Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes

Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/osdi21/presentation/wang-qing

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes

Qing Wang, Youyou Lu*, Junru Li, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

We present Nap, a black-box approach that converts concurrent persistent memory (PM) indexes into 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.

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 [1–16], 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 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 8 threads) experience a bandwidth cliff (§2.1). 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 [17]), 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 [18,19]. 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 paper, we propose Nap (NUMA-Aware Persistent Memory Indexes), 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 [20–24], 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.

*Jiwu Shu and Youyou Lu are the corresponding authors.
{shujw, luyouyou}@tsinghua.edu.cn

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 [25]). 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 are bounded. Our evaluation on a four-node machine running 72 threads shows that, when maintaining 100K hot items in the NAL, NAP uses less than 70MB 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*.

In summary, this paper 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 enable NAP's fast reaction to workloads change.
- Experimental evidence showing the efficiency of NAP.

2 Background and Motivation

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

2.1 NUMA Impacts on PM

PM is a new memory technology that enjoys 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

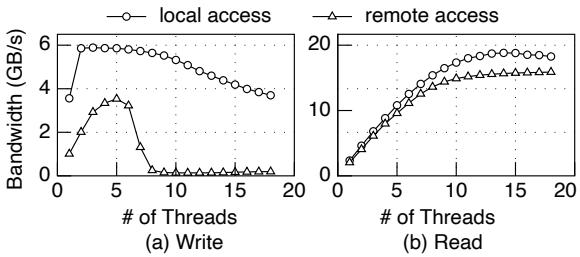

Figure 1: Bandwidth of three 128GB Optane DIMMs with varying threads. ***local access:*** *threads access Optane DIMMs that are local to them;* ***remote access:*** *threads access Optane DIMMs installed on another NUMA node. We use* `ntstore` *instructions for PM write.*

community is actively redesigning storage systems to gain full exploitation of its potential [8, 16, 26–36]. 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).

Figure 1 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 2GB PM space. We use 32-byte non-temporal stores (`ntstore`) for PM write. The peak write bandwidth of remote accesses (3.5GB/s) is only 59% of that of local accesses (5.9GB/s). Worse, the bandwidth of remote write collapses (< 250MB/s) in case of more than 8 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 (~100ns), exacerbating the already high PM read latency (~300ns, [17]). 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 [17, 36–38]. We attribute the low performance of remote PM write to two reasons. First, `nstore` instructions may behave like cache line read-modify-write instructions, reducing the available PM bandwidth [38]. Second, due to the read-modify-write behavior, remote writes may trigger multi-socket cache coherence traffic, which induces extra PM writes [39].

2.2 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 [8, 11, 13, 14, 29, 42]. Here, we investigate the NUMA impacts on PM indexes by analyzing CCEH [9], 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 2(a), when performing

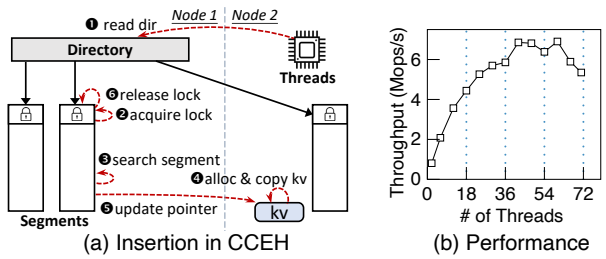

Figure 2: NUMA impacts on PM indexes, using CCEH as an example. *We use source code from [40], which relies on PMDK [41] for PM allocation and supports variable-length keys.* ***(a)*** *An insert operation. Access threads reside on node 2, while the directory and the targeted segment are on node 1. This 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.*

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 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 6240M CPUs (18 cores per CPU), each with three 128GB Optane DIMMs (1.5TB in total). More details of hardware configurations are shown in §6. Figure 2(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 resource, boosting the throughput, but soon, slow PM remote accesses become the key performance determinant, degrading the throughput.

2.3 Limitations of DRAM-orient 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 Node Replication (NR) [18], a state-of-the-art approach that obtains NUMA-aware DRAM indexes. NR maintains a global shared log and per-node replicas of DRAM indexes. Using flat combining [43] within nodes, threads record their operations into the shared 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,

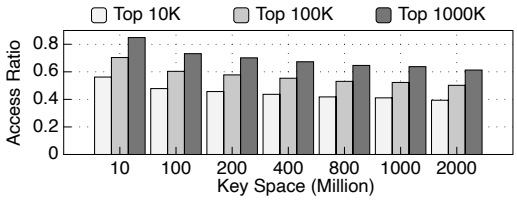

Figure 3: Access ratio of hot items (Zipfian 0.99).

