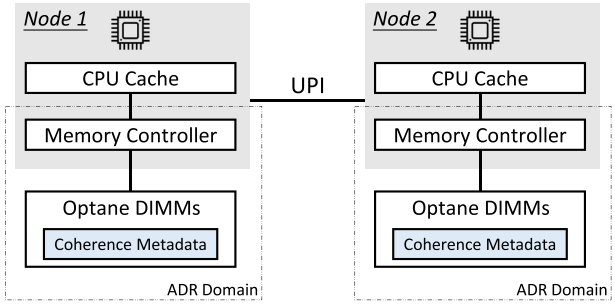

Fig. 1. Architecture of a two-node NUMA server equipped with Optane DIMMs.

In summary, this article makes the following contributions:

— Nap, a black-box and practical approach that converts concurrent PM indexes into NUMA-aware counterparts.
— A set of techniques that enables Nap's fast reaction to workloads change.
— Experimental evidence showing the efficiency of Nap.

2 BACKGROUND AND MOTIVATION

In this section, we firstly introduce PM architecture (Section 2.1). Then, we show that access to remote PM suffers from low performance (Section 2.2), and how it cripples PM indexes (Section 2.3). Finally, we analyze why existing approaches for DRAM indexes are inefficient when applied to PM (Section 2.4).

2.1 PM Architecture

PM is a new memory technology that enjoys the benefits of both storage and memory: it provides byte-addressable storage with DRAM-comparable performance and high density. With the release of Optane DIMMs, the first PM product, the system community is actively redesigning storage systems to gain full exploitation of its potential [12, 21, 22, 34, 39, 40, 45, 52, 55, 61, 64, 71, 82]. Figure 1 presents the architecture of a two-node NUMA server equipped with Optane DIMMs. Optane DIMMs are installed on DIMM slots and thus can handle requests from the memory controller. Different NUMA nodes are connected via Intel UPI. In the current configuration, a PM DIMM must operate side-by-side with a DRAM DIMM [38], so every PM server contains DRAM resources, forming a PM/DRAM hybrid system.

Data persistence. The memory controller is protected by the ***asynchronous DRAM refresh*** (**ADR**) domain, which ensures store instructions reaching the memory controller can survive power failures. Since the CPU cache is volatile, to enforce persistence, CPUs have to adopt one of the following two methods: ❶ issuing flush instructions, including `clflush`, `clflushopt`, and `clwb` for explicitly writing back data from the cache to the ADR domain, or ❷ using **non-temporal stores** (`ntstore`) to bypass the cache and write directly to the ADR domain. Above instructions except `clflush` are non-blocking, so we always need to issue fence instructions (i.e., `sfence`) to wait for the completeness of persistence.

2.2 NUMA Impacts on PM

A NUMA machine with numerous CPU cores and Optane DIMMs should be an ideal architecture for fast and large-volume storage; however, this is not true, due to slow remote PM accesses (i.e., accessing PM on remote NUMA nodes).

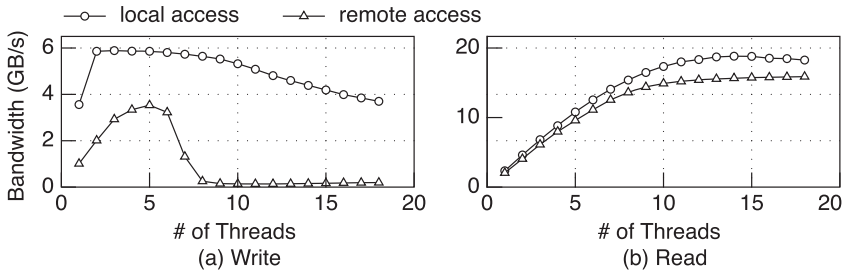

Fig. 2. Bandwidth of three 128 GB Optane DIMMs with varying threads. ***local access:*** *threads access Optane DIMMs that are local to them;* ***remote access:*** *threads access Optane DIMMs installed on another NUMA node. We use* `ntstore` *instructions for PM write. UPI uses directory-based cache coherence protocol.*

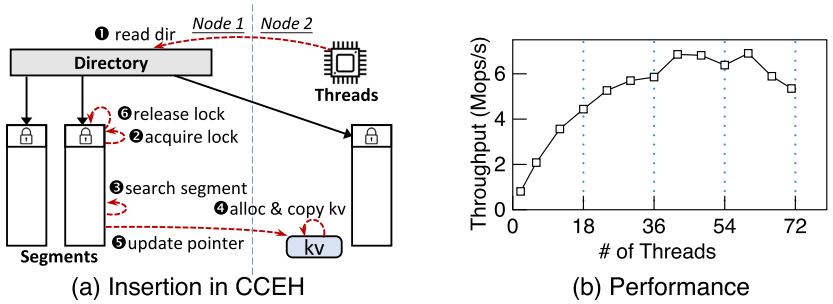

Fig. 3. NUMA impacts on PM indexes, using CCEH as an example. *We use source code from [5], which relies on PMDK [4] for PM allocation and supports variable-length keys.* ***(a)*** *An insert operation. Access threads reside on node 2, while the directory and the targeted segment are on node 1. This insertion needs 2 remote reads (❶❸) and 3 remote writes (❷❺❻).* ***(b)*** *Throughput of CCEH. Each thread allocates PM space from its local node. Vertical lines show the boundaries between NUMA nodes.*

Figure 2 reports the local/remote bandwidth of Optane DIMMs (3 Optane DIMMs and 18 CPU cores per NUMA node). Each thread performs sequential access to a 2 GB PM space. We use 32-byte `ntstore` for PM write. The peak writes bandwidth of remote accesses (3.5 GB/s) is only 59% of that of local accesses (5.9 GB/s). Worse, the bandwidth of remote write collapses (<250 MB/s) in case of more than eight concurrent threads. For read operations, though Optane DIMMs have a relatively smaller gap (16.9%) between local bandwidth and remote bandwidth, the extra access latency induced by inter-node links, i.e., UPI, is considerable (~100 ns), exacerbating the already high PM read latency (~300 ns, [79]). Based on these observations, we conclude that a high-performance PM system should avoid accessing remote PM, especially for writes.

Our experimental result is consistent with recent studies [22, 27, 50, 69, 76, 79]. We attribute the low performance of remote PM writes to two reasons. First, `ntstore` instructions may behave like cache line read-modify-write instructions, reducing the available PM bandwidth [27]. Second, due to the read-modify-write behavior, remote writes may trigger multi-socket cache coherence events, which induces extra PM writes [8]. Now, we explain why multi-socket cache coherence generates PM writes. Intel UPI uses a directory-based protocol to guarantee cache coherence between NUMA nodes [10]. The directory protocol records coherence metadata (e.g., cache lines' distribution) for cache invalidation. The coherence metadata is stored in Optane DIMMs, as shown in Figure 1. Thus, when triggering a cache coherence event, the directory protocol modifies the coherence metadata, which generates PM writes and further consumes limited PM write bandwidth.

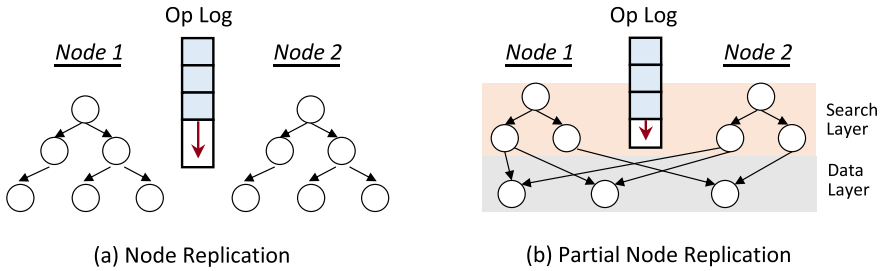

Fig. 4. Two existing DRAM-oriented approaches that support NUMA-aware indexes.

2.3 NUMA Impacts on PM Indexes

By leveraging the persistence and byte-addressability of PM, PM indexes can recover instantly in the presence of power outages. Although there has been an influx of PM indexes designed for Optane DIMMs, most of them are evaluated in a single NUMA node environment [23, 55, 58, 60, 64, 83]. Here, we investigate the NUMA impacts on PM indexes by analyzing CCEH [65], a variant of extendible hashing optimized for PM. CCEH manages a set of segments, which are pointed by a global directory. As shown in Figure 3(a), when performing an insertion, a thread may trigger multiple times (up to 2 remote reads and 3 remote writes) of remote PM accesses. Such remote accesses can significantly degrade the performance of PM indexes. We measure the performance of CCEH under the multi-node environment with a synthetic workload, where the ratio of lookup to insert/update is 1:1 and keys follow the Zipfian distribution with parameter 0.99. We use 15-byte keys and 8-byte values. Our platform is comprised of four Intel Xeon Gold 6240 M CPUs (18 cores per CPU), each with three 128 GB Optane DIMMs (1.5 TB in total). More details of hardware configurations are shown in Section 6. Figure 3(b) shows the result. CCEH scales well within a single NUMA node. However, the growth rate of throughput slows down significantly when the thread number increases from 18 to 36; the main cause is remote PM accesses. When more NUMA nodes are added, i.e., thread number increases from 36 to 72, the throughput fluctuates: it increases first and then decreases. This is because that a newly added NUMA node brings extra PM bandwidth resources, boosting the throughput, but soon, slow PM remote accesses become the key performance determinant, degrading the throughput.

2.4 Limitations of DRAM-oriented Approaches

A natural question now arises: are existing NUMA-aware approaches for DRAM indexes still efficient when applied to PM? We give a negative answer to this question by examining two existing approaches to NUMA-aware DRAM indexes.

Node Replication (NR). NR [15] is a state-of-the-art black-box approach that obtains NUMA-aware DRAM indexes from single-thread indexes. As shown in Figure 4(a), NR maintains a global operation log and per-node replicas of DRAM indexes. Using flat combining [41] within nodes, threads record their operations into the log, and execute the log entries to make their local replicas consistent between nodes. Three main limitations leave NR ill-suited for PM indexes.

First, obviously, NR does not consider failure atomicity, which is indispensable for PM indexes. Second, NR experiences severe space overhead: For a machine with n NUMA nodes, NR consumes n times more PM due to replication. As important storage system components, PM indexes always occupy a large portion of PM space; hence, such consumption is unacceptable. Third, the performance of insert/update/delete operations is limited by PM write bandwidth of a single NUMA node. To maintain consistent replicas between nodes, each node must execute the same series

Table 1. A Comparison of Different Approaches to NUMA-aware PM Indexes

	NR [15]	PNR [28, 63]	NAP
Black-Box	✓	✗	✓
Failure Atomicity	✗	✗	✓
Extra Local Accesses	High	High	None
Memory Consumption	High	Medium	Low

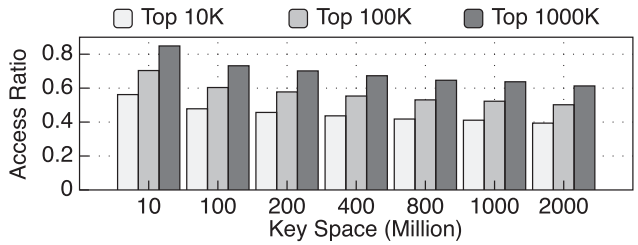

Fig. 5. Access ratio of hot items (Zipfian 0.99).

of operations, which wastes precious local PM write bandwidth (only 1/6 of DRAM) and further bottlenecks the overall throughput.

Partial Node Replication (PNR). To mitigate memory consumption of NR, several works [28, 63] adopt an approach we call **Partial Node Replication** (**PNR**). As shown in Figure 4(b), PNR decomposes an index structure into two-layer, namely, data layer and search layer. The data layer stores data items; the search layer is responsible for locating items in the data layer, so the search layer always stores a subset of keys. PNR replicates the search layer across NUMA nodes using operation logs like NR, while different NUMA nodes access the data layer in a shared manner. Compared with NR, PNR trades remote accesses for memory consumption: PNR does not replicate the data layer at the cost of accessing items residing in remote nodes. PNR also suffers from write amplification of local accesses due to replication. Moreover, PNR is not a general (black-box) approach since not all indexes can be divided into a data layer and a search layer (e.g., B Tree).

Different from the above two replication-based approaches, only using a small additional PM and DRAM space, NAP reduces remote PM accesses for any concurrent PM indexes, while not squandering extra local PM bandwidth. Table 1 shows a comparison of these approaches.

3 KEY IDEAS

(1) Making hot accesses NUMA-aware. Real-world workloads often feature Zipfian popularity distribution [13, 16, 18, 42, 81], where a small portion of hot items receive extremely frequent accesses. A recent study from Twitter [81] shows that their in-memory cache workloads are usually even more skewed than YCSB [25]. We design NAP to target these skewed workloads by making accesses to hot items NUMA-aware. To show the potential benefits of such a design, we run a simulation to present the access ratio of hot items. The key popularity follows the Zipfian distribution with parameter 0.99. From Figure 5, we observe that under a wide range of keyspace (from 10 M to 2000 M), the top 10 K/100 K/1000 K hottest items receive more than 39%/50%/61% accesses. Hence, if we can absorb accesses to hot items (e.g., top 100 K) in a NUMA-aware way, a significant percentage of remote PM accesses are avoided.

