NStore: A High-Performance NUMA-Aware Key-Value Store for Hybrid Memory

Zhonghua Wang, Kai Lu, Jiguang Wan, Hong Jiang, *Fellow, IEEE*, Zeyang Zhao, Peng Xu, Biliang Lai, Guokuan Li, and Changsheng Xie, *Member, IEEE*

***Abstract*—Emerging persistent memory (PM) promises near-DRAM performance, larger capacity, and data persistence, attracting researchers to design PM-based key-value stores. However, existing PM-based key-value stores lack awareness of the Non-Uniform Memory Access (NUMA) architecture on PM, where accessing PM on remote NUMA sockets is considerably slower than accessing local PM. This NUMA-unawareness results in sub-optimal performance when scaling on NUMA. Although DRAM caching alleviates this issue, existing cache policies ignore the performance disparity between remote and local PM accesses, keeping remote PM access as a performance bottleneck when scaling PM stores on NUMA. Furthermore, creating hot data views in each socket's PM fails to eliminate remote PM writes and, worse, induces additional local PM writes. This paper presents NStore, a high-performance NUMA-aware key-value store for the PM-DRAM hybrid memory. NStore introduces a NUMA-aware cache replacement strategy, called Remote Access First (RAF) cache in DRAM, to minimize remote PM accesses. In addition, NStore deploys Nlog, a write-optimized log-structured persistent storage, purposed to eliminate remote PM writes. NStore further mitigates the NUMA impacts through localized scan operations, efficient garbage collection, and multi-thread recovery for Nlog. Evaluations show that NStore outperforms state-of-the-art PM-based key-value stores, achieving up to 13.9× and 11.2× higher write and read throughput, respectively.**

***Index Terms*—Key-value store, persistent index, persistent memory, log structure, NUMA-aware index.**

Received 30 August 2023; revised 5 January 2024; accepted 30 March 2024. Date of publication 21 November 2024; date of current version 12 February 2025. This work was supported by the National Key Research and Development Program of China under Grant 2023YFB4502701, by the National Natural Science Foundation of China under Grant 62072196, by the Key Research and Development Program of Guangdong Province under Grant 2021B0101400003, by the Creative Research Group Project of NSFC under Grant 61821003, and by the National Natural Science Foundation of China under Grant 62302465. Recommended for acceptance by D. Liu. *(Corresponding author: Kai Lu.)*

Zhonghua Wang, Kai Lu, Jiguang Wan, Zeyang Zhao, Biliang Lai, Guokuan Li, and Changsheng Xie are with Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: wangzhonghua@hust.edu.cn; kailu@hust.edu.cn; jgwan@hust.edu.cn; vctzzy@hust.edu.cn; lbl@hust.edu.cn; liguokuan@hust.edu.cn; cs_xie@hust.edu.cn).

Hong Jiang is with the University of Texas at Arlington, Arlington, TX 76019 USA (e-mail: hong.jiang@uta.edu).

Peng Xu is with the Research Center for Graph Computing, Zhejiang Laboratory, Hangzhou 311100, China (e-mail: xup@zhejianglab.com).

Digital Object Identifier 10.1109/TC.2024.3504269

I. Introduction

PERSISTENT memory (**PM**) technologies, such as Phase Change Memory (PCM) [1] and Spin-Transfer Torque RAM (STT-RAM) [2], offer large capacity, low latency, low power consumption, byte-addressability, and non-volatility [3], which have drawn significant attention from both academia and industry. Recently, lots of researchers have tried to deploy key-value stores on PM (called PM-based key-value stores or PM KV stores) [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, existing PM KV stores, whether using log structures, B+-trees or learned indexes, rarely consider the impact of non-uniform memory access (**NUMA**) architectures. How to alleviate or eliminate the NUMA effect is still an unsolved problem [15].

It is well known that the NUMA architecture is designed to enhance computing scalability [16], particularly in terms of supporting a higher number of CPU cores and threads. In a NUMA architecture, CPU cores and memory DIMMs are grouped into nodes (called NUMA nodes or **sockets**). These sockets are interconnected by high-speed inter-node links, such as Intel Ultra Path Interconnect. Compared to a single socket with limited DIMM slots and cores, NUMA architectures provide enormous cores as well as massive bandwidth and capacity of PM. However, the performance of a CPU core accessing PM in other sockets (**remote PM access**) is much worse than accessing the PM in the same socket (**local PM access**). Our evaluations (Section II-A1) show that the peak bandwidth of remote PM writes and reads is only 51% and 80% of that of local PM writes and reads, respectively. Worse, remote PM writes and reads experience bandwidth meltdowns when the thread count is high. Besides, the average latency of remote PM writes and reads is 1.6× and 1.5× higher than that of local PM writes and reads, respectively.