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

3 Key Ideas

1) Making hot accesses NUMA-aware. Real-world workloads often feature Zipfian popularity distribution [20–24], where a small portion of hot items receive extremely frequent accesses. A recent study from Twitter [20] shows that their in-memory cache workloads are usually even more skewed than YCSB [44]. We design NAP to target these skewed workloads by making accesses to hot items NUMA-aware. To show potential benefits of such a design, we run a simulation to present the access ratio of hot items. The key popularity follows Zipfian distribution with parameter 0.99. From Figure 3, we observe that under a wide range of key space (from 10M to 2000M), the top 10K/100K/1000K hottest items receive more than 39%/50%/61% accesses. Hence, if we can absorb accesses to hot items (e.g., top 100K) in a NUMA-aware way, a significant percentage of remote PM accesses are avoided.

NAP introduces a NUMA-aware layer (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 (e.g., 100K), different from replication-based approaches (e.g., NR [18]), 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 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

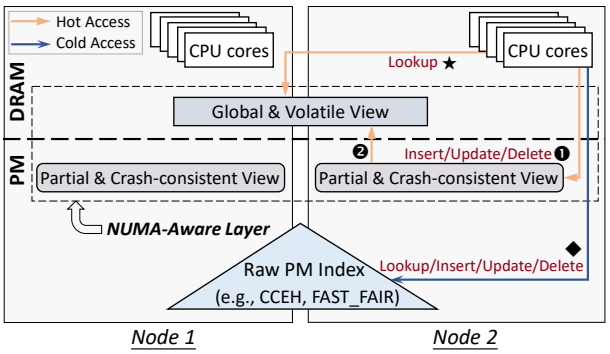

Figure 4: NAP's architecture and interactions.

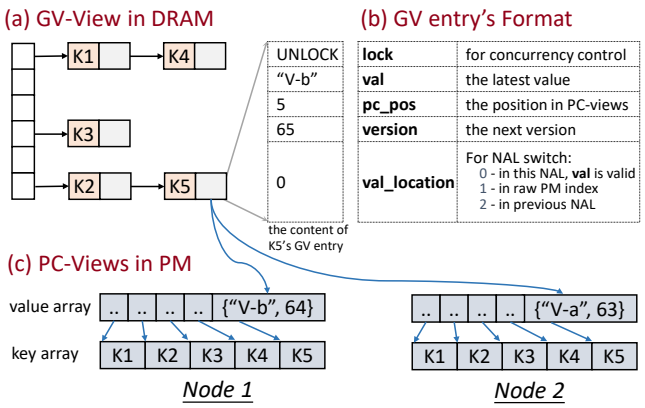

Figure 5: Structures of the GV-view and PC-views.

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 paper proposes NAP, an approach that converts concurrent PM indexes into NUMA-aware ones. Figure 4 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 [9], FAST_FAIR [7]), 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 4), which account for an extremely huge proportion of the total dataset.
- *NUMA-aware layer (NAL)*. NAP steers accesses to hot items to the NAL, which contains two parts: a *global & volatile view* (i.e., GV-view, §4.2) and per-node *partial & crash-consistent views* (i.e., PC-views, §4.3). GV-view resides in DRAM, and maintains the latest values of hot items to serve lookup requests (★ in Figure 4). 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 (§4.4) and switch to a new version of NAL (§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 5(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, K5*} in Figure 5), 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 5(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 (§4.3); 5) an enumerated value that assists in NAL switch (§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 §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 [7] and P-Masstree [8], the range query operations in Nap are not atomic with 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 [45].

4.3 Partial & 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 5(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 5, 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 5). The most significant bit of a version is *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.