NAP introduces a NAL to absorb accesses to these hot items. In addition to reducing remote PM accesses, the NAL features two advantages. First, since the set of hot items is always small

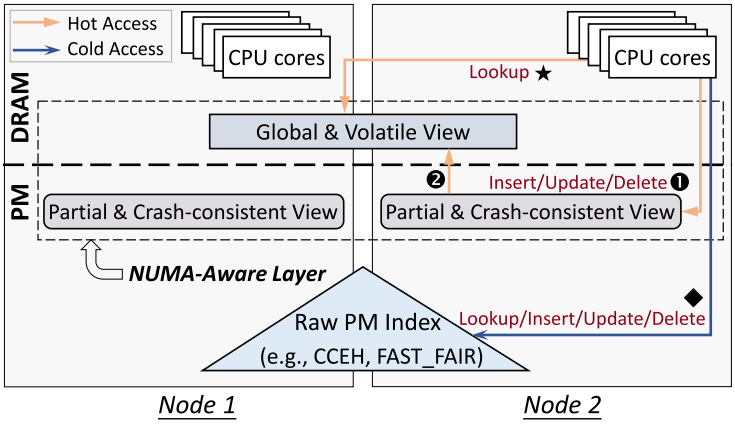

Fig. 6. Nap's architecture and interactions.

(e.g., 100 K), different from replication-based approaches (e.g., NR [15]), the DRAM/PM space used by the NAL is limited. Second, upon system crashes, the small-sized NAL can be recovered fast, bounding the recovery time.

(2) Black-box approach. Nap exploits hotness of items to handle the NUMA problem, which enables a black-box approach for converting existing concurrent PM indexes into NUMA-aware ones. Specifically, in Nap, the NAL absorbs accesses to hot items, and an underlying PM index accommodates a large number of cold items. Nap requires no inner knowledge of the underlying PM index. Any existing PM index that is crash-consistent and thread-safe can be used; thus, Nap takes advantage of the mature, well-tested codes of PM indexes, which are usually implemented via myriad engineering efforts.

(3) Minimizing state synchronization between PM nodes. The NAL records updates to hot items into the local PM and does not synchronize PM-resident states between different NUMA nodes; thus, in addition to reducing consumption of remote PM bandwidth, no extra local PM bandwidth is consumed in Nap. To enable efficient lookup operations in such a synchronization-less approach, the NAL maintains the latest values of hot items in the DRAM.

(4) Fast reaction to handle hotspot shift. Hotspots change over time, so Nap adopts several techniques to enable fast reaction. Specifically, Nap maintains the current hot items in real time. Upon detecting a new set of hot items, Nap generates a new NAL and installs it into the system in an atomic manner.

4 DESIGN

4.1 Overview

This article proposes Nap, an approach that converts concurrent PM indexes into NUMA-aware ones. Figure 6 presents the architecture and interactions of Nap. Nap consists of two main components: a raw PM index and a NUMA-aware layer.

— *Raw PM index.* The raw PM index can be an arbitrary existing concurrent PM index (e.g., CCEH [65], FAST_FAIR [43]), regardless of its concurrency control mechanism (lock-based or lock-free) and structure (tree-based, hashtable-based or hybrid). The raw PM index spans multiple NUMA nodes; it manages cold items (◆ in Figure 6), which account for an extremely huge proportion of the total dataset.

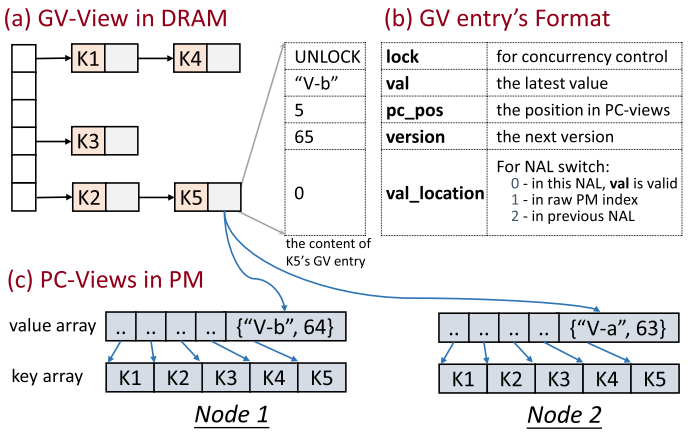

Fig. 7. Structures of the GV-view and PC-views.

— *NUMA-aware layer (NAL).* Nap steers accesses to hot items to the NAL, which contains two parts: a *global* and *volatile view* (i.e., GV-view, Section 4.2) and per-node *partial* and *crash-consistent views* (i.e., PC-views, Section 4.3). GV-view resides in DRAM, and maintains the latest values of hot items to serve lookup requests (★ in Figure 6). Per-node PC-views reside in PM. When a thread issues an insert/update/delete operation to a hot item, the PC-view in the same NUMA node absorbs the operation, and persists the operation's effect in a crash-consistent manner (❶). Then, the corresponding value in GV-view is updated (❷), to ensure the GV-view always owns the latest values of hot items. To eliminate remote PM accesses and avoid extra local PM accesses, we do not synchronize states between different PC-views, and thus each PC-view only has *partial* latest values of hot items. In case of hotspot shift, Nap can timely identify the new set of hot items (Section 4.4) and switch to a new version of NAL (Section 4.5); meanwhile, hot items in the old NAL are flushed to the underlying raw PM index.

4.2 Global & Volatile View (GV-View)

Design goals. In addition to serving lookup for hot items, the DRAM-resident GV-view is also responsible for (1) controlling concurrent accesses to the NAL, and (2) checking an item whether belongs to the hot set.[1] Thus, the design of GV-view must be lightweight and efficient.

Design details. Nap organizes the GV-view as a DRAM-resident index, which maintains the mapping from key to *GV entry* for every hot item. Figure 7(a) shows the GV-view's structure. The GV-view uses a hashtable by default; but if the raw PM index supports range query, it uses a tree-based data structure. Since the hot set is fixed unless the NAL is switched (e.g., the hot set is {*K1, K2, K3, K4,* and *K5*} in Figure 7), the GV-view's index is constructed *entirely* during the NAL's initialization and thereafter does not make any changes to its structure. As a result, any *thread-unsafe* index with high performance is applicable (e.g., C++ `unordered_map`).

For each hot item, the associated GV entry maintains its runtime information. Figure 7(b) shows the GV entry's format, which consists of five fields: (1) a readers-writer lock to control concurrent accesses to the hot item; (2) the latest value of the item; (3) a pointer indicating where to persist the item in PC-views. (4) the version of this item, which is used for recoverability of PC-views (Section 4.3); (5) an enumerated value that assists in NAL switch (Section 4.5).

[1]To simplify exposition, we term the set of hot items as *hot set*. Here, we assume the content of hot set is known in advance (Obtaining the hot set is detailed in Section 4.4).

Lookup operation. In case of no NAL switch, a lookup operation is performed as the following: the access thread checks the GV-view for the targeted item; if the targeted item does not exist, the lookup is redirected to the raw PM index; otherwise, the thread acquires read lock in corresponding GV entry, copies the value, and finally releases the lock.

Range query operation. Nap complicates the range query, because items for a targeted range may exist in the GV-view and raw PM index simultaneously. An access thread performs a range query as the following: it searches the GV-view, getting the items in the targeted range (S_1); then, it obtains the S_2 by invoking the range query interface of the raw PM index; finally, it merges S_1 and S_2 (if an item exists in both S_1 and S_2, we leave the one in S_1), returning the result. Like FAST_FAIR [43] and P-Masstree [55], the range query operations in Nap are not atomic with a concurrent insert/update/delete operations; if a system (e.g., database) atop Nap requires a higher isolation level (e.g., *repeatable read*), it needs to implement next-key locking or version mechanisms [73].

4.3 Partial and Crash-consistent View (PC-View)

Design goals. The per-node PM-resident PC-views absorb update/insert/delete operations and ensure the effects of these operations can survive power outages. PC-views have two design goals: (1) *Recoverability.* The states between PC-views are inconsistent, and thus Nap must be able to identify the latest values upon recovery. (2) *Low-overhead failure atomicity.* To guarantee failure atomicity, we must explicitly persist data with flush instructions (e.g., `clflush`, `clwb`, and `clflushopt`) and avoid store reordering with fence instructions (e.g., `sfence`). Minimizing the usage of these expensive instructions is key for high performance.

Design details. Nap organizes each per-node PC-view into two PM-resident arrays: a read-only *key array* and a writable *value array* (Figure 7(c)). The key array stores all the keys of the hot set. The value array reserves a *PC entry* for each hot item to record values. A hot item's PC entries are specified via the `pc_pos` field of the corresponding GV entry; for example, in Figure 7, the 5th PC entry in each PC-view belongs to `K5`. Note that each PC entry contains a pointer to the associated key in the key array, to make the NAL recoverable.

Because two threads may update the same hot item but manipulate different PC-views, values of hot items are inconsistent between PC-views. To identify the latest values upon recovery, we adopt a simple version-based mechanism. Each hot item has a monotonically increasing 64-bit version, which is recorded in the GV-view (`version` field in Figure 7). The most significant bit of a version is the *deletion marker*.

Insert/Update operation. In case of no NAL switch, an insert/update operation is performed as following steps:

(1) The access thread searches the GV-view for the targeted item; if the targeted item does not belong to the hot set, the operation is redirected to the raw PM index.
(2) The thread acquires the targeted item's write lock in the GV-view, then obtains a new version.
(3) The thread persists the version with the new value (i.e., $\langle value, version \rangle$ pair) atomically into the targeted PC entry in the local NUMA node.
(4) The thread updates the volatile value in the GV-view (for future lookup operations), and finally releases the lock.

Delete operation. A delete operation has the same process as an insert/update operation, except for the above Step (3): the access thread sets the deletion marker of the obtained version and persists it into its local PC-view.

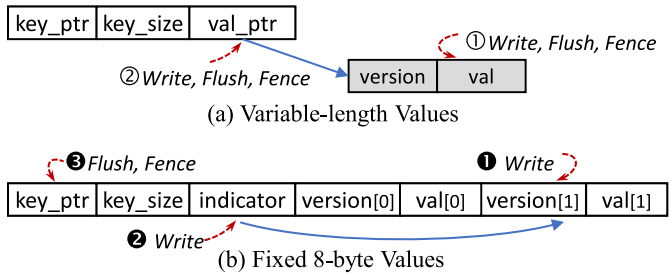

Fig. 8. The structure of two types of PC entry. `key_ptr` *points to corresponding key in the key array and* `key_size` *stands for the size of the key.* ***(a)*** *For variable-length values, we use copy-on-write for failure atomicity. Each PC entry is 24-byte. The grey space of* [*version*, *val*] *is allocated from PM.* ***(b)*** *For fixed 8-byte values, we adopt a lightweight two-incarnation toggle mechanism. Each PC entry is 49-byte (*`indicator` *is 1-byte, every other field is 8-byte) and cache-line-aligned, and contains two incarnations of* ⟨*value*, *version*⟩ *pair.*

Using the version-based mechanism, we can accurately identify the latest value for a hot item from multiple PC-views: The value with maximal version (without deletion marker) is the latest; if the deletion marker of the maximal version is set, the corresponding hot item has been deleted. For example, in Figure 7(c), with the maximal version, `''V-b''` in the PC view of node 1 is the latest value of `K5`.

Now we describe how to guarantee failure atomicity of update to ⟨*value*, *version*⟩ pair with low overhead. Nap adopts two different mechanisms to efficiently support variable-length values and fixed 8-byte values, respectively.

For variable-length values, we leverage **copy-on-write** (**CoW**) to update ⟨*value*, *version*⟩ pair; Figure 8(a) shows the corresponding PC entry. The access thread firstly allocates free PM space and copies ⟨*value*, *version*⟩ pair to it; then, the thread flushes the pair via `clflushopt` instructions followed by a `sfence` (①); finally, the thread updates 8-byte pointer atomically to the address of ⟨*value*, *version*⟩ pair, flushes the pointer via a `clwb`, and issues `sfence` to ensure the persistence is completed (②). We use `clflushopt` (which invalidates flushed cache lines) rather than `clwb` (which does not perform cache invalidation) for ⟨*value*, *version*⟩ pair, so as to save CPU cache space for other operations; this is because that values in PC-views are only read during recovery.