This asymmetric PM access performance between NUMA socket local and remote CPU cores poses a challenge for PM KV stores to efficiently store and retrieve data in PMs belonging to different NUMA sockets. Specifically, we observe that *the performance of existing PM KV stores scale sub-optimally with the number of CPU threads in NUMA architectures*. Taking Viper [4], a hash-based log-structure PM-DRAM KV store, APEX [11], a state-of-the-art hybrid PM-DRAM learned index, and FAST+FAIR [10], a typical PM-only B+-tree, as examples, our test results (Section II-A2) show that the trend in their throughput changes significantly after introducing more

CPU threads from remote NUMA sockets. All three approaches perform well within a single socket. However, as we scale them by adding threads from socket 2, the growth rate of put throughput drops significantly, and the get throughput also experiences a decrease. These observations indicate that minimizing remote PM accesses is crucial to the design of PM KV stores for NUMA architectures.

After comprehensively surveying existing PM KV stores and NUMA optimization efforts, we observe that existing PM KV stores are sub-optimal for NUMA (Section VI). The state-of-the-art NUMA-aware approach for PM KV stores (e.g. Nap [17]) reduces remote PM reads and writes by caching hot data set to DRAM and creates the views of hot data set in PM per socket, respectively. However, these strategies have obvious limitations: First, the existing cache replacement policy focuses only on the access frequency and ignores the impact of NUMA on PM accesses. Thus, the proportion of remote PM accesses is still very high. The test results in Fig. 3(a) show that the number of remote PM reads accounted for more than 25% and 30% of all read requests at Zipfian 0.99 and 0.9, respectively. Second, creating a local view for hot data in each socket's PM can only reduce remote PM writes for update/delete operations of hot data. Moreover, the creation of per-socket views during each hot data switch incurs massive extra local PM writes and consumes a lot of local PM bandwidth. Our experimental results in Fig. 3(b) show that there are not only a considerable number of remote PM writes but also a large number of extra local PM writes.

In this paper, we propose NStore, a high-performance NUMA-aware Key-Value Store for hybrid PM-DRAM memory. The key idea behind NStore is to improve both read and write performance by avoiding remote PM access as much as possible in the NUMA architectures. For this end, NStore presents two optimized designs: a novel NUMA-aware cache replacement strategy, called Remote Access First (RAF), to minimize remote PM reads, and a write-optimized log-structured persistent storage (called Nlog) that eliminates remote PM writes under various workloads without incurring extra local PM writes.

First, RAF cache grants higher cache priority to remote accesses because remote PM accesses are more expensive than local PM accesses. The goal of RAF cache is to reduce the cache misses directed to the remote PM, thereby maximizing the performance improvement of the cache on NUMA-based persistent indexes. Second, unlike traditional memory log structures that are either DRAM-based (e.g., Nibble [18] cannot guarantee data consistency) or are not NUMA-friendly (e.g., FASTER [19] has a lot of remote writes in the critical path), Nlog is designed for NUMA with PMs. Nlog independently creates local Nlog files in each socket. All the writes can be appended to the latest local Nlog files with a single local PM write. In addition, we propose three strategies to improve the performance of Nlog, including lightweight and efficient garbage collection with data consistency, localized scan operations, and multi-threaded recovery.

We evaluate NStore on a real PM device and compare it to the seven available state-of-the-art PM systems under various workloads. The test results demonstrate that NStore achieves 2.2×-13.9× higher write throughput and 1.4×-11.2× higher read throughput than existing persistent indexes. In addition, compared to Nap, the update and get throughput of NStore is 15× and 20× higher. Finally, NStore delivers better scalability with an increasing number of sockets compared to existing PM KV stores. It also provides acceptable scan performance and fast recovery. In summary, the main contributions of this work are as follows.

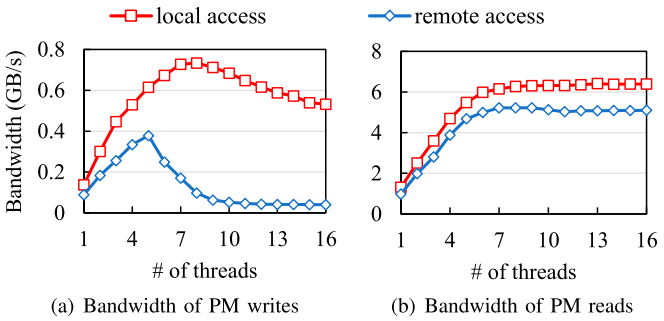
- We conduct a series of experiments to show the impact of the NUMA architecture on PM and existing PM KV stores and analyze the limitations of the state-of-the-art NUMA-aware approaches.
- We propose NStore, an efficient persistent KV store for NUMA architectures. By a novel NUMA-aware cache replacement strategy (RAF) and a write-optimized log-structured persistent storage (Nlog), NStore achieves excellent read and write performance. To the best of our knowledge, RAF is the first cache replacement strategy designed for a NUMA-aware PM index.
- We evaluate NStore with state-of-the-art PM KV stores on Optane DCPMM. The evaluation results show that NStore achieves write throughput that is 2.2×-13.9× higher in NUMA architectures. It also achieves a 1.4×-11.2× higher read throughput. The source code is available at https://github.com/PDS-Lab/NStore.