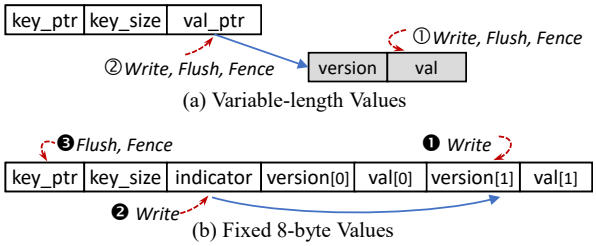

Figure 6: The structure of two types of PC entry. `key_ptr` *points to corresponding key in the key array and* `key_size` *stands for the size of the key.* ***(a)*** *For variable-length values, we use copy-on-write for failure atomicity. Each PC entry is 24-byte. The grey space of* [*version*,*val*] *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.*

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., ⟨*value*, *version*⟩ 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.

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 5(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 6(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 [8] (8-byte value is usually a pointer indicating the location of real data). Figure 6(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 TSO (total store order) architecture of Intel CPUs [7, 10, 28]. 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 [21]; thus, NAP requires to identify the hot set in real time. The design goals of identifying hot set are 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 7 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 ⟨*timestamp*, *key*⟩ pair into the record buffer, where *timestamp* is a 64-bit number generated via `rdtsc` instructions and *key* is the key of current index operation. The access thread blindly appends ⟨*timestamp*, *key*⟩ 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 [25] 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 ⟨*timestamp*, *key*⟩ pairs from it; this batched fetch reduces cache line movements. Two types of ⟨*timestamp*, *key*⟩ pairs are considered invalid: i) 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. ii) (*current time* − *timestamp*) is greater than a threshold value (e.g., 100ms), indicating this pair is too stale; thus, the pair is skipped. Note that although the timestamps generated via `rdstc` are not strictly synchronized between CPU cores [46, 47], 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 [48].
3) The min heap maintains the current hot set in the form of ⟨*key*, *frequency*⟩ 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 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.

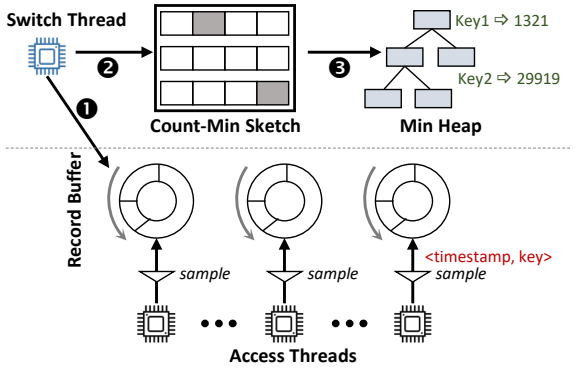

Figure 7: Hot set identification. *Access threads publish their access patterns into record buffers with sampling. The switch thread uses a count-min sketch to estimate frequency of keys and a min heap to maintain the current hot set.*

Periodically (e.g., per 1 second), the switch thread compares the heap with the hot set being used by 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 (§4.5) with the new hot set (i.e., keys in the heap). 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

```
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 }
```

Listing 1: 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`⟩.

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 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 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 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 [49]), to ensure its modifications are visible for all ongoing and future index operations.

Listing 1 shows the procedure of the NAL switch, which consists of three phases:

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 5(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}.

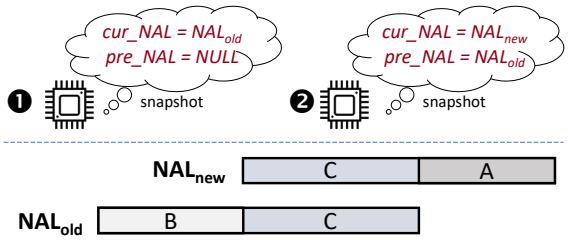

Figure 8: Different access threads see different system states. *A, B and C each stand for an exclusive set of items.*

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 [50]. Before performing an index operation, the access thread saves a snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair under the protection of the seqlock, and accesses NAL according to the snapshot. The seqlock minimizes cache coherence traffic at the reader-side (i.e., access threads).

At this time, the different ongoing index operations may have saved different snapshot of ⟨`cur_NAL`, `pre_NAL`⟩ pair, as shown in Figure 8: type ❶ access threads only see the NAL_{old} and do not realize the concurrent NAL switch; type ❷ access threads see the both NAL_{new} and NAL_{old}. For type ❶ threads, they manipulate NAL_{old} and workflow of index operations is the same as cases of no NAL switch (§4.2 and §4.3). The index operations becomes a bit complicated for type ❷ threads:

i) For an insert/update/delete operation, if the targeted item belongs to NAL_{new}, NAL_{new} absorbs this operation like the case of no NAL switch (§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} (range `B` in Figure 8), 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.

ii) 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 8) 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 ac-

cess 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). 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 due to lazy initialization, 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 8) 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 line 5 and line 12 of Listing 1, the switch thread records undo log entries for atomicity. For line 15, an 8-byte atomic write is enough.