Nap designs a *two-incarnation toggle mechanism* for fixed 8-byte values, which is very common in PM indexes [55] (8-byte value is usually a pointer indicating the location of real data). Figure 8(b) shows the structure of the corresponding PC entry, which is 49-byte and cache-line-aligned. There are two incarnations of 8-byte values and 8-byte versions, and an indicator pointing to the valid incarnation. When writing a new ⟨*value*, *version*⟩ pair, the access thread first copies the pair into the invalid incarnation (❶), which can be calculated according to the indicator (e.g., if the indicator points to the first incarnation, the second one is invalid). Then, the thread toggles the indicator (❷), letting it point to the updated incarnation. Finally, the thread issues a `clwb` to the PC entry followed by a `sfence` (❸). Compared to the CoW, the two-incarnation toggle mechanism saves a flush instruction and a fence instruction, enabling its efficiency. We do not need a fence before toggling the indicator, because writes to the same cache line reach PM *in program order* under **total store order** (**TSO**) architecture of Intel CPUs [24, 43, 82]. Of note, although each PC entry takes up a 64-byte PM space to enforce cache line alignment, the PM consumption is limited; this is because the hot set is small.

4.4 Hot Set Identification

Design goals. In real-world workloads, the hot set keeps changing over time [18]; thus, Nap requires to identify the hot set in real-time. The design goals of identifying hot sets are

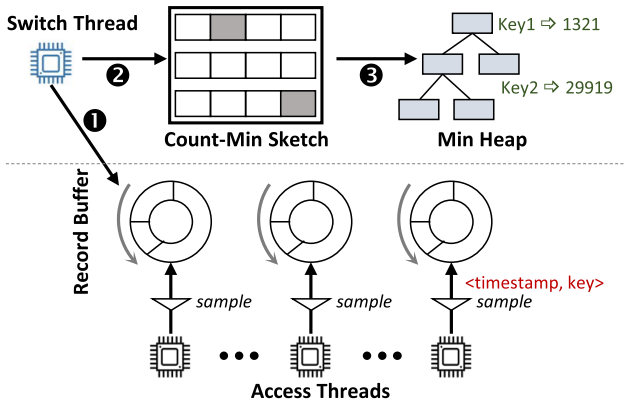

Fig. 9. Hot set identification. *Access threads publish accessed keys into record buffers with sampling. The switch thread uses a count-min sketch to estimate frequency of keys and a min-heap to maintain the current hot set.*

(1) minimizing interferences with foreground index operations, and (2) small memory footprint in the face of infinite streams of index operations.

Design details. Nap uses a dedicated *switch thread* for hot set identification, to detach this process from the critical path of index operations. Figure 9 shows how the switch thread interacts with access threads and identifies the hot set.

Each access thread maintains a circular *record buffer* to publish its access patterns. To reduce interferences caused by hot set identification, access threads use sampling and make writes to record buffers coordination-free. Specifically, every several operations (e.g., 32), an access thread writes a $\langle timestamp, key \rangle$ pair into the record buffer, where the *timestamp* is a 64-bit number generated via `rdtsc` instructions and *key* is the key of the current index operation. The access thread blindly appends $\langle timestamp, key \rangle$ pairs to the circular buffer, regardless of whether the overwritten data has been consumed by the switch thread (i.e., no coordination with the switch thread).

With the help of a count-min sketch [26] and a min-heap, the switch thread digests record buffers in following repeated three steps.

(1) The switch thread chooses a record buffer in a round-robin manner, and fetches a batch (e.g., 8) of new $\langle timestamp, key \rangle$ pairs from it. Two types of $\langle timestamp, key \rangle$ pairs are considered invalid: (1) the *timestamp* is less than maximal timestamp that has been read from corresponding record buffer, indicating we approach the tail of the record buffer; thus, the fetch stops. (2) $(current\ time - timestamp)$ is greater than a threshold value (e.g., 100 ms), indicating this pair is too stale; thus, the pair is skipped. Note that although the timestamps generated via `rdstc` instructions are not strictly synchronized between CPU cores [48, 57], it has not caused any visible impacts for Nap.
(2) For each key fetched from record buffers, the switch thread leverages a count-min sketch to update and estimate its access frequency. The count-min sketch is memory efficient, since it only uses a few small arrays. Sampling used by access threads filters out most infrequent keys, avoiding overflow of the sketch [44].
(3) The min-heap maintains the current hot set in the form of $\langle key, frequency \rangle$ pairs that are ordered by the $frequency$ field. The size of the heap has an upper bound (e.g., 10,000), which can be configured. For a key fetched from record buffers (we call it `K`, and call its estimated frequency `F`), if it is already in the heap, the switch thread updates the corresponding frequency field to `F`; otherwise, the switch thread inserts the ⟨`K`, `F`⟩ pair into the heap. If the

```
1   void Switch_NAL(new_hotset) {
2
3     // Phase 1, initialize the new NAL and install it.
4     NAL_new = Lazy_initialize_NAL(new_hotset, cur_NAL);
5     cur_NAL, pre_NAL = NAL_new, cur_NAL; // logging
6
7     // Phase 2, flush previous NAL into the PM index
8     Wait_for_grace_period();
9     Flush(pre_NAL);
10
11    // Phase 3, release the space used by previous NAL
12    gc_NAL, pre_NAL = pre_NAL, NULL; // logging
13    Wait_for_grace_period();
14    Collect_garbage(gc_NAL);
15    gc_NAL = NULL; // 8-byte atomic write
16  }
```

Fig. 10. Switching to a new NAL. *Global pointers* cur_NAL, pre_NAL *and* gc_NAL *are stored in PM. Line 5 is protected via a global seqlock to ensure access threads can get a snapshot of* ⟨cur_NAL, pre_NAL⟩.

heap is full and F is greater than the frequency of heap root, the thread replaces the pair in the heap root with the ⟨K, F⟩. Every time the heap is modified, we need to adjust its structure to enforce its ordering property.

Periodically (e.g., per 1 s), the switch thread compares the heap with the hot set being used by the current NAL. If there is a big difference between them, i.e., the proportion of different keys exceeds 25%, the switch thread triggers a NAL switch (Section 4.5) with the new hot set (i.e., keys in the heap). The process of comparison is simple. We implement the heap using an array (we call it HeapArray here), and maintain the hot set used by the current NAL in a sorted array (we call it HotArray here). We first sort the HeapArray then compare it with HotArray in an item-by-item manner. The comparison can terminate in advance, once the number of different keys is enough.

All statistics data, including the count-min sketch and the min-heap, are cleared periodically.

Handling uniform workloads. Nap minimizes overhead induced by the NAL under uniform workloads. Specifically, the switch thread detects uniform workloads, under which it initializes an empty NAL (with 0-sized GV-view). For index operations, access threads check the size of GV-view before searching it, which only incurs less than five CPU cycles. The switch thread can use two signals to approximatively identify uniform workloads: ① items in the heap receive less than 10% of all accesses; ② the hottest item in the heap receives comparable accesses (i.e., within 3×) to the coldest.

4.5 NAL Switch

Design goals. Nap switches to a new NAL for handling dynamic workloads. The design goals of the NAL switch lie in two aspects. First, Nap must minimize the blocking of foreground index operations during the NAL switch, to avoid latency spikes. Second, the data races between the switch thread and access threads should be addressed carefully, to guarantee the consistency of the whole system.

Design details. Nap introduces a *three-phase switch*, which is fast and does not block most of the foreground index operations. Its key idea is: The switch thread detects the states of access threads via a grace-period-based method (inspired by epoch-based reclamation [36]), to ensure its modifications are visible for all ongoing and future index operations.

Figure 10 shows the procedure of the NAL switch, which consists of three phases:

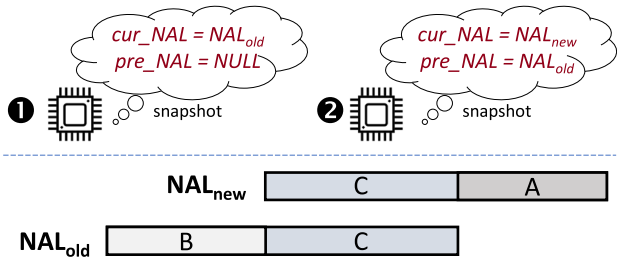

Fig. 11. Different access threads see different system states. *A, B, and C each stands for an exclusive set of items. Types ❶ and ❷ threads are distinguished based on* `pre_NAL` *being null or not.*

```
1  void Insert_type_2(item) {
2
3    if (item.key is in NAL_new) {
4      //  set the val_location to 0
5      Insert_NAL(item, NAL_new);
6      return;
7    }
8
9    if (item.key is in NAL_old) {
10     while (pre_NAL != NULL) {
11       continue;
12     }
13
14     Insert(item); // retry
15     return;
16   }
17
18   Insert_raw_index(item);
19 }
```

```
20 void Lookup_type_2(item) {
21
22   if (item.key is in NAL_new) {
23     gv_entry = Lookup_NAL(item, NAL_new);
24     if (gv_entry.val_location == 0) {
25       return gv_entry.val;
26     }
27     if (gv_entry.val_location == 1) {
28       return Lookup_raw_index(item);
29     }
30     return Lookup_NAL(item, NAL_old).val;
31   }
32
33   if (item.key is in NAL_old) {
34     return Lookup_NAL(item, NAL_old).val;
35   }
36
37   return Lookup_raw_index(item);
38 }
```

Fig. 12. Insert and lookup operations for type ❷ threads.

(1) Initialize a new NAL. The switch thread initializes the new NAL according to the new hot set (line 4, NAL_{new}; we term the current NAL as NAL_{old}). Specifically, the switch thread constructs the GV-view and per-node PC-views; the PC-views are persisted for failure atomicity. For now, the GV-view of NAL_{new} only records locations of values of hot items (i.e., in raw PM index or in NAL_{old}), rather than the values themselves, by setting the `val_location` field in GV entries (Figure 7(b)). Such a *lazy initialization* is necessary for correctness: if we directly copy the latest values to the GV-view of NAL_{new}, the concurrent insert/update/delete operations to raw PM indexes or NAL_{old} will make the value in NAL_{new} stale, violating the correctness of future lookups to NAL_{new}.

Then, the switch thread makes NAL_{new} visible to access threads, by setting global pointers `cur_NAL` and `pre_NAL` to NAL_{new} and `cur_NAL`, respectively (line 5). To ensure that access threads always see the atomic effect of this operation, the line 5 is protected via a global seqlock [7]. Before performing an index operation, the access thread saves a snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair under the protection of the global seqlock, and accesses NAL according to the snapshot (details about the global seqlock are in Section 5).

At this time, the different ongoing index operations may have saved different snapshots of ⟨`cur_NAL`, `pre_NAL`⟩ pair, as shown in Figure 11: Type ❶ accesses threads only see the NAL_{old} and do not realize the concurrent NAL switch; Type ❷ accesses threads see the both NAL_{new} and

```
1   void Wait_grace_period() {
2
3     for (int i = 0; i < ACCESS_THREAD_CNT; ++i) {
4       meta[i] = thread_meta[i];
5     }
6
7     for (int i = 0; i < ACCESS_THREAD_CNT; ++i) {
8       while (true) {
9         // the ith access thread is out of index operations
10        if (meta[i].running == false) {
11          break;
12        }
13        //  the ith access thread has finished an index operation
14        if (thread_meta[i].cnt > meta[i].cnt) {
15          break;
16        }
17      }
18    }
19  }
```

Fig. 13. Wait a grace period. `ACCESS_THREAD_CNT` *is the number of access threads.* `thread_meta` *is a global array, which is updated by access threads and probed by the switch thread.*

NAL_{old}. Specifically, types ❶ and ❷ threads are distinguished based on `pre_NAL` being null or not. For type ❶ threads, they manipulate NAL_{old} and the workflow of index operations is the same as in cases of no NAL switch (Sections 4.2 and 4.3). The index operations becomes a bit complicated for type ❷ threads (as shown in Figure 12):

(1) For an insert/update/delete operation, if the targeted item belongs to NAL_{new} (this item may also be in NAL_{old}, e.g., range C in Figure 11), NAL_{new} absorbs this operation like the case of no NAL switch (Section 4.3); besides, the thread copies the value into the corresponding GV entry, and updates the `val_location` field to `0` in order to indicate the value can be served for future lookups. If the targeted item falls in NAL_{old} and not in NAL_{new} (range B in Figure 11), the operation is blocked until the global pointer `pre_NAL` becomes `NULL` (i.e., phase 3 of the three-phase switch, see below); then, the operation is retried. Otherwise, the operation is redirected to the raw PM index.
(2) For a lookup operation, the thread checks GV-view of NAL_{new}, GV-view of NAL_{old}, and the raw PM index one by one. In the case that the targeted item falls in NAL_{new}, the thread checks the `val_location` field: if the value can not be served from the NAL_{new} (i.e., `val_location` is not `0`), the thread fetches the value from NAL_{old} (for range C in Figure 11) or the raw PM index (for range A) according to the `val_location` field. Range query operations experience the same workflow: access threads search NAL_{new}, NAL_{old} and the raw PM index in order, then merge results.