II. Background and Motivation

A. NUMA Effect on PM and Existing PM KV Stores

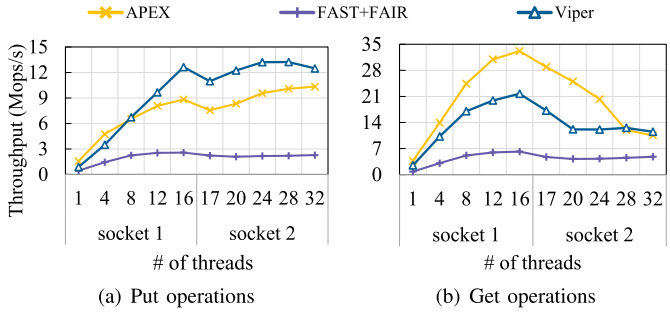
Persistent Memory (PM), or Non-volatile memory (NVM), not only offers persistence, byte-addressability, low power consumption, and a much lower price than DRAM, but also provides sub-microsecond access latency through a directly connected memory bus and load/store instructions [3]. With the release of the first PM product Optane DIMMs, a large number of researchers are vigorously exploring its possibilities on key-value store or mainstream index structures [4], [5], [6], [7], [8], [9], [10], [11], [12], [14]. Although Intel Optane DCPMM, was discontinued last year [20], PM technologies continue to progress [21]. We use Optane to evaluate the efficacy of our work because it is the only commercially available PM product today, but this work is not specific to Optane. Our solution is suitable for most PM devices, which are byte-addressable and have slightly lower read/write performance than DRAM.

The NUMA architecture is widely used in high-performance computing and large-scale server [16], providing enormous bandwidth, capacity, and computing power compared to a single CPU. All the computing and storage resources in NUMA architectures are grouped in different sockets (i.e., NUMA nodes). These sockets communicate internally via Integrated Memory Controller (IMC) bus and are interconnected by high-speed inter-node links, such as Intel Ultra Path Interconnect (UPI). While the NUMA architecture seems ideal as a platform for PM KV stores to enhance their data volume and performance, the existing PM KV stores fail to run efficiently in NUMA architectures for two reasons. First, the NUMA effect (i.e., remote access performance is lower than local access) on PM is severe (Section II-A1). Second, the existing PM



(a) Bandwidth of PM writes (b) Bandwidth of PM reads

Fig. 1. Effect of NUMA on Optane DIMMs. *For each run, we perform sequential accesses to a 2 GB PM space, which is the same trend under random access (not shown).* ***local access****: threads access DIMMs that are local to them;* ***remote access****: threads access DIMMs from another socket; We use* `clflush` *and* `sfence` *instructions for PM writes.*



(a) Put operations (b) Get operations

Fig. 2. Effect of NUMA on PM KV stores, using APEX, FAST+FAIR, and Viper as examples. *For each run in (a), we put 200M (million) KVs in stores; and for each run in (b), we first put 200M KVs, then perform 10M get operations. Threads 1-16 from socket 1, and 17-32 from socket 2.*

KV stores are designed in the oblivion of NUMA effects (Section II-A2).

1) NUMA Effects on PM: We test the local/remote bandwidth and latency of Optane DIMMs in a NUMA architecture (four 128 GB modules and two 16-core CPUs evenly distributed across two sockets) to explore the performance impact of NUMA on PM. All threads were physical to prevent interference from hyper-threading.

Fig. 1(a) and 1(b) show the bandwidth of PM writes and reads (access size 32B), respectively. The peak bandwidth of remote writes is 0.37 GB/s, only 51% of that of local writes (0.73 GB/s). Even worse, the remote write bandwidth collapses when the number of threads exceeds 5. For the PM reads, the gap between local bandwidth (6.38 GB/s) and remote bandwidth (5.09 GB/s) is relatively smaller, but there is still a gap of 20%. Similar results are manifested in latency: remote accesses have significantly higher latency than local accesses. In our test, when the access size (i.e., IO size) is 32 B, the latency of remote writes is 496.83 ns, 1.6× higher than that of local writes (310.52 ns); and the latency of remote reads is 180.65 ns, 1.5× higher than that of local reads (113.4 ns).