4.6 Recovery

Recovery in Nap is simple. First, we invoke the recovery procedure of the underlying raw PM index. Second, by scanning the undo log and global pointers (i.e., `cur_NAL`, `pre_NAL` and `gc_NAL`), we construct the valid version of these pointers. Third, we flush the PC-views of NALs pointed by `pre_NAL` (if not null) and `cur_NAL` in order; the latest values in PC-views of each NAL are identified by versions (§4.3). Finally, we free the PM space of PC-views in NALs pointed by these three pointers, avoiding the memory leak.

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 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 `cur_NAL` and then the one pointed by `pre_NAL`, so as to ensure only the latest values appear in the raw PM index.

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

5 Implementation

We have implemented NAP in C++ (~2000 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` to organize the GV-view by default; if the underlying raw PM index supports range query, we use C++ `map`.

We leverage PMDK [41] 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 (§4.3), we adopt a simple customized allocator. Each thread requests 1MB 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.

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? (§6.2)
- How does NAP perform when value size is variable? (§6.3)
- How does NAP react to dynamic workloads? (§6.4)
- How do the characteristics of workloads and NUMA configurations affect the performance of NAP? (§6.5)
- How does NAP compare with Node Replication? (§6.6)
- What are the overheads incurred when using NAP? (§6.7)
- What is the benefit of NAP to real applications? (§6.8)

6.1 Experimental Setup

The experiments are conducted on a 4-socket (NUMA node) machine. Each NUMA node is populated with an 18-core Intel Xeon Gold 6240M CPUs, three 128GB Optane DIMMs and three 32GB DDR4 DIMMs, resulting in a machine with 72 CPU cores, 1.5TB PM and 384GB 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 100K, and the switch thread tries to perform the NAL switch per 0.2 seconds. Each per-core record buffer is 300KB. The count-min sketch contains 3 counter arrays, each with 32-bit 850,000 counters. The sampling interval is 32.

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 [44]). 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* [9]. 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* [11]. A lock-free version of level hashing [12], which is organized as two bucket arrays.
- *P-CLHT* [8]. PM version of CLHT [51], which is a linked-list-based hashtable. It supports lock-free lookups and uses bucket-grained locks for other operations.
- *P-Masstree* [8]. PM version of Masstree [52], a trie-like concatenation of B+ tree nodes. It adopts lock-free lookups and lock-based writes.
- *FAST_FAIR* [7]. A PM B+ tree with lock-free lookups and lock-based writes.

For CCEH, Clevel, and P-CLHT, we use the source code from [40], 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 [53], avoiding its scalability issues. For P-Masstree and FAST_FAIR, we use the source code from [54] 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 (§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 9 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 45~54% 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 ex-