(2) Flush NAL_{old}. In this phase, the switch thread first waits for a *grace period* to ensure all access threads become type ❷ (line 8). Our grace period mechanism is simple: each access thread publishes its states into a slot in a global array; a slot consists of two fields: a boolean `running` and a 64-bit `cnt`. The access thread sets its `running` and increases `cnt` when starting an index operation (before saving the snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair), and resets the `running` when completing the operation. The switch thread probes the global array until every access thread is out of index operations (`running` is false) or has finished an index operation (`cnt` is changed), as show in Figure 13. After this grace period, all the access threads realize the concurrent NAL switch for ongoing and future index operations, i.e., they are type ❷ threads; hence, the NAL_{old} will never

be modified (recall that insert/update/delete operations to NAL_{old} are blocked for type ❷ threads). Now, the switch can flush the latest values in the GV-view of NAL_{old} to the raw PM index rapidly (via invoking interfaces of the raw PM index) without considering any data race (line 9).

(3) Recycle NAL_{old}. Now, the NAL_{new} and the raw PM index reflect complete and consistent states of the system. The switch thread needs to recycle the DRAM/PM space occupied by NAL_{old}. It first saves the NAL_{old} into a global pointer `gc_NAL` and sets the `pre_NAL` to `NULL` (line 12). Then, the switch thread waits for a grace period to ensure no ongoing and future lookup operations are performed on NAL_{old} (line 13). Finally, the DRAM and PM space used by NAL_{old} is released safely (line 14), and `gc_NAL` is set to `NULL` (line 15). The access threads that realize the null `pre_NAL` are in a normal condition without any blocking; for a lookup operation to NAL_{new}, if the targeted value is not in the GV-view, the access thread fetches the value from the raw PM index, saves it to the GV-view, and updates corresponding `val_location` field to `0`.

In the above three-phase switch, the insert/update/delete operations to a part of NAL_{old} (i.e., range `B` in Figure 11) are blocked during the phase 2. Such a blocking has only a small impact on the system for two reasons. First, since the new hot set is maintained by NAL_{new}, items in the range `B` is cold, receiving a negligible percentage of accesses. Second, since the hot set is small and flushing items from NAL_{old} to the raw PM index is data-race-free, the phase 2 is fast.

Failure atomicity. The switch thread guarantees failure atomicity of three global pointers: `cur_NAL`, `pre_NAL`, and `gc_NAL`. These three pointers are allocated in PM and persisted when modified. The switch thread also maintains a small PM undo log. For lines 5 and 12 of Figure 10, the switch thread records undo log entries for failure atomicity. For line 15, an 8-byte atomic write is enough.

4.6 Recovery

In this section, we detail the recovery of Nap after a normal shutdown and system crash.

Recovery after a normal shutdown. Before a normal shutdown, Nap first flushes the hot items from the NAL's GV-View to the raw PM index. Then, it frees the PM space consumed by per-node PC-views. In this way, during recovery, Nap only needs to initialize an empty NAL and restore the underlying raw PM index.

Recovery after a system crash. In this case, the recovery is somewhat complicated. We divide the recovery into four steps.

(1) Reconstruct the raw index. We invoke the recovery procedure of the underlying raw PM index. Note that this step is necessary, even without Nap.

(2) Reconstruct NALs. Nap scans the undo log and global pointers (i.e., `cur_NAL`, `pre_NAL` and `gc_NAL`), and constructs the valid version of these pointers. After that, if pointer `pre_NAL` is not null, it means that the system crashes during the NAL switch.

(3) Merge NALs into the raw index. In this step, Nap first merges the NAL pointed by `pre_NAL` (if not null) into the raw index; then, it merges the NAL pointed by `cur_NAL` in the same way. Merging a NAL needs to identify the latest committed items from the NAL's PC-views, which relies on the techniques we propose in Section 4.3. Specifically, ① Nap extracts committed items and associated versions for each PC-view. For variable-length values, the committed $\langle value, version \rangle$ pairs are indicated by 8-byte pointers in value arrays; for 8-byte values, the committed $\langle value, version \rangle$ pairs are specified by 1-byte indicators in value arrays (see Figure 8). The associated keys are extracted from the PC-views' key arrays. ② Next, for each item that appears in multiple PC-views, Nap identifies the value with maximal version as the latest one. ③ Finally, Nap inserts these latest committed items into the raw PM index by invoking its interfaces.

(4) Reclaim PM space. Nap frees the PM space of NALs that are pointed by `cur_NAL`, `pre_NAL` and `gc_NAL`, to avoid the memory leak. The PM space includes two parts: (1) value arrays and key arrays in PC-views and (2) $\langle value, version \rangle$ pairs, which are allocated in case of variable-length values. Finally, Nap clears the undo log. At this point, Nap is in a consistent state.

4.7 Correctness

4.7.1 Definitions.

— `IL_RAW`: isolation level of the underlying raw PM index.
— `IL_NAP`: isolation level of the Nap-converted index.

4.7.2 Isolation Guarantee.

Theorem 1. *For range queries,* `IL_NAP` *is equal to the lower level of one between* `IL_RAW` *and read committed.*

Proof. In Nap, a range query merges committed results from the NAL and raw PM index *without coordination*, so it is not atomic with concurrent updates. Hence, range queries reach up to read committed. □

Theorem 2. *For point queries,* `IL_NAP` *is equal to* `IL_RAW`.

Proof. For hot items managed by NALs (i.e., NAL_{new} and NAL_{old}), Nap enforces linearizability for point queries to them. There are four cases for two conflicting operations.

— If two conflicting operations target the same NAL, readers-writer locks in the NAL's GV-view serialize them.
— If a thread updates an item in NAL_{old}, future lookups[2] to NAL_{new} can see the value due to the lazy initialization.
— If a thread updates an item in NAL_{new}, it means the NAL_{new} has been installed. Hence, all future lookups will see the NAL_{new} and get the correct value.
— If two conflicting operations OP_1 and OP_2 perform updates on NAL_{old} and NAL_{new}, respectively, all future lookups will see OP_2, which means OP_1 happens before OP_2 in the linearizable history. This is legal since it is impossible that OP_1 is invoked after OP_2's response. □

4.7.3 Failure Atomicity.

Theorem 3. *Nap-converted indexes do not change failure atomicity semantic of raw PM indexes.*

Proof. Nap ensures that lookups after recovery can find the latest committed updates to NALs. First, in a single PC-view, Nap adopts the two-incarnation toggle mechanism and CoW for atomic persistence. Second, among multiple PC-views in a NAL, Nap stores values along with increasing versions, which are used for accurately identifying the latest values upon recovery. Third, changes to the global PM pointers `cur_NAL` and `pre_NAL` are protected by undo logging; upon recovery, we first flush the NAL pointed by `pre_NAL` and then the one pointed by `cur_NAL`, so as to ensure only the latest values appear in the raw PM index. □

5 IMPLEMENTATION

We have implemented Nap in C++ (~2,000 lines of code). Nap provides a template class in the form of "`template<T> class Nap`", where `T` is a wrapper class for a concurrent PM index with specific index operation interfaces invoked by Nap. Our programming experience shows that converting a PM index using Nap needs roughly 30 lines of wrapper class codes. We use C++ `unordered_map`

[2]For an operation, its *future* operations are operations that are invoked after its response.

to organize the GV-view by default; if the underlying raw PM index supports range query (e.g., B+ tree), we use C++ `map`.

PM space management. We leverage PMDK [4] to manage PM space. Specifically, for each NUMA node, we initialize a PMDK pool, from which Nap allocates PM space for PC-views. To reduce expensive PMDK allocation upon CoW (Section 4.3), we adopt a simple customized allocator. Each thread requests 1 MB chunks from its local PMDK pool, and allocates PM for CoW using classic slab mechanism. The addresses of chunks are recorded in the PM, and the allocator metadata is maintained in the DRAM. Upon recovery, after flushing the PC-views into the underlying raw PM index, Nap frees these used chunks.

Mitigate cache coherence traffic. Since Nap uses fine-grained concurrency (e.g., three-phase switch) for high performance, which inevitably induces communication between different threads, we strive to mitigate cache coherence traffic to ensure Nap's multi-core scalability. Specifically, to reduce cache line movements between the switch thread and the access threads, the switch thread digests each per-access-thread record buffer in a batched manner: it fetches and processes one cache line (rather than one $\langle timestamp, key \rangle$ pair) at a time. In addition, to eliminate unnecessary cache coherence traffic between access threads, Nap leverages a global seqlock [7] to control concurrent accesses to global pointers (i.e., `cur_NAL` and `pre_NAL`). Using global seqlock, access threads detect concurrent writes (caused by the switch thread) by reading versions, and thus do not trigger any memory writes when getting the snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair (Line 5 in Figure 10). In this way, unlike a counter-based reader-writers lock, the global seqlock ensures that no cache coherence traffic is generated between access threads when accessing these global pointers.

Data persietence. As described in Section 4.3, Nap carefully chooses different flush instructions (i.e., `clwb` and `clflushopt`). However, we found that `clwb` can not reduce latency of future PM reads. This is because in current hardware implementation, `clwb` invalidates cache lines (like `clflushopt`) [46]. We hope that in next-generation CPUs supporting PM, `clwb` can retain flushed data in the cache, so as to enable optimizations of choosing different flush instructions.

6 EVALUATION

In this section, we use a number of microbenchmarks and applications to evaluate Nap, seeking to answer the following questions:

- How does Nap-converted PM indexes compare with original PM indexes? (Section 6.2)
- How does Nap perform when value size is variable? (Section 6.3)
- How does Nap react to dynamic workloads? (Section 6.4)
- How do the characteristics of workloads and NUMA configurations affect the performance of Nap? (Section 6.5)
- How does the two-incarnation toggle mechanism improve Nap's performance? (Section 6.6)
- Can Nap improve the performance of PM/DRAM hybrid indexes? (Section 6.7)
- How does Nap compare with Node Replication? (Section 6.8)
- What are the overheads incurred when using Nap? (Section 6.9)
- What is the benefit of Nap to real applications? (Section 6.10)
- How does Nap perform when UPI uses the snoop-based cache coherence protocol? (Section 6.11)

6.1 Experimental Setup

The experiments are conducted on a 4-socket (NUMA node) machine, where UPI uses the directory-based cache coherence protocol. Each NUMA node is populated with an 18-core Intel Xeon Gold

6,240 M CPUs, three 128 GB Optane DIMMs and three 32 GB DDR4 DIMMs, resulting in a machine with 72 CPU cores, 1.5 TB PM and 384 GB DRAM. Our machine runs Ubuntu 18.04 with Linux kernel version 5.4.0.

Unless otherwise stated, for Nap, the size of the hot set is configured to 100 K, and the switch thread tries to perform the NAL switch per 0.2 seconds. Each per-core record buffer is 300 KB. The count-min sketch contains 3 counter arrays, each with 32-bit 850,000 counters. The sampling interval is 32. The switch thread is enabled in all the experiments.

Workloads. We leverage a YCSB-like benchmark to evaluate the performance of PM indexes. The benchmark contains five types of workloads: (1) *write-intensive:* 50% lookup and 50% update/insert, (2) *read-intensive:* 95% lookup and 5% update/insert, (3) *write-only:* 100% update/insert, (4) *read-only:* 100% lookup, and (5) *scan-intensive:* 95% range query and 5% update/insert. By default, the key space (i.e., the range of keys) is 200 million and the key popularity follows a Zipfian distribution with parameter 0.99 (the default setting in YCSB [25]). For each experiment, we first load 16 million items then perform the workloads, which contains 64 million index operations. The ratio of insert operations to update operations is about 1:3. We use 15-byte keys and 8-byte values.

6.2 Real Indexes

Using Nap, we convert five state-of-the-art PM indexes:

— *CCEH* [65]. An extendible hashtable that is structured as a set of segments pointed by a global directory. It uses readers-writer locks for concurrency control.
— *Clevel* [23]. A lock-free version of level hashing [84], which is organized as two bucket arrays.
— *P-CLHT* [55]. PM version of CLHT [30], which is a linked-list-based hashtable. It supports lock-free lookups and uses bucket-grained locks for other operations.
— *P-Masstree* [55]. PM version of Masstree [62], a trie-like concatenation of B+ tree nodes. It adopts lock-free lookups and lock-based writes.
— *FAST_FAIR* [43]. A PM B+ tree with lock-free lookups and lock-based writes.

For CCEH, Clevel, and P-CLHT, we use the source code from [5], which relies on PMDK for PM allocation and supports variable-length keys. We modify the code to make each thread allocate PM from its local PMDK pool. For CCEH, we replace the global directory lock with an in-DRAM distributed readers-writer lock [2], avoiding its scalability issues. For P-Masstree and FAST_FAIR, we use the source code from [3] and modify the code for allocation with PMDK; besides, we improve range query implementations by making them return both keys and values. Of note, we do not use our customized allocator (Section 5) for these indexes; this is because the customized allocator cannot provide failure atomicity for each (de)allocation operation due to its DRAM-resident metadata.