We attribute the low performance of remote PM accesses to two reasons. The first and main reason is that inter-socket accesses require updating the coherence information, which introduces additional PM writes and thus causes bandwidth crashes for remote access at high thread counts [22], [23]. The second reason is that UPI has a higher latency than IMC bus [24]. Thus, taking NUMA effects into large-scale PM designs is important for seeking better performance.

2) NUMA Effects on PM KV Stores: We analyze the effect of NUMA on PM KV stores by taking Viper [4], APEX [11], and FAST+FAIR [10] as examples. Viper is a PM-DRAM KV store that places a hash table in DRAM and a log-structured storage in PM. APEX is a state-of-the-art hybrid PM-DRAM learned index that persists all nodes in PM and places the information, such as fingerprints of keys, in DRAM. FAST+FAIR is a typical PM-only B+-tree optimized for low crash consistency overhead and high concurrency performance.

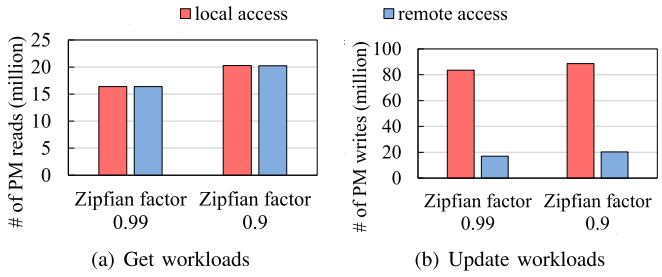
Fig. 2(a) and 2(b) show the put throughput and get throughput of Viper, APEX, and FAST+FAIR as the number of sockets increases, respectively (details of experiment configurations in Section V). They scale well within a single socket. However, the growth rate of put throughput drops significantly, and the get throughput decreases when socket 2 is added. Furthermore, increasing the number of threads from 17 to 32 results in a significant decrease in get throughput for Viper and APEX. The main reason is that the NUMA architecture brings a lot of cross-socket PM accesses, while the existing PM KV stores are only optimized for single sockets without considering the NUMA problem.

Based on the above experiments and analysis, allocating more CPU threads for existing PM KV stores does not lead to the expected higher index throughputs due to the NUMA effect (i.e., a lot of low-performance remote PM accesses). Therefore, the key to designing a high-performance NUMA-aware PM KV store is eliminating remote PM accesses as much as possible.

B. Limitations of Existing NUMA-Aware Approach for PM KV Stores

To address the NUMA effect on PM KV stores, Wang et al. propose Nap [17], a state-of-the-art general-purpose NUMA-aware approach that can be applied to PM KV stores. Nap migrates the latest top-k hot data from PM KV stores into the DRAM cache and per-socket PM views; and writes the old hot data back to the PM KV stores. By caching hot data into the DRAM cache, Nap avoids PM reads to hot data. In addition, by buffering the latest hot data to the PM views of each socket, Nap avoids remote PM writes to hot data. However, Nap has obvious limitations from both a read optimization and a write optimization perspective.

From a read optimization perspective, Nap's cache replacement algorithm does not consider the NUMA effect. The cache replacement algorithm of Nap treats the number of accesses to a key-value pair (KV) as its access frequency and then selects the top-k KVs as hot data in order of access frequency, assuming that the overhead of each access is the same. However, in NUMA architectures, the bandwidth and latency of remote accesses are far worse than those of local accesses (Section II-A1), so the penalty of a miss for remote access is more expensive than that for local access. Fig. 3(a) shows the number of local and remote PM reads in Nap under skewed get workloads with Zipfian of 0.99 and 0.9. The number of local PM reads and remote PM reads is always equal, which



(a) Get workloads (b) Update workloads

Fig. 3. Number of local and remote PM accesses in Nap, with FAST+FAIR as the raw PM index of Nap (called Nap-FF, see Section V). *For each run, we first put 16M KVs, then performs 64M update/get operations from workloads, and finally count the number of local and remote update/get KVs in PM.*

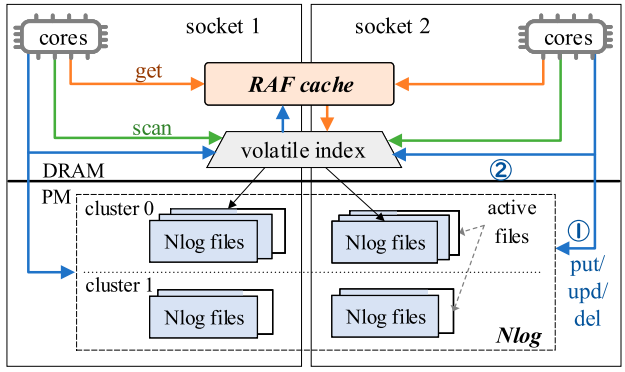


Fig. 4. Overall architecture and interactions of NStore.

indicates that Nap ignores the asymmetry between local and remote accesses. Furthermore, even when the Zipfian reaches 0.99, there are still 16.3M remote PM reads, accounting for 25% of the total number of operations (64M). High-latency remote access lowers overall read performance.