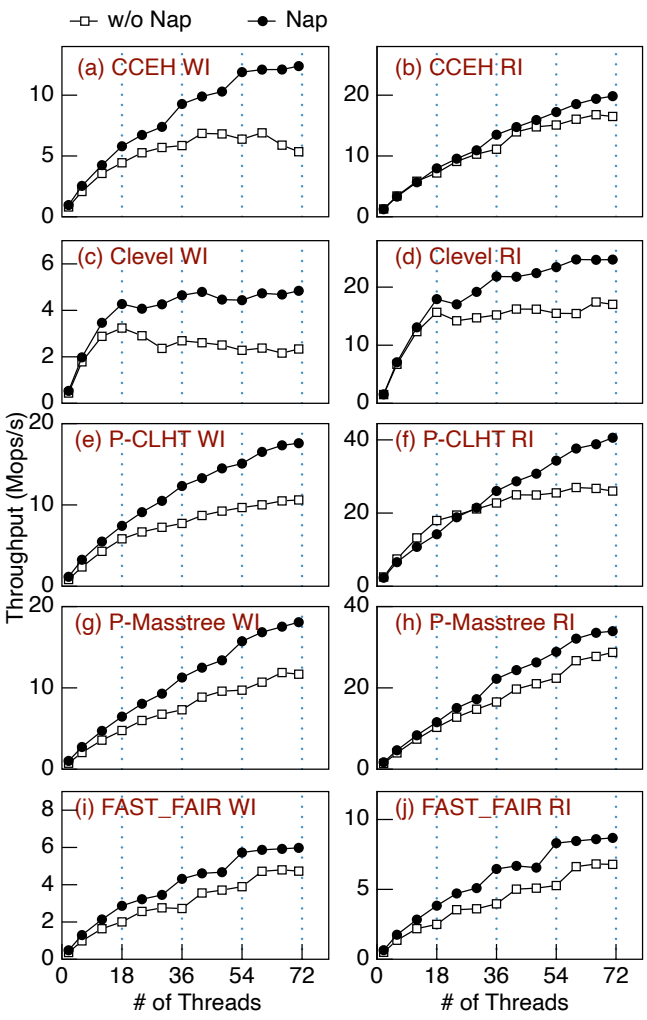

Figure 9: Throughput under write/read-intensive workloads. ***WI****: write-intensive workloads;* ***RI****: read-intensive workloads. Vertical lines show the boundaries between NUMA nodes.*

hibit much higher performance than remote PM accesses: 5.7× higher throughput for writes (20GB/s : 3.5GB/s) and 2× lower latency for reads (200ns : 400ns).

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 9(c) and (d)). These hashtables always use several continuous and large arrays for fast indexing (e.g., the global directory of CCEH, bucket 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.

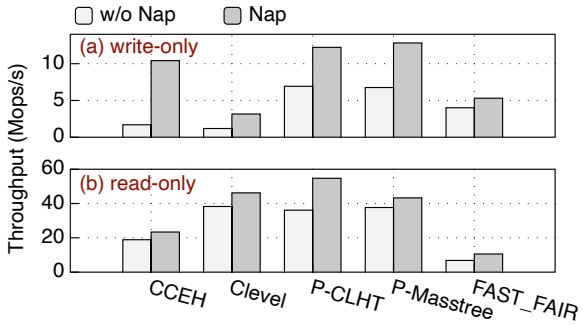

Figure 10: Throughput under write/read-only workloads. *We run 72 threads spanning 4 NUMA nodes.*

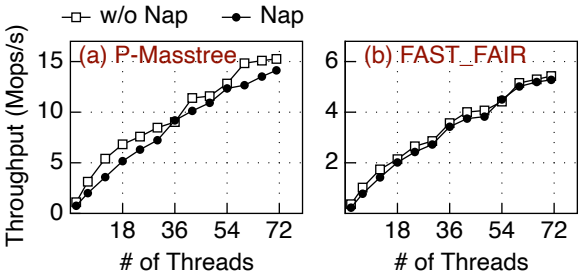

Figure 11: Throughput under scan-intensive workloads.

Throughput under write/read-only workloads. Figure 10 shows the throughput under write-only and read-only workloads. Due to space limitations, we only reports results of 72 threads. 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.

Throughput under scan-intensive workloads. Figure 11 shows the range query performance of P-Masstree and FAST_FAIR. We set the query range to 10. With 72 threads spanning 4 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.

Latency. Figure 12 depicts the latency distribution of P-

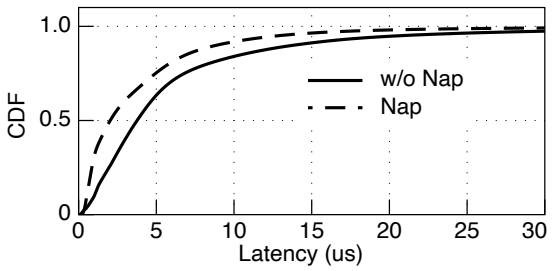

Figure 12: Latency distribution (P-CLHT, 72 threads, write-intensive workloads). *The 50th and 99th latencies of the original index are 3.77µs and 49.95µs (not shown in the figure), respectively. The 50th and 99th latencies of* NAP*-converted index are 2.04µs and 27.64µs, respectively.*

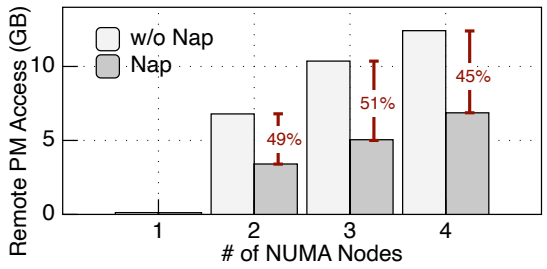

Figure 13: The amount of data via remote PM accesses (P-CLHT, write-intensive workloads). *We run 18, 36, 54, and 72 threads to measure results under different NUMA nodes.*

CLHT under write-intensive workloads. The number of access threads is 72. Due to space limitations, 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 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 1).