Throughput under write/read-intensive workloads. Figure 14 shows the throughput of these PM indexes under write-intensive and read-intensive workloads, and we make the following observations:

First, compared with the original indexes, Nap-converted indexes yield much better scalability under both write-intensive and read-intensive workloads. Specifically, in four-node environment (i.e., 72 threads), Nap improves the throughput by 1.26× (FAST_FAIR) to 2.3× (CCEH) for write-intensive workloads and 1.18× (P-Masstree) to 1.56× (P-CLHT) for read-intensive workloads. This is because the NAL of Nap absorbs plenty of operations, where the per-node PC-views eliminate the remote PM writes and the GV-view eliminates the remote PM reads. Note that the global GV-view induces remote DRAM accesses; yet, remote DRAM accesses exhibit much higher performance than remote PM accesses: 5.7× higher throughput for writes (20 GB/s : 3.5 GB/s) and

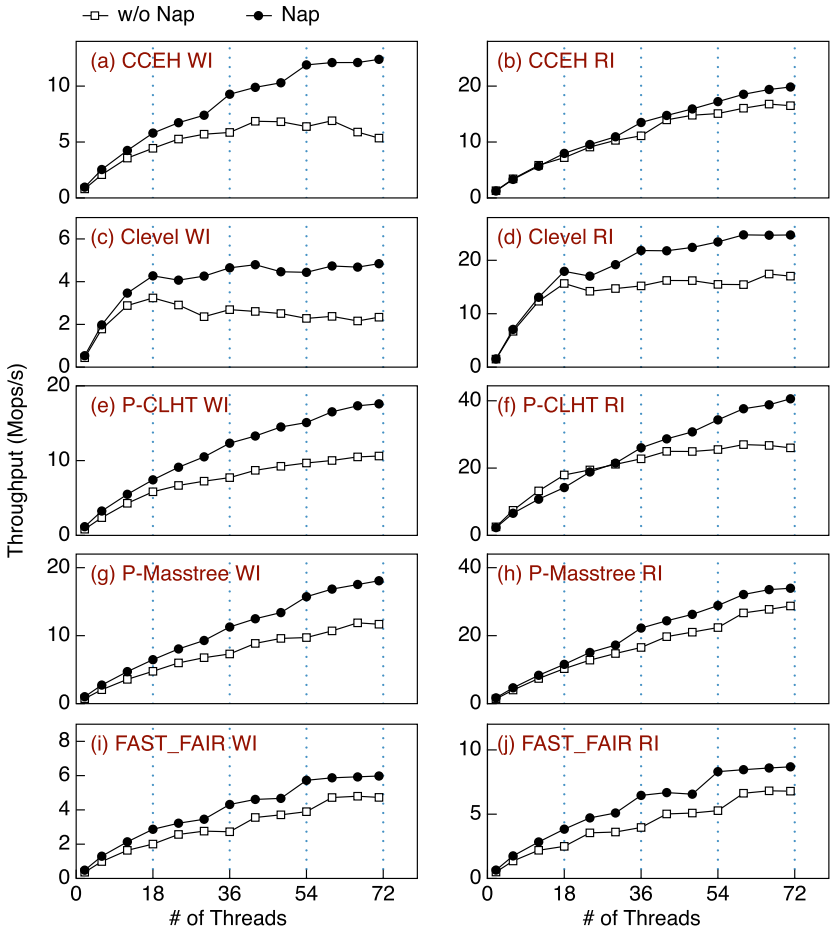

Fig. 14. Throughput under write/read-intensive workloads. ***WI****: write-intensive workloads;* ***RI****: read-intensive workloads. Vertical lines show the boundaries between NUMA nodes.*

2× lower latency for reads (200 ns : 400 ns). Figure 15 reports the NAL's hit ratio, which archives 45% ∼ 54%. There is a small gap between these hit ratios and the theoretical upper limit (i.e., 0.58). This is because we use the probabilistic data structures (e.g., count-min sketch) and sampling to capture the hot set, resulting in a small degree of inaccuracy.

Second, even within a single NUMA node, Nap-converted indexes outperform the original ones (except P-CLHT in read-intensive workloads). This is mainly because (1) For lookup operations, the GV-view avoids the latency of PM reads. (2) For insert/update operations, the two-incarnation toggle mechanism of PC-views minimizes the overhead of PM writes. For P-CLHT, a highly optimized hashtable for cache locality, most of lookup operations are met in CPU caches under read-intensive workloads within a NUMA node, enabling its high performance. Hence, it outperforms the Nap-converted version slightly, which induces overheads of searching the GV-view for every lookup operations.

Third, compared with tree-based PM indexes, hashtable-based PM indexes are more vulnerable to NUMA architectures (particularly for Clevel, Figure 14(c) and (d)). These hashtables always use several continuous and large arrays for fast indexing (e.g., the global directory of CCEH, bucket

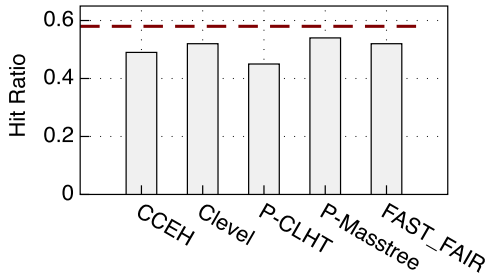

Fig. 15. Hit ratio of the NAL (write-intensive workloads, 72 threads). *The red dotted line represents the theoretical upper limit of the hit ratio (*i.e., *0.58) for our experiments (the size of hot set is 100 K, the Zipfian parameter is 0.99, the key space is 200 million).*

arrays of Clevel, and P-CLHT). For threads that do not reside on the same NUMA nodes with these arrays, almost all PM accesses to these arrays are remote, limiting the available PM bandwidth and further deteriorating the performance. The worst one is Clevel, because it only uses two bucket arrays for indexing; by contrast, in addition to global arrays, CCEH uses segments and P-CLHT uses linked list, which can be allocated on different NUMA nodes, increasing the available PM bandwidth of PM indexes.

Finally, compared with crafted PM indexes (i.e., CCEH, Clevel, and FAST_FAIR), Recipe-converted PM indexes [55] (i.e., P-CLHT and P-Masstree) deliver much higher throughput (e.g., P-Masstree outperforms FAST_FAIR by more than 2×). The superior performance of Recipe stems from two reasons. First, Recipe guarantees weaker consistency level, i.e., buffered durable linearizability [37, 53]. Second, Recipe can leverage mature concurrent DRAM indexes, rather than building PM indexes from the scratch. For example, since Masstree uses a trie-like structure to embed keys in tree nodes for improving cache locality, when applied to PM using Recipe, P-Masstree can reduce massive expensive PM reads compared with FAST_FAIR. Combining Nap with Recipe is a promising approach for converting existing concurrent DRAM indexes into NUMA-aware PM indexes.

Throughput under write/read-only workloads. Figure 16 shows the throughput under write-only and read-only workloads. We make the following observations.

First, in case of 72 threads spanning four NUMA nodes, Nap boosts the throughput by 1.32× (FAST_FAIR) to 6.15× (CCEH) for write-only workloads and 1.15× (P-Masstree) to 1.55× (FAST_FAIR) for read-only workloads. Such improvement results from the NAL, which handles hot items in an efficient and NUMA-aware manner.

Second, under read-only workloads, Clevel and P-CLHT perform better than Nap-converted counterparts when threads span one or two NUMA nodes (i.e., thread counts ≤36). This is because in Nap, for items that are not maintained in the NAL, accesses to them need unnecessary searches of the NAL. These overheads overshadow the benefits of absorbing accesses to hot items with the NAL. Adopting a bloom filter (to check if an item belongs to the NAL) should mitigate the overheads of searching the NAL.

Throughput under scan-intensive workloads. Figure 17 shows the range query performance of P-Masstree and FAST_FAIR. We set the query range to 10. With 72 threads spanning four NUMA nodes, Nap reduces the throughput of P-Masstree and FAST_FAIR by 3% and 14%, respectively. This is because Nap needs to search both the GV-view and the raw PM index; yet, with the good locality of the GV-view and low latency of DRAM, the extra overhead is bounded. Note that P-Masstree has much better range query performance than FAST_FAIR. This is because when supporting

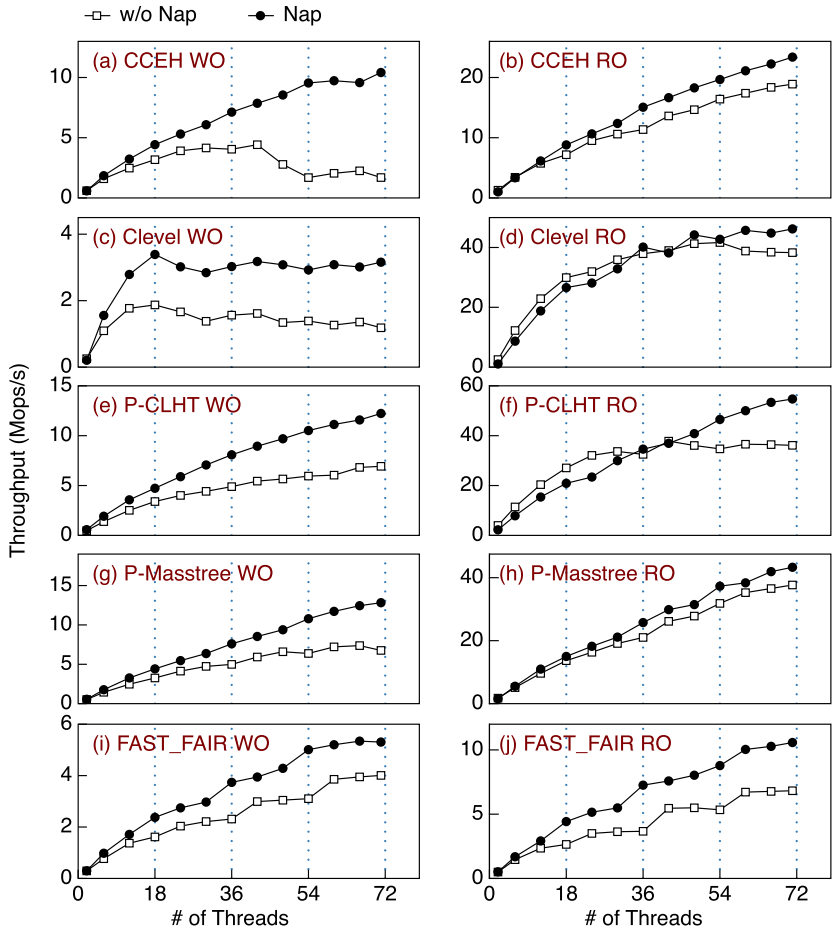

Fig. 16. Throughput under write/read-only workloads. ***WO****: write-only workloads;* ***RO****: read-only workloads. Vertical lines show the boundaries between NUMA nodes.*

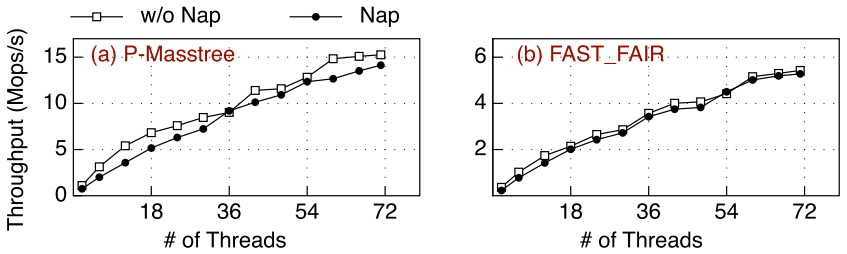

Fig. 17. Throughput under scan-intensive workloads.

variable-length keys, FAST_FAIR suffers a large number of expensive PM reads, which results from frequent references to keys [61].