From a write optimization perspective, Nap not only does not eliminate remote PM writes, but also introduces additional local PM writes. First, Nap reduces remote PM writes to hot data by buffering hot data in the local PM view. Therefore, it is only effective for update and delete operations on hot data in skewed workloads, and has little benefit in other cases. Fig. 3(b) shows the number of local and remote PM writes in Nap under skewed update workloads with Zipfian 0.99 and 0.9. There are still many remote PM writes in Nap, and the number of remote PM writes increases as the skewness decreases. When the Zipfian is 0.99, the number of remote PM writes is 16.9M (million), and when the Zipfian is 0.9, the number of remote writes increases to 20.25M. Second, Nap creates the hot data views in per-socket PMs using local threads during each hot data switch, which causes a lot of additional local PM writes. In Fig. 3(b), the number of local PM writes in Nap is very large. At a Zipfian of 0.9, the number of local PM writes is 88.55M, much higher than the total number of update operations in the workload (64M). These remote writes and extra local writes consume the limited bandwidth of the PM, limiting the write performance improvement of Nap on PM indexes. Third, Nap does not separate read and write operations when identifying hot data, so a large amount of hot data from read operations may be meaninglessly migrated to PM in all sockets and written back to raw PM indexes, wasting the write bandwidth of PM.

The above experiments show that the existing NUMA-aware work for PM KV stores has significant limitations in reducing the number of remote PM accesses. This clearly calls for a high-performance NUMA-aware PM KV store.

III. NStore Design

A. Overview

To address these limitations, we present NStore, a high-performance NUMA-aware key-value store for hybrid PM-DRAM memory. Fig. 4 illustrates the architecture of NStore, which consists of three components: a NUMA-aware cache with a remote access first (RAF) replacement strategy to minimize remote PM reads, a persistent storage (Nlog) for localized PM writes, and a volatile index for fast indexing of Nlog. Among them, Nlog and RAF cache are at the heart of NStore.

- ***RAF cache.*** RAF cache is a DRAM-oriented cache for the NUMA-aware PM index, covering multiple sockets. The key advantage of the RAF cache over existing approaches lies in the novel NUMA-aware RAF cache replacement strategy, which aims to minimize remote PM reads in PM indexes by giving higher cache priority to remote-accessed data (Section III-B). The RAF strategy can directly replace any existing cache replacement algorithms (such as LRU, LFU) to invigorate them in the NUMA architecture. The RAF cache uses a hash-table-based structure for fast access and is primarily designed to improve performance for read-intensive skewed workloads. Therefore, we turn off the RAF cache to reduce overhead after write-intensive workloads or uniform workloads are detected.
- ***Nlog.*** Nlog is a NUMA-aware log-structured storage located in PM. Nlog creates local Nlog files in each socket and restricts all write operations only to be appended to the latest local Nlog file (called active file), so each write operation can be persisted with a single local PM write. Further, to prevent write operations from being blocked by the single one active file in a socket, we cluster the data based on the hash function and create a local active file for each cluster in each socket to maximize the write concurrency (Section III-C). In addition to remote-free writes, Nlog conducts other optimizations to revitalize the log-structured design in NUMA architecture. First, Nlog designs a NUMA-aware garbage collection scheme that efficiently reclaims free PM space using a unique *h_addr* field (Section III-D and Section III-F). Second, Nlog delivers a local PM read scheme for scan operations to eliminate remote reads and exploit the intrinsic parallelism of PM (Section III-E). Third, Nlog implements multi-threaded recovery based on the clusters to speed up recovery (Section III-G).
- Volatile index. The volatile index stores all the keys and the addresses of log entries. It can be an arbitrary existing concurrent rang index for DRAM, regardless of its

TABLE I
VARIABLES AND DEFINITIONS

Symbols	Definition
S_i	Socket (NUMA node) i
C_j	Item j
L_i	The set of latencies of accessing PMs in S_i from every socket
l_{i-x}	Latency of accessing to PM of S_i from S_x
F_j	The set of access frequency of C_j from every socket
f_{j-y}	Access frequency of C_j from S_y
TF_j	Total access frequency of C_j in RAF
n	Total number of sockets (NUMA nodes)

structure (tree-based or learned-model-based). Learned-model-based indexes are preferred as the default choice because they usually have good performance and a small space footprint. We deploy the volatile index across multiple sockets because remote DRAM access is faster than local PM access [3] (in other words, remote DRAM access is not a performance bottleneck for persistent indexes).