Quantitative measurement of remote PM accesses. We use Intel's PCM tools [55] 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 13 reports the result. NAP reduces remote PM accesses by 45% to 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. Figure 14 presents the result, from which we make two observations. First, due to more flush and fence instructions in CoW, NAP's throughput

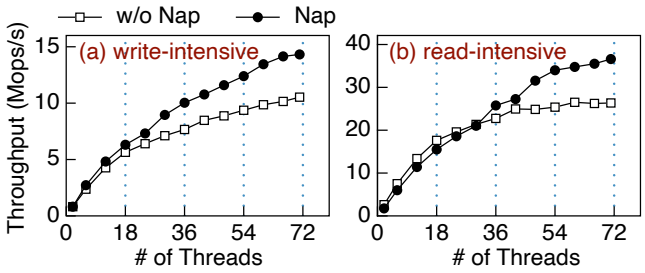

Figure 14: Throughput of P-CLHT. *The value size is randomly selected from 8 bytes to 256 bytes .*

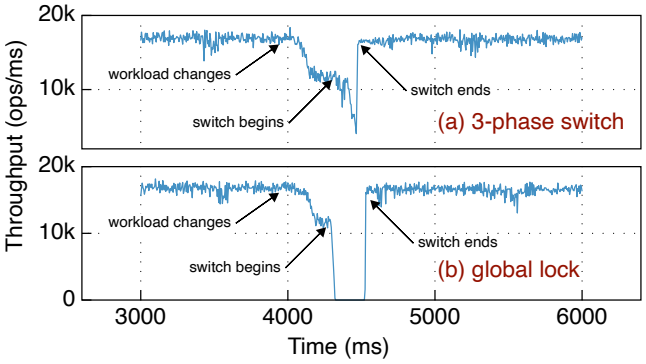

Figure 15: Throughput over time with workloads change (P-Masstree, 71 threads, write-intensive workloads).

degrades (compared with Figure 9(e) and (f)). Second, NAP-converted P-CLHT still 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.

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 [53]. We run NAP-converted P-Masstree under write-intensive workloads with 71 threads (one core is reserved to record total throughput per 5ms). Figure 15 shows the throughput over time. The workload changes at time 4s. Since the NAL can not absorb the accesses to current hot set, the throughput drops. After about 200~300ms, NAP identifies the new hot set (recall that the switch period is 0.2s, §6.1), and triggers the NAL switch. In our three-phase switch, the throughput can be maintained more than 10K ops/ms for about 130ms, then drops to 4K~8K ops/ms for about 35ms. This is because the three-phase switch only blocks some insert/update operations to a part of old NAL

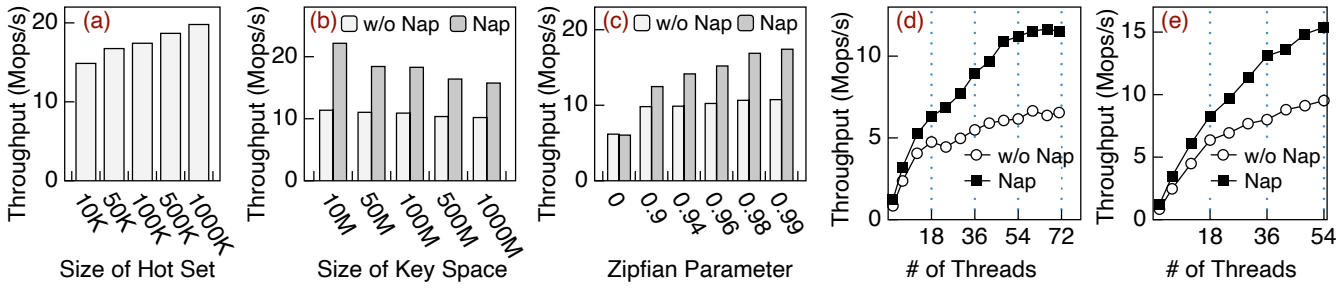

Figure 16: Sensitivity Analysis (P-CLTH, write-intensive). ***(a)*** *Varying the size of hot set (72 threads).* ***(b)*** *Varying the size of key space (72 threads).* ***(c)*** *Varying the Zipfian parameter (72 threads).* ***(d)*** *two Optane DIMMs per NUMA node.* ***(e)*** *four Optane DIMMs per NUMA node.*