Latency. Figure 18 depicts the latency distribution of P-CLHT under write-intensive workloads. The number of access threads is 72. We omit other PM indexes that have similar results. Nap decreases the median latency by 46% (from 3.77μs to 2.04μs) and the 99th percentile latency by

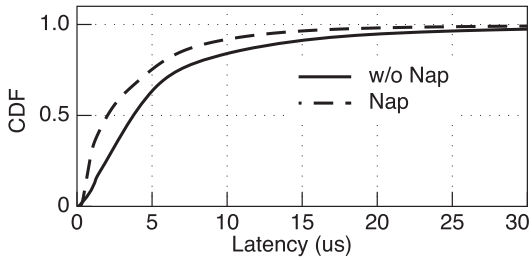

Fig. 18. Latency distribution (P-CLHT, 72 threads, and write-intensive workloads). *The 50th and 99th latencies of the original index are 3.77µs and 49.95µs (not shown in the figure), respectively. The 50th and 99th latencies of Nap-converted index are 2.04µs and 27.64µs, respectively.*

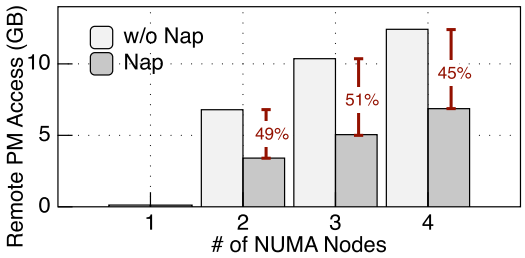

Fig. 19. The amount of data via remote PM accesses (P-CLHT, write-intensive workloads). *We run 18, 36, 54, and 72 threads to measure results under different NUMA nodes.*

45% (from 49.85µs to 27.64µs). The improvement is mainly from the per-node PC-views, which eliminate remote PM writes for hot items, reducing the possibility of multiple threads within a node access remote PM simultaneously (recall that when multiple threads write remote PM, the bandwidth collapses, affecting the access latency, Figure 2).

Quantitative measurement of remote PM accesses. We use Intel's PCM tools [6] to measure the remote PM accesses. The `pcm.x` sub-tool provides the amount of data through UPI links and the `pcm-numa.x` sub-tool monitors remote DRAM accesses. Leveraging the two sub-tools, we calculate the remote PM accesses of P-CLHT under write-intensive workloads. Figure 19 reports the result. Nap reduces remote PM accesses by 45%–51%, enabling its high performance.

6.3 Variable-length Values

This experiment tests variable-length values, which trigger CoW in Nap. We run P-CLHT and randomly select the value size from 8 bytes to 256 bytes. We also evaluate a Nap invariant that replaces our customized allocator with PMDK allocator. Figure 20 presents the result, from which we make three observations. First, due to more flush and fence instructions in CoW, Nap's throughput degrades (compared with Figure 14(e) and (f)). Second, Nap-converted P-CLHT outperforms P-CLHT by 1.36× and 1.39× under write-intensive and read-intensive workloads, respectively. This is because Nap mitigates remote PM accesses and adopts low-overhead customized allocator for CoW. Third, compared with PMDK allocator, our customized allocator brings 20% and 13% performance gain under write-intensive and read-intensive workloads, respectively. This is because the customized allocator maintains allocation metadata in DRAM, reducing persistency overhead. Note that even with PMDK allocator, due to the NUMA-aware design, Nap-converted P-CLHT

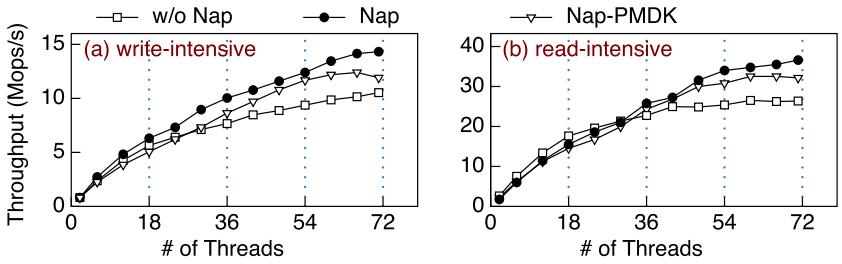

Fig. 20. Throughput of P-CLHT. *The value size is randomly selected from 8 bytes to 256 bytes.* `Nap-PMDK` *uses PMDK to allocate values in the NAL.*

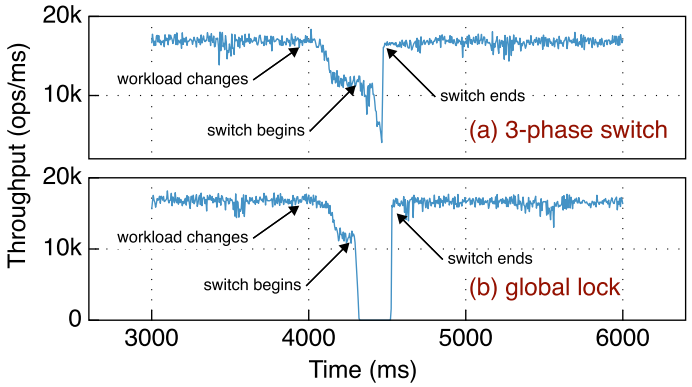

Fig. 21. Throughput over time with workloads change (P-Masstree, 71 threads, and write-intensive workloads).

still outperforms P-CLHT by 1.13× and 1.21× under write-intensive and read-intensive workloads, respectively.

6.4 Dynamic Workloads

In this experiment, we evaluate Nap's ability to react to dynamic workloads by changing the popularity of keys. We compare our three-phase switch mechanism with a conservative mechanism that uses a global readers-writer lock: the switch is protected by the write lock, and every index operation is protected by the read lock. To avoid the cache thrashing among access threads caused by the centralized global lock, we apply per-core reader indicator [2]. We run Nap-converted P-Masstree under write-intensive workloads with 71 threads. (One core is reserved to record total throughput per 5 ms.) Figure 21 shows the throughput over time. The workload changes at time 4 s. Since the NAL can not absorb the accesses to current hot set, the throughput drops. After about 200 ~ 300 ms, Nap identifies the new hot set (recall that the switch period is 0.2 s, Section 6.1), and triggers the NAL switch. In our three-phase switch, the throughput can be maintained more than 10 K ops/ms for about 130 ms, then drops to 4 K ~ 8 K ops/ms for about 35 ms. This is because the three-phase switch only blocks some insert/update operations to a part of old NAL during phase 2. However, when using the global lock, the system is unavailable (i.e., throughput is 0) for about 195 ms. To sum up, using the three-phase switch, Nap is robust enough to react to dynamic workloads quickly without sacrificing availability.

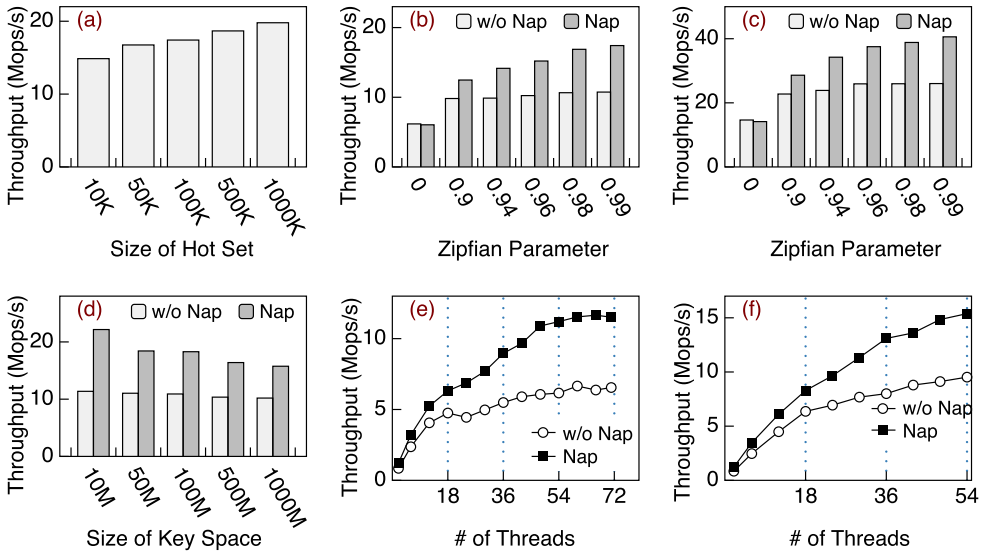

Fig. 22. Sensitivity analysis (P-CLTH). *All experiments except (c) use write-intensive workloads.* ***(a)*** *Varying the size of hot set (72 threads).* ***(b)*** *Varying the Zipfian parameter (write-intensive workloads, 72 threads).* ***(c)*** *Varying the Zipfian parameter (read-intensive workloads, 72 threads).* ***(d)*** *Varying the size of key space (72 threads).* ***(e)*** *Two Optane DIMMs per NUMA node.* ***(f)*** *Four Optane DIMMs per NUMA node.*

6.5 Sensitivity Analysis

Size of hot set. Figure 22(a) shows how the configured hot set size affects the Nap's performance. As the size of hot set increases from 10 K to 1 M, the throughput grows by 1.33×, and the percentage of operations absorbed by the NAL increases from 43% to 63%. Yet, using a large hot set consumes more PM/DRAM space and prolongs the time of NAL switch and system recovery.

Skewness of workloads. We study how the skewness of workloads affects Nap's performance. Figure 22(b) shows the result under write-intensive workloads. We make three observations. First, with increasing skewness, Nap's improvement over original indexes grows. This is because the NAL can absorb more index operations. For the medium skewness case (i.e., 0.9 Zipfian parameter), Nap boosts the throughput by 1.27×. Second, under uniform workloads (i.e., 0 Zipfian parameter), throughput of both indexes drops, since there are more insert operations in uniform workloads, leading the P-CLHT to resize frequently. Third, the throughput of both indexes is comparable under uniform workloads. This is because Nap handles uniform workloads by initializing an empty NAL, which minimizes the overhead of searching the NAL.

Figure 19(c) shows the result under read-intensive workloads. Nap can obtain more performance gain with higher skewness. This is because as skewness increases, more lookup operations are served by the GV-view of the NAL, so the GV-view can translate more expensive PM reads into fast DRAM reads, boosting the system throughput.

Through this experiment, we can also quantify the overhead of the switch thread. Specifically, with uniform key distribution, Nap degrades throughput by 2.1% under write-intensive workloads and 3.5% under read-intensive workloads. Since the empty NAL only incurs less than five CPU cycles on every index operation, the performance degradation mainly results from interaction with the switch thread: Index threads publish accessed keys into record buffers and may trigger cache coherence traffic when the switch thread digests record buffers. In addition, record buffers, the count-min sketch, and the min-heap occupy part of limited CPU cache space.

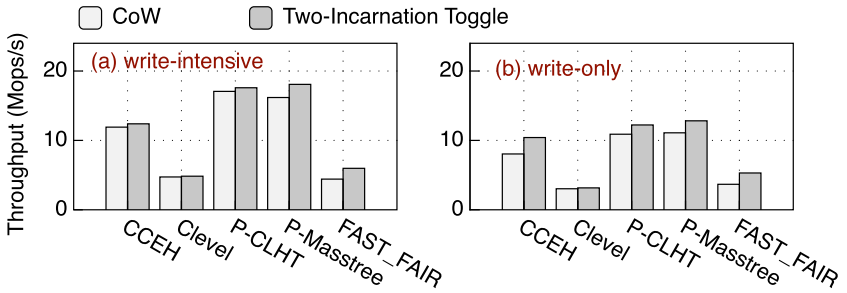

Fig. 23. Performance impact of two-incarnation toggle mechanisms (72 threads).

Size of key space. Figure 22(d) presents the throughput of P-CLHT and its Nap-converted version with varying key space. As the key space increases, the number of hot items increases, degrading the throughput of Nap, which maintains a fixed-size hot set. Even for a very large key space, i.e., 1,000 million, Nap can boost the throughput by 1.55×, which demonstrates that Nap can handle large-scale workloads.

Different PM bandwidth configurations. Here, we change PM bandwidth configurations by adding/removing Optane DIMMs, and show how the available PM bandwidth affects Nap. We get two new PM bandwidth configurations: (1) 2 Optane DIMMs per node; (2) 4 Optane DIMMs per node (only 3 nodes due to the total of 12 DIMMs). Figure 22(e) and (f) show the results of (1) and (2), respectively. With 2 Optane DIMMs per node, the available PM bandwidth drops and remote PM access suffers lower write bandwidth, degrading the throughput of PM indexes; yet, under this configuration, by mitigating remote PM accesses, Nap boosts the throughput of the original index by 1.76×, which is higher than improvement under default 3-DIMMs-per-node configuration (1.66×, Figure 14(e)). Under 4-DIMMs-per-node configuration, Nap outperforms the original index by 1.62×. Overall, Nap is efficient under different PM bandwidth configurations.

6.6 Effectiveness of Two-Incarnation Toggle Mechanism

To understand the effectiveness of our proposed two-incarnation toggle mechanism, we compare it with CoW. Since the two-incarnation toggle mechanism is an optimization for write operations, we only test write-intensive and write-only workloads here. We set the number of threads to 72. Figure 23 shows results of different PM indexes. The two-incarnation toggle mechanism boosts the throughput by 2.3% (Clevel) to 35.2% (FAST_FAIR) under write-intensive workloads and 4.2% (Clevel) to 44.4% (FAST_FAIR) under write-only workloads. This is because compared with CoW, for every write to the NAL, the two-incarnation toggle mechanism saves one fence instruction and one flush instruction. In addition, it reduces a 256-byte PM media write (PM's internal access granularity is 256-byte) for each update/insert operations absorbed by the NAL, saving PM's limited write bandwidth.