Fig. 4 also shows the workflow and interaction among key components of NStore. For the get operation, if the RAF cache is disabled, the get thread directly searches for the targeted key in the volatile index. Otherwise, the get thread checks the RAF cache for the searched key and returns if it exists; if the searched key does not exist, the get thread is redirected to the volatile index; For the scan operation, the scan thread searches for targeted keys in the volatile index, regardless of whether the RAF cache is enabled. For the write operation (including put, update and delete), the write thread first persists the action to Nlog and then updates the volatile index. Finally, if the RAF cache is enabled, the write thread checks for the updated/deleted key in the RAF cache and updates it if it exists. Since volatile index can use the existing structure, this paper only describes in detail the workflow of read and write operations in Nlog (Section III-E).

B. RAF Cache

To scale the performance of PM KV stores on NUMA architectures, the state-of-the-art work uses a buffer cache in DRAM to reduce PM reads [17], where the cache replacement algorithm selects cached items (i.e., key-value pairs) based on the access frequency (i.e., the number of accesses over a period of time) of items. Such a replacement algorithm assumes that the penalty of each miss is the same, and improves the throughput of the overall PM systems by reducing cache misses. However, in the NUMA architecture, the penalty of a miss for remote access is much higher than that for local access because remote PM access is significantly slower than local PM access. Therefore, in a NUMA system configured with PMs, the cache replacement algorithm should be designed to minimize the number of cache misses for remote PM access. To achieve this goal, we design a NUMA-aware cache replacement strategy called Remote Access First (RAF), which assigns higher weights to remote accesses according to access latency in evaluating access frequency. Next, we describe the RAF cache in detail. We summarize the symbols in Table I for a clear description.

Assume that there are n sockets in a NUMA architecture. For socket i (denoted as S_i), RAF maintains a set of latencies $L_i = \{l_{i-x}, 1 \leq x \leq n \cap x \in \mathbb{N}\}$, where l_{i-x} is the latency of accessing PM in S_i from S_x. For item j (denoted as C_j), RAF maintains a set of access frequency $F_j = \{f_{j-y}, 1 \leq y \leq n \cap y \in \mathbb{N}\}$, where f_{j-y} is the access frequency of C_j from S_y. Based on the above information, RAF can calculate a more realistic access frequency of C_j stored on S_i following Equation 1.

$$TF_j = \sum_{r=1}^{n} \frac{l_{i-r}}{l_{i-i}} f_{j-r} \tag{1}$$

The larger the TF of an item, the greater the benefit of being cached in DRAM. Unlike prior schemes, whether an item is cached in RAF depends not only on the access frequency but also on the miss penalty. This strategy allows us to prioritize the caching of frequently accessed data from remote NUMA sockets, optimizing the overall performance of the NStore.

The RAF cache structure is organized as a hash table, consisting of multiple hash buckets. Each bucket contains a fixed number of slots to store the cached items. For each hot/cached items, the RAF cache maintains four fields: the *lock* for concurrency control, the TF is used for hot item replacement, the socket ID on which the corresponding item is persisted, and the key value pairs.

The workflow of the RAF cache is described as follows: First, if a searched item j exists in the RAF cache, we update its corresponding TF following Equation 2.

$$TF_{new} = TF_{old} + \frac{l_{i-y}}{l_{i-i}} \tag{2}$$

Where, TF_{new} and TF_{old} represent TF values before and after the update, respectively. i and y represent the socket ID to which the searched item is persisted and the socket ID to which the get thread belongs, respectively. Second, if a searched item j is not found in the RAF cache, we read the item from the Nlog and insert it into the RAF cache by replacing the coldest item (i.e., the smallest TF value) in the same buckets. Finally, to prevent TF values from overflowing, we halve all TF values within the same bucket when an overflow occurs. However, the frequent triggering of TF updates or cached item replacements during every get operation negatively impacts the foreground operation performance. To address this issue, we adopt a periodical sampling strategy. After a certain number of get operations (32 by default), an TF value update or cached item replacement is triggered.

NStore minimizes the performance impact of the RAF cache by disabling it when detecting a uniform workload or write-intensive workload. To identify these workloads, NStore uses three signals: (1) the RAF cache receives less than 10% of all get operations (uniform), (2) all cached items in the same buckets have similar value of TF (uniform), and (3) the number of get operations in RAF cache is less than 10% of all operations (write-intensive).