during phase 2. However, when using the global lock, the system is unavailable (i.e., throughput is 0) for about 195ms. To sum up, NAP is robust enough to react to dynamic workloads quickly without sacrificing availability.

6.5 Sensitivity Analysis

Size of hot set. Figure 16(a) shows how the configured hot set size affects the NAP's performance. As the size of hot set increases from 10K to 1M, 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.

Size of key space. Figure 16(b) 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., 1000 million, NAP can boost the throughput by 1.55×, which demonstrates that NAP can handle large-scale workloads.

Skewness of workloads. Figure 16(c) shows how the skewness of workloads affects NAP's performance. 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 almost the same under uniform workloads. This is because NAP handles uniform workloads by initializing an empty NAL, which induces negligible overhead.

Different NUMA configurations. Here, we change NUMA configurations by adding/removing Optane DIMMs, and show how the available PM bandwidth affects NAP. We get two new NUMA configurations: i) 2 Optane DIMMs per node; ii) 4 Optane DIMMs per node (only 3 nodes due to the total of 12 DIMMs). Figure 16(d) and (e) show the results of i) and ii), 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 9(e)). Under 4-DIMMs-per-node configuration, NAP outperforms the original index by 1.62×. Overall, NAP is efficient under different NUMA configurations.

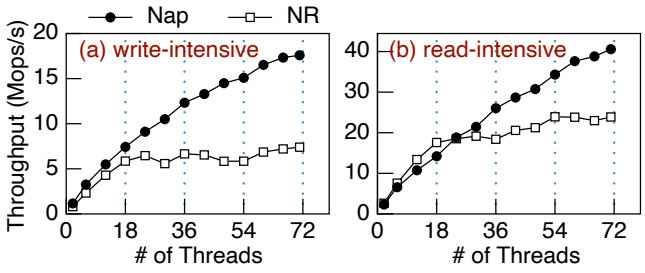

Figure 17: Performance of NAP and NR (P-CLHT).

6.6 Comparison with NR

We compare NAP with Node Replication (NR) [18], 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 17 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 combin-

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

Table 1: Consumption of DRAM and PM in Nap. *We ignore some very small usage, such as the 64-byte persistent undo log used by the switch thread.*

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

Table 2: Recovery time.

ing [43] (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 (§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 thrashing); thus, like Nap, a NUMA-aware PM index should reduce remote PM accesses without consuming extra local PM bandwidth.

6.7 Overheads of Nap

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

Memory consumption. Table 1 shows the memory consumption by Nap in our evaluation (4 NUMA nodes and 72 threads), and the total memory consumption is less than 70MB. 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 2 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 one second.

6.8 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 [56] 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 [28].

In this experiment, we use our four-node machine as the server and the other 5 machines as clients. Each machine is equipped with a Mellanox ConnectX-6 NIC (200Gbps); due to the limited bandwidth of PCIe 3.0×16, the available bandwidth of the NIC is about 13GB/s. The key-value size follows the Facebook ETC pool [23, 57]. The key popularity follows a Zipfian distribution with parameter 0.99. We consider a write-intensive workload (50% PUT). Figure 18 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.

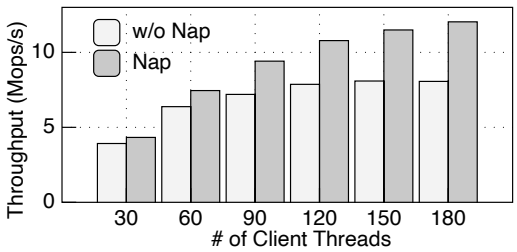

Figure 18: Throughput of a networked PM-based key-value store (write-intensive, Zipfian 0.99, 72 threads on the server). *Key-value size follows Facebook ETC workloads.*

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 [17, 39]. Second, Nap lowers latency of index operations: for lookup operations, the GV-view eliminates remote PM reads (400ns) by using less expensive remote DRAM reads (200ns); 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.