6.7 Integrating Nap to PM/DRAM Hybrid Indexes

Some tree-based PM indexes adopt a PM/DRAM hybrid approach: They place internal tree nodes on the DRAM to reduce persistence overhead [20, 67]. During recovery, such PM/DRAM hybrid indexes need to reconstruct internal tree nodes according to PM-resident leaf tree nodes. Since these PM/DRAM hybrid indexes avoid remote PM accesses when traversing internal tree nodes, they should have a better NUMA scalability than pure PM indexes. In this experiment, we modify FAST_FAIR to let it allocate DRAM for internal tree nodes (we term it FAST_FAIR-Hybrid), and test its performance. We also integrate Nap to FAST_FAIR-Hybrid. We run write-intensive

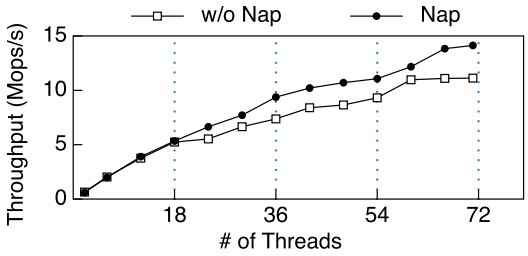

Fig. 24. Performance of FAST_FAIR-Hybrid (write-intensive workloads, 72 threads).

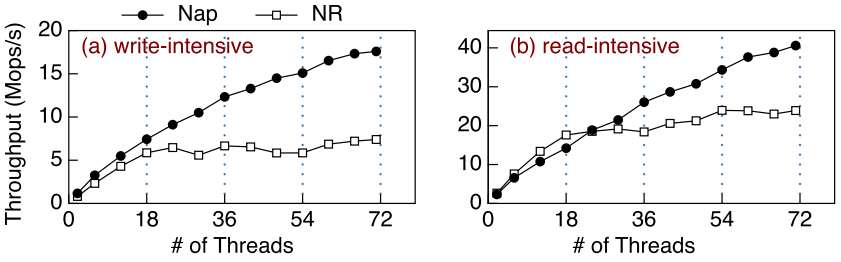

Fig. 25. Performance of Nap and NR (P-CLHT).

workloads. Figure 24 shows the results. We make two observations. First, FAST_FAIR-Hybrid delivers much better performance and scalability than FAST_FAIR (see FAST_FAIR's performance in Figure 14(i)), e.g., FAST_FAIR-Hybrid outperforms FAST_FAIR by 2.35× in case of 72 threads. This is because FAST_FAIR-Hybrid eliminates expensive PM read and persistence overhead in internal tree nodes. Second, despite the adoption of PM/DRAM hybrid approach, using Nap can still improve throughput when threads span multiple NUMA nodes. Specifically, in case of 72 threads, Nap boosts FAST_FAIR-Hybrid by 1.27×. This is because in FAST_FAIR-Hybrid, accessing to leaf nodes can induce remote PM writes and reads. In contrast, the NAL in the Nap-converted counterpart absorbs accesses to hot items in a NUMA-aware manner.

6.8 Comparison with NR

We compare Nap with **Node Replication** (**NR**) [15] to present some key insights of designing NUMA-aware PM indexes. We put the shared log of NR in the DRAM and disable log recycle. Figure 25 shows the throughput of NUMA-aware P-CLHT converted by Nap and NR. Note that NR-converted P-CLHT is not crash-consistent: upon crash, the shared log is lost and P-CLHT on different NUMA nodes may be inconsistent. In case of 72 threads, Nap outperforms NR by 2.34× and 1.69× under write-intensive and read-intensive workloads, respectively. The inefficiency of NR on PM indexes stems from two reasons. First, by maintaining consistent replicas between NUMA nodes, each insert/update operation consumes n times more PM bandwidth (n is the number of NUMA nodes), limiting the throughput. Second, NR leverages flat combining [41] (a technique that uses a combiner to execute a batch of collected updates) to handle updates within a node. Flat combining can mitigate cache thrashing but restrict concurrency to a single thread; yet, the single-thread performance of PM indexes is much lower than that of DRAM indexes, due to expensive flush/fence instructions and high PM read latency. Combining previous experimental results (Section 6.2), we can conclude that the most important performance determinant of NUMA-aware PM indexes is precious PM bandwidth of both local and remote accesses (rather than cache

Table 2. Consumption of DRAM and PM in Nap

DRAM				PM
record buffers	count-min sketch	min-heap	GV-view	PC-views
21.1 MB	9.7 MB	4.2 MB	3.6 MB	30.1 MB
Altogether, **38.6 MB** DRAM and **30.1 MB** PM				

We ignore some very small usage, such as the 64-byte persistent undo log used by the switch thread.

Table 3. Recovery Time

Index Type	CCEH	Clevel	P-CLHT	P-Masstree	FAST_FAIR
Time (ms)	477	522	432	306	963

thrashing); thus, like Nap, a high-performance NUMA-aware PM index should reduce remote PM accesses without consuming extra local PM bandwidth.

6.9 Overheads of Nap

The overheads of Nap lie in two aspects: memory consumption and recovery time.

Memory consumption. Table 2 shows the memory consumption by Nap in our evaluation (4 NUMA nodes and 72 threads), and the total memory consumption is less than 70 MB. Specifically, since our NAL only maintains the hot set, the size of the min-heap, GV-view and PC-views are limited. Besides, by using sampling, the small-sized count-min sketch and per-core record buffers are enough.

Recovery time. Table 3 reports the recovery time of Nap-converted PM indexes. Due to the limited size of NAL, the recovery time is bounded, which is less than 1 s. FAST_FAIR has the longest recovery time, since it has the worst performance of update/insert operations (see Figure 16).

6.10 Real Application

To show the benefits that a Nap-converted PM index can bring to real applications, we build a networked PM-based key-value store. The key-value store uses eRPC [47] for network communication, P-CLHT for indexing and PMDK for allocation of key-value pairs. Such a key-value store can be used for in-memory caching to reduce the total cost of ownership (comparing with DRAM-based memcached) and alleviate the impact of failures [82].

In this experiment, we use our four-node machine as the server and the other five machines as clients. Each machine is equipped with a Mellanox ConnectX-6 NIC (200 Gbps); due to the limited bandwidth of PCIe 3.0 × 16, the available bandwidth of the NIC is about 13 GB/s. The key-value size follows the Facebook ETC pool [13, 33]. The key popularity follows a Zipfian distribution with parameter 0.99. We consider a write-intensive workload (50% PUT). Figure 26 shows the throughput with varying clients threads. By using Nap, the throughput is improved by 1.1× under low loads (i.e., 30 client threads) and 1.49× under high loads (i.e., 180 client threads), demonstrating practical benefits of Nap.

6.11 Nap with Snoop-based Coherence

A recent work [51] demonstrates that in some servers, we can configure BIOS settings to change the UPI cache coherence protocol to snoop-based one, so as to mitigate NUMA impacts on PM.

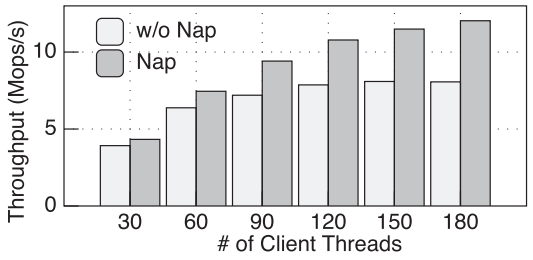

Fig. 26. Throughput of a networked PM-based key-value store (write-intensive, Zipfian 0.99, and 72 threads on the server). *Key-value size follows Facebook ETC workloads.*

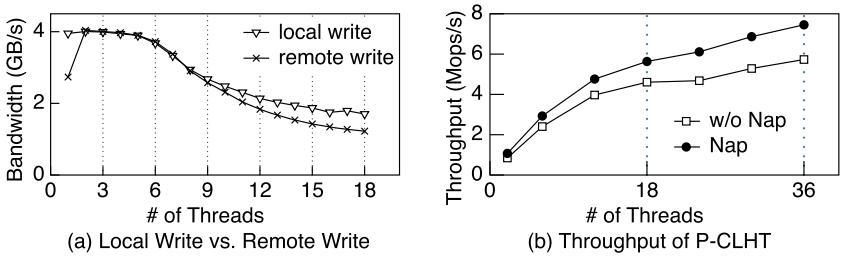

Fig. 27. Nap under snoop-based coherence. ***(a)*** *In case of 18 threads, local access has 1.39× higher write bandwidth than remote access.* ***(b)*** *We use P-CLHT with write-intensive workloads.*

In this experiment, we evaluate Nap under the snoop-based coherence protocol. Since our 4-node server cannot change coherence protocols in BIOS settings, we use another 2-node server. In the 2-node server, each NUMA node is equipped with one Intel Xeon Gold 5220 CPU (18 cores) and two 128 GB Optane DIMMs; UPI is configured to use the snoop-based cache coherence protocol.

Figure 27(a) reports the write bandwidth of local access and remote access in the 2-node server. Each thread performs sequential 32-byte `ntstore` instructions. With snoop-based coherence, NUMA penalty is mitigated significantly (recall Figure 2(a)): remote access can reach peak PM write bandwidth with 2 ∼ 6 threads. Yet, when more threads concurrently access PM, remote access is still much slow than local access. Specifically, in case of 18 threads, local access has 1.39× higher write bandwidth than remote access. Such a result indicates that we also need to take NUMA impacts into consideration when designing a high-performance PM system, even with snoop-based coherence.

Next, we compare P-CLHT with the Nap-converted one under the 2-node server. In case of 36 threads, Nap boosts throughput by 30%. This improvement is lower than the 1 under the directory-based coherence protocol (In Figure 14(e), Nap obtains 1.59× performance gain with 36 threads). This is because snoop-based coherence reduces PM write traffic upon remote access compared with directory-based coherence, decreasing the benefits that Nap can bring.

7 DISCUSSION

Generality of the Nap approach. Even if microarchitectures of hardware (e.g., CPU) evolve and remote PM write can deliver high bandwidth, Nap is still capable of boosting PM indexes under multi-node servers for two reasons. First, since Nap reduces remote accesses significantly, highly concurrent accesses to the same NUMA nodes can be avoided, mitigating contention in the same memory controllers and Optane DIMM XPBuffers; it is well known that such contention degrades the PM performance severely [8, 79]. Second, Nap lowers latency of index operations:

for lookup operations, the GV-view eliminates remote PM reads (400 ns) by using less expensive remote DRAM reads (200 ns); for other operations, per-node PC-views replace remote PM writes with local ones.

Alternative designs. We discuss alternative designs to NUMA-aware PM indexes, and why we do not adopt them.

(1) *Use per-core logs.* In this solution, each thread logs its updates into its local PM node and builds a global DRAM-resident index for lookups. This solution has three issues. First, considering the high bandwidth of PM, using a dedicated core for log recycle is insufficient to digest fast-growing logs; thus, we must use foreground threads or multiple dedicated cores to do this task, which has negative impact on CPU usages and performance. Second, to recycle logs, we must flush items (include hot items) into the underlying PM index, inducing remote accesses. Third, the global DRAM-resident index consumes large DRAM space.

(2) *Abandon NAL switch and maintain per-node PM caches as PC-views.* This solution adopts the architecture of Nap but abandons NAL switch. Instead, it keeps the hot set in per-node PM caches and evicts cold items at the runtime. This solution comes with three drawbacks. First, designing an ideal replacement method is difficult: If we maintain a global hotness-list for cache replacement, the multicore scalability issue happens; if we maintain a hotness-list for each set (set-associative cache), a hot item may be evicted, inducing unnecessary remote accesses. Second, when evicting a cold item from a PM cache (very common events), we must enforce failure atomicity of the cache, yielding extra performance overhead. Third, to guarantee correct lookups and recovery, all items in every PM cache should be presented in the GV-view, which complicates the execution logic. For example, when removing an item from the GV-view, we need to clear corresponding items in all PM caches.

Nap in eADR servers. Newer generations of Intel processors support eADR [9], where the data in the CPU cache will be flushed to PM upon power failure. eADR can eliminate flush instructions, so as to improve the performance of PM indexes. Yet, even with eADR, when lots of threads concurrently access a large-scale PM index, massive data is evicted to PM and these eviction PM writes limit throughput. Hence, reducing remote PM accesses is still important for high-performance PM indexes; thus, we believe Nap can also boost the performance of PM indexes in eADR servers. The benefits of two-incarnation toggle mechanism will drop with eADR, since we do not need to issue flush instructions and fence instructions.

Takeaways. We present our main takeaways from this work.

(1) *A fast NUMA-aware PM index must reduce remote PM accesses without consuming extra local PM bandwidth.* The limited PM bandwidth adds a new dimension to the NUMA problem, which frustrates traditional replication-based approaches designed for DRAM indexes.