C. Nlog Structure

To avoid remote PM writes, Nlog creates local Nlog files in each socket to accommodate writes from local threads. Besides,

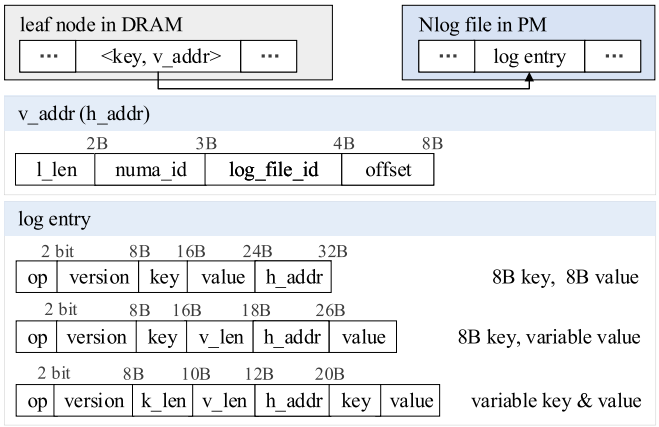


Fig. 5. Data layout in NStore.

we disable modification of full Nlog files, and only the latest Nlog file (called active file) in each socket supports writing. However, as the number of threads grows, a single local active file can become a bottleneck for write operations. To address this challenge, we cluster the keys using a hash function. The set of all log entries with the same hash ID (= hash(*key*)) is treated as a cluster (illustrated as Fig. 4). The clusters in each socket have their active files. Once an active file is full, it becomes a full Nlog file that cannot be written, and a new active file is created simultaneously. The default size of the Nlog file is 8 MB. Since there are no duplicate keys between hash clusters, write contention only occurs within the hash cluster, improving the concurrency of write operations.

D. Data Layout

The data layout of Nlog and volatile index is shown in Fig. 5. In Nlog, all write operations are persisted as log entries to the corresponding Nlog file. Only the address of the log entry with the latest value is stored in the volatile index (i.e., *v_addr*). The layout of *v_addr* is shown in Fig. 5. The *numa_id* is the socket ID to which the log entry belongs. The *log_file_id* is the Nlog file ID to which the log entry belongs. By using the *numa_id*, *log_file_id*, and the hash cluster ID calculated by hash(*key*), we can locate precisely a specific Nlog file. The *offset* is the address offset of the log entry in the Nlog file, and the *l_len* indicates the size of the log entry.

Nlog has three types of log entry structure for different kinds of workloads. For a workload with a fixed size for both keys and values, the log entry consists of five fields: *op*, *version*, *key*, *value*, and *h_addr*. The *op* records the type of operation, including insert (11, *ins*), update (10, *upd*), delete (01, *del*) and invalidate (00, *inv*); where *ins*, *upd*, and *del* are all valid log entries. The *version* records the version number for recovery; all log entries in a cluster share one version. *key* and *value* are used to persist the data. *h_addr* is a unique field for Nlog to implement lightweight and efficient garbage collection (Section III-F), which holds the address of the parent log entry of the current log entry. A key may be updated several times at runtime, and each modification will generate a new log entry. We store the address of the previous log entry (i.e., the parent log entry) of the key in the *h_addr* of the new log entry. With *h_addr*, all historical log entries of a key can be linked together. For a workload with variable-size values and fixed-size keys, we add the *v_len* field to record the length of the value. For a workload with variable size for both values and keys, we add *k_len* field to record the length of the key, in addition to the *v_len* field.

E. Basic Operations

Read. The read operation consists of the get operation to look up a targeted value using a given key and the scan operation to search all values in a key range.

Get. The get thread first finds the address of the targeted log entry in the volatile index (i.e., *v_addr*), then reads the targeted value in Nlog based on the *v_addr*.

Scan. The scan thread first gets all the *v_addr* of the keys within the query range, then reads the targeted values in Nlog. This process may involve random or remote PM reads, causing performance fluctuations. Therefore, inspired by delegation methods [22], [25], Nlog adopts the local PM reads scheme, which creates several background read threads for each socket (called local read threads) to improve scan performance. Each scan operation dispatches the PM read requests to the corresponding local read threads based on the *numa_id* in *v_addr*. With the local read threads, the scan operation not only fully utilizes the parallelism of PM to accelerate random reads but also eliminates remote reads. Our tests show that this scheme can significantly improve scan performance for most value sizes, but is unable to fully demonstrate the advantages of multi-threading when the value size is very small (e.g., 8B).

Concurrent read. Since all valid log entries cannot be modified, Nlog supports lock-free reads.

Write. The write operations include put, update, and delete. The put operation is to insert a new KV; the update operation is to modify an existing value; and the delete operation is to remove a KV from the index.

Put. The put thread first determines the corresponding local active file based on the hash (*key*) and then requests log entry space for the new KV according to the value length. It next persists the new KV with the latest version number and finally inserts the key and the *v_addr* of the new log entry into the volatile index. Since there is no parent log entry, the *h_addr* filed in the ins log entry is filled with the *invalid h_addr*. The *invalid h_addr* is a specified *h_addr* used to indicate no history log entries.