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 [1–16, 29, 58]. Among them, RECIPE [8], Pronto [29] and TIPS [58] 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 [17, 37, 38]. Xu et al. [59] provide NUMA-aware interfaces to NOVA file system [60], which can set the preferred NUMA node for a file. Wang et al. [61] alleviate the NUMA issues of PM file systems by thread migration. Assise [27], 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 [18] replicates data structures and synchronizes replicas between NUMA nodes by a shared log. NrOS [62] improves NR's scalability by allowing multiple shared logs and multiple per-node combiners. HydraList [19] and NUMASK [63] 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 [64–68], 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 [69]. Besides, Carrefour [70] supports page replication, which can alleviate traffic hotspots and eliminate remote accesses. Further, Mitosis [71] transparently replicates and migrates page-tables across NUMA nodes to accelerate page-table walks. 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 back-end servers by using high-performance caches to handle lookup operations to hot items [48, 73–75]. In the aspect of the latter, HotRing [21] 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

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 NUMA-aware layer to absorb accesses to hot items, which eliminates remote PM accesses without inducing extra local PM accesses. NAP significantly boosts the performance of PM indexes on multi-node machines.

Acknowledgements

We sincerely thank our shepherd Changwoo Min for helping us improve the paper. We also thank the anonymous reviewers for their feedback. 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).

References

[1] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[2] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 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*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Shimin Chen and Qin Jin. Persistent B$^+$-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[4] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 167–181, USA, 2015. USENIX Association.

[5] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 257–270, USA, 2017. USENIX Association.

[6] Faisal Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, Dhruva R. Chakrabarti, and M. Scott. Dalí: A Periodically Persistent Hash Map. In *DISC*, 2017.

[7] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[8] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[10] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 441–454, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[12] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.

[13] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.*, 13(4):421–434, December 2019.

[14] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.*, 13(7):1078–1090, March 2020.

[15] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[16] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16. USENIX Association, February 2021.

[17] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[18] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the*

Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Ajit Mathew and Changwoo Min. HydraList: A Scalable in-Memory Index Using Asynchronous Updates and Partial Replication. *Proc. VLDB Endow.*, 13(9):1332–1345, May 2020.

[20] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[21] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[22] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, page 1–7, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Graham Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms*, 55(1):58–75, April 2005.

[26] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.

[27] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[28] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1029–1046. USENIX Association, November 2020.

[29] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), October 2020.

[34] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with

timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95. USENIX Association, February 2021.

[37] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304–315, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China*, SIGMOD '21. ACM, 2021.

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

[40] PMDK Implementation of Clevel, CCEH and P-CLHT. https://github.com/chenzhangyu/Clevel-Hashing/, 2020.

[41] Persistent Memory Development Kit. https://pmem.io/pmdk/, 2020.

[42] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.

[43] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[44] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[46] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, UK, 2004.

[50] Sequential Locks. https://www.kernel.org/doc/html/latest/locking/seqlock.html, 2020.

[51] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[52] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

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

[54] Implementation of P-Masstree and FAST_FAIR. https://github.com/utsaslab/RECIPE/, 2020.

[55] Processor Counter Monitor (PCM). https://github.com/opcm/pcm, 2020.

[56] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs Can Be General and Fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[57] Diego Didona and Willy Zwaenepoel. 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*, NSDI'19, page 79–93, USA, 2019. USENIX Association.

[58] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *Proceedings of the 2021 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '21, USA, 2021. USENIX Association.

[59] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[61] Ying Wang, Dejun Jiang, and Jin Xiong. NUMA-Aware Thread Migration for High Performance NVMM File Systems. In *36th International Conference on Massive Storage Systems and Technology*, MSST '20, 2020.

[62] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective Replication and Sharing in an Operating System. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021.

[63] Henry Daly, A. Hassan, M. Spear, and R. Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In *DISC*, 2018.

[64] Z. Radovic and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 241–252, 2003.

[65] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[66] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 215–226, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-Aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 603–615, USA, 2017. USENIX Association.

[68] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[69] AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/, 2020.

[70] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[71] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. IOctopus: Outsmarting Nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 101–115, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[74] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 31–44, USA, 2016. USENIX Association.

[75] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 143–157, USA, 2019. USENIX Association.