(2) *We conjecture that we cannot design a NUMA-aware PM index that is optimal in ❶ minimizing remote PM accesses, ❷ not inducing extra local PM accesses, and ❸ constant DRAM/PM consumption.* Nap achieves a sweet spot by leveraging the characteristics of common skewed workloads: it meets ❷ and ❸, and partially meets ❶ (the remote PM accesses to cold items cannot be reduced).

8 RELATED WORK

PM indexes. A large body of work exists for PM indexes with the ultimate goal of minimizing overheads of failure atomicity and improving concurrency [19, 20, 23, 24, 43, 53–55, 58, 61, 64–67, 74, 80, 83, 84]. Among them, RECIPE [55], Pronto [64], and TIPS [53] propose general conversion

methods. Specifically, RECIPE can convert concurrent DRAM indexes that meet a set of conditions into PM indexes; Pronto persists DRAM data structures via asynchronous semantic logging; TIPS can convert any concurrent DRAM index into PM index with durable linearizability guarantee. To the best of our knowledge, NAP is the first work that addresses NUMA problems of PM indexes.

NUMA problems on PM. Several recent studies observe pronounced NUMA impacts on Optane DIMMs [27, 69, 79]. Xu et al. [77] provide NUMA-aware interfaces to NOVA file system [78], which can set the preferred NUMA node for a file. Wang et al. [75] alleviate the NUMA issues of PM file systems by thread migration. Assise [12], a distributed PM file system, uses on-die DMA engines for remote PM writes, to bypass hardware cache coherence. These approaches for file systems cannot be easily applied to PM indexes, because PM indexes (1) use a set of fixed interfaces, (2) are shared by numerous threads, and (3) generate lots of small-sized writes.

NUMA-aware systems. There has been also work migrating NUMA impacts for DRAM indexes, locks, operating systems, and IO devices. NR [15] replicates data structures and synchronizes replicas between NUMA nodes by a shared log. NrOS [14] improves NR's scalability by allowing multiple shared logs and multiple per-node combiners. HydraList [63] and NUMASK [28] are crafted DRAM indexes that replicate index search layer (exclude index data) across NUMA nodes; compared with NR, these two indexes reduce memory consumption, but increase remote memory accesses due to shared index data. Lots of NUMA-aware locks are proposed [17, 31, 32, 49, 70], and most of them feature a hierarchical structure and try to keep the lock ownership within the same node. Linux automatically migrates data pages across NUMA nodes to reduce remote data access [1]. Besides, Carrefour [29] supports page replication, which can alleviate traffic hotspots and eliminate remote accesses. Mitosis [11] transparently replicates and migrates page-tables across NUMA nodes to accelerate page-table walks. Furthermore, vMitosis [68] considers the NUMA problem of page-table walks in virtualized environments. IOctopus [72] addresses the NUMA effects on IO devices by unifying PCIe functions to a logic one. Different from the above systems, the NUMA-aware PM indexes are unique for the limited PM bandwidth and requirements of failure atomicity.

Hotness-aware systems. Hotspots can be seen everywhere in the real world. There are two lines of work: (1) mitigating the effects of hotspots, and (2) leveraging hotspots to boost system performance. In the aspect of the former, lots of systems mitigate the load imbalance across backend servers by using high-performance caches to handle lookup operations to hot items [35, 44, 56, 59]. In the aspect of the latter, HotRing [18] designs an in-memory hashtable that can move pointers to make hot items be served with fewer memory accesses. Like HotRing, NAP regards hotspots as an opportunity to boost system performance, but targets NUMA-aware PM indexes.

9 CONCLUSION

PM provides memory-level storage, making indexes survive power outages with DRAM-comparable performance. However, PM indexes today do not scale well when spanning multiple NUMA nodes. In this work, we have designed, implemented, and evaluated NAP, a black-box approach that converts concurrent PM indexes into NUMA-aware counterparts. NAP uses a NAL to absorb accesses to hot items, which eliminates remote PM accesses without inducing extra local PM accesses. To cope with constantly changing workloads, NAP incorporates a lightweight hot set identification mechanism and a fast three-phase NAL switch technique. NAP significantly boosts the performance of PM indexes on multi-node machines.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their enlightening feedback.

REFERENCES

[1] 2020. AutoNUMA: The Other Approach to NUMA Scheduling. Retrieved 01 Dec., 2020 from https://lwn.net/Articles/488709/.

[2] 2020. Distributed Reader-Writer Mutex. Retrieved 01 Dec., 2020 from http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex.

[3] 2020. Implementation of P-Masstree and FAST_FAIR. Retrieved 01 Dec., 2020 from https://github.com/utsaslab/RECIPE/.

[4] 2020. Persistent Memory Development Kit. Retrieved 01 Dec., 2020 from https://pmem.io/pmdk/.

[5] 2020. PMDK Implementation of Clevel, CCEH and P-CLHT. Retrieved 01 Dec., 2020 from https://github.com/chenzhangyu/Clevel-Hashing/.

[6] 2020. Processor Counter Monitor (PCM). Retrieved 01 Dec., 2020 from https://github.com/opcm/pcm.

[7] 2020. Sequential Locks. Retrieved 01 Dec., 2020 from https://www.kernel.org/doc/html/latest/locking/seqlock.html.

[8] 2021. Intel 64 and IA-32 Architectures Optimization Reference Manual. Retrieved 15 Oct., 2021 from https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[9] 2021. Intel Optane Persistent Memory 200 Series Brief. Retrieved 15 Oct., 2021 from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[10] 2021. Intel Xeon Processor Scalable Family Technical Overview. Retrieved 15 Oct., 2021 from https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html.

[11] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 283–300. DOI: https://doi.org/10.1145/3373376.3378468

[12] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and availability via client-local NVM in a distributed file system. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 1011–1027. Retrieved from https://www.usenix.org/conference/osdi20/presentation/anderson.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, New York, NY, 53–64. DOI: https://doi.org/10.1145/2254756.2254766

[14] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective replication and sharing in an operating system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 207–221. DOI: https://doi.org/10.1145/3037697.3037721

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 209–223. Retrieved from https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

[17] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, 215–226. DOI: https://doi.org/10.1145/2688500.2688503

[18] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 239–252. Retrieved from https://www.usenix.org/conference/fast20/presentation/chen-jiqiang.

[19] Shimin Chen and Qin Jin. 2015. Persistent B^+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (Feb. 2015), 786–797. DOI: https://doi.org/10.14778/2752939.2752947

[20] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: A persistent B+-Tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 2634–2648. DOI: https://doi.org/10.14778/3407790.3407850

[21] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural*

Support for Programming Languages and Operating Systems. Association for Computing Machinery, New York, NY, 1077–1091. DOI: https://doi.org/10.1145/3373376.3378515

[22] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable persistent memory file system with kernel-userspace collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. USENIX Association, 81–95. Retrieved from https://www.usenix.org/conference/fast21/presentation/chen-youmin.

[23] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX Association, 799–812. Retrieved from https://www.usenix.org/conference/atc20/presentation/chen.

[24] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 441–454. DOI: https://doi.org/10.1145/3297858.3304046

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[26] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55, 1 (April 2005), 58–75. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001

[27] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*. ACM.

[28] Henry Daly, A. Hassan, M. Spear, and R. Palmieri. 2018. NUMASK: High performance scalable skip list for NUMA. In *Proceedings of the 32nd International Symposium on Distributed Computing*.

[29] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 381–394. DOI: https://doi.org/10.1145/2451116.2451157

[30] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 631–644. DOI: https://doi.org/10.1145/2694344.2694359

[31] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, 65–74. DOI: https://doi.org/10.1145/1989493.1989502

[32] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing* 1, 2, (Feb. 2015), 42 pages. DOI: https://doi.org/10.1145/2686884

[33] Diego Didona and Willy Zwaenepoel. 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 79–93.

[34] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 478–493. DOI: https://doi.org/10.1145/3341301.3359637

[35] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, Article 23, 12 pages. DOI: https://doi.org/10.1145/2038916.2038939

[36] Keir Fraser. 2004. *Practical Lock-Freedom*. Ph. D. Dissertation. University of Cambridge, UK. Retrieved from http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193.

[37] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 377–392. DOI: https://doi.org/10.1145/3385412.3386031

[38] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* Association for Computing Machinery, New York, NY, 1218–1232. DOI: https://doi.org/10.1145/3453483.3454105

[39] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 913–928.

[40] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 775–788. DOI: https://doi.org/10.1145/3373376.3378472

[41] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, 355–364. DOI: https://doi.org/10.1145/1810479.1810540

[42] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, 1–7. DOI: https://doi.org/10.1145/2670518.2673882

[43] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. USENIX Association, Oakland, CA, 187–200. Retrieved from https://www.usenix.org/conference/fast18/presentation/hwang.

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 121–136. DOI: https://doi.org/10.1145/3132747.3132764

[45] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 494–508. DOI: https://doi.org/10.1145/3341301.3359631

[46] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, 105–119. DOI: https://doi.org/10.1145/3419111.3421294

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 1–16.

[48] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A scalable ordering primitive for multicore machines. In *Proceedings of the 13th EuroSys Conference*. Association for Computing Machinery, New York, NY, Article 34, 15 pages. DOI: https://doi.org/10.1145/3190508.3190510

[49] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 603–615.

[50] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the design space of page management for multi-tiered memory systems. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 715–728. Retrieved from https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon.

[51] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 424–439. DOI: https://doi.org/10.1145/3477132.3483589

[52] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable transactional memory can scale with timestone. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 335–349. DOI: https://doi.org/10.1145/3373376.3378483

[53] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *Proceedings of the 2021 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association.

[54] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*. USENIX Association, 257–270.

[55] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 462–477. DOI: https://doi.org/10.1145/3341301.3359635

[56] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be fast, cheap and in control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. USENIX Association, 31–44.

[57] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. Association for Computing Machinery, New York, NY, 21–35. DOI: https://doi.org/10.1145/3035918.3064015

[58] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7 (March 2020), 1078–1090. DOI: https://doi.org/10.14778/3384345.3384355

[59] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. USENIX Association, 143–157.

[60] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 10 (April 2020), 1147–1161. DOI: https://doi.org/10.14778/3389133.3389134

[61] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. USENIX Association, 1–16. Retrieved from https://www.usenix.org/conference/fast21/presentation/ma.

[62] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, 183–196. DOI: https://doi.org/10.1145/2168836.2168855

[63] Ajit Mathew and Changwoo Min. 2020. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proceedings of the VLDB Endowment* 13, 9 (May 2020), 1332–1345. DOI: https://doi.org/10.14778/3397230.3397232

[64] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 789–806. DOI: https://doi.org/10.1145/3373376.3378456

[65] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. USENIX Association, Boston, MA, 31–44. Retrieved from https://www.usenix.org/conference/fast19/presentation/nam.

[66] Faisal Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, Dhruva R. Chakrabarti, and M. Scott. 2017. Dalí: A periodically persistent hash map. In *Proceedings of the 31st International Symposium on Distributed Computing*.

[67] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, 371–386. DOI: https://doi.org/10.1145/2882903.2915251

[68] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. Fast local page-tables for virtualized NUMA servers with VMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 194–210. DOI: https://doi.org/10.1145/3445814.3446709

[69] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. Association for Computing Machinery, New York, NY, 304–315. DOI: https://doi.org/10.1145/3357526.3357568

[70] Z. Radovic and E. Hagersten. 2003. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. 241–252. DOI: https://doi.org/10.1109/HPCA.2003.1183542

[71] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. 2020. TH-DPMS: Design and implementation of an RDMA-enabled distributed persistent memory storage system. *ACM Transactions on Storage* 16, 4, (Oct. 2020), 31 pages. DOI: https://doi.org/10.1145/3412852

[72] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. 2020. IOctopus: Outsmarting nonuniform DMA. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 101–115. DOI: https://doi.org/10.1145/3373376.3378509

[73] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, 18–32. DOI: https://doi.org/10.1145/2517349.2522713

[74] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. USENIX Association, 5.

[75] Ying Wang, Dejun Jiang, and Jin Xiong. 2020. NUMA-aware thread migration for high performance NVMM file systems. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*.

[76] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 523–536. Retrieved from https://www.usenix.org/conference/atc21/presentation/wei.

[77] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, 427–439. DOI: https://doi.org/10.1145/3297858.3304077

[78] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 323–338. Retrieved from https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[79] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 169–182. Retrieved from https://www.usenix.org/conference/fast20/presentation/yang.

[80] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, 167–181.

[81] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 191–208. Retrieved from https://www.usenix.org/conference/osdi20/presentation/yang.

[82] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent state machines for recoverable in-memory storage systems with NVRam. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 1029–1046. Retrieved from https://www.usenix.org/conference/osdi20/presentation/zhang-wen.

[83] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4 (Dec. 2019), 421–434. DOI: https://doi.org/10.14778/3372716.3372717

[84] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Carlsbad, CA, 461–476. Retrieved from https://www.usenix.org/conference/osdi18/presentation/zuo.

Received October 2021; revised December 2021; accepted December 2021