Update and delete. The update operation is the same as the delete operation and consists of three steps. First, find the *v_addr* of the targeted key (which is called *old v_addr*) in the volatile index. Second, append the update or delete operation to the corresponding local active file using the latest version number. The *old v_addr* obtained in the first step is written to the new log entry as *h_addr*. Third, update the *v_addr* of the targeted key in the volatile index or delete directly.

Concurrent write. Each hash cluster in a socket uses a write lock maintained in DRAM to support the write concurrency of

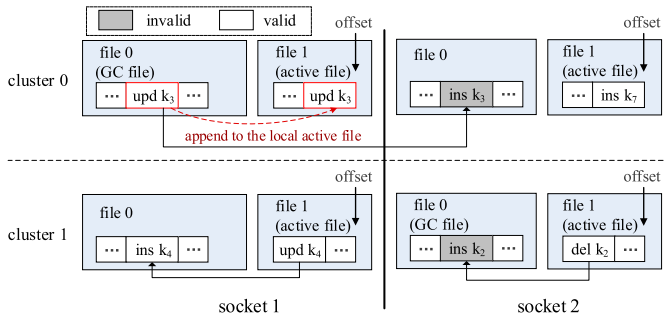


Fig. 6. Garbage collection for Nlog.

the active file. Note that when a write thread is allocated log entry space, it releases the lock and does not have to wait for the data in the log entry to be persisted.

Consistency of write operations. The write thread always writes *op* at the end when persisting the data in the log entry. Each active file is initialized with 0, so a new log entry is invalid until its *op* is updated to 01, 10, or 11. After a system crash, the recovery thread ignores invalid log entries, thus ensuring crash consistency of write operations.

F. Garbage Collection

NStore designs a lightweight and efficient garbage collection (GC) scheme to reclaim free space in the NLog. The GC scheme consists of two main steps: identify invalid log entries and reclaim Nlog files.

Identify invalid log entry. There are two common ways to track invalid log entries. The first is to directly modify the existing log entry during delete or update operations [4], [19]. However, this method may cause remote PM writes since the update/delete thread may not be in the same socket as the existing log entry. The second is to append updates and deletions directly and check whether the log entry is valid by accessing the volatile index [5], [26]. However, this approach interferes with user requests since each log entry, except deletion, requires a read operation in the volatile index. To minimize the impact on the foreground requests, NStore introduces a new field in log entry: *h_addr*. The addresses of the key's parent log entries are recorded in *h_addr*. Garbage collection threads can identify invalid log entries along *h_addr* without frequent accesses to the volatile index.

Fig. 6 illustrates the process of identifying invalid log entries. In a Nlog file, the GC thread reads every valid log entry. (1) When the *upd* log entry is read, all history log entries are searched along *h_addr* until encountering *invalid h_addr* (e.g., *ins* log entry), invalid log entry (i.e., *op* of the log entry is *inv*), or *h_addr* whose address does not exist (i.e., *h_addr* points to a log entry that has already been reclaimed). Then these entries are marked reversely as invalid (i.e., set the *op* to 00). Finally, check whether the *upd* log entry is valid, and set the *op* to 00 if it is invalid. (2) When the *ins* log entry is read, it is needed only to check whether the log entry is valid since the *h_addr* of the *ins* log entry is filled with *invalid h_addr*. (3) When *del* log entry is read, mark the log entry and its history log entry as invalid via the same process as described in (1), but without accessing the volatile index. With *h_addr*, garbage collection in Nlog minimizes interference with foreground threads. Since all history entries are obsolete, the GC thread does not interfere with the foreground requests when setting them to an invalid entry. Although remote PM writes may occur during access to history log entries (e.g., the GC thread 1 in socket 1 in Fig. 6 sets *ins k3* in socket 2 to invalid), these remote accesses are not on the critical path.

Reclaim Nlog file. For log structures that append deletes or updates to the latest log file, they must preferentially recycle older log files to avoid losing updates. For example, in Fig. 6, if *del k2* in file 1 (socket 2, hash cluster 1) is reclaimed before *ins k2* in file 0, deleted *k2* will restore life during crash recovery. However, it is difficult for Nlog to recycle log files based on creation time because Nlog files can be created concurrently on different sockets. Fortunately, because all historical log entries of a log entry are marked as invalid before it is recycled, any Nlog file can be recycled independently using *h_addr* to improve garbage collection efficiency. Whether a Nlog file will be reclaimed depends on its percentage of invalid log entries. Nlog appends the valid log entries to the corresponding local active file before reclaiming the Nlog file, which will cause serious rewrite overhead if there are too many valid log entries in the reclaimed Nlog file. Therefore, only Nlog files with a percentage of invalid log entries exceeding the preset threshold (*gc_threshold*) will be reclaimed by the GC thread.

As illustrated in Fig. 6, once the reclaimed Nlog file (noted as a GC file) is selected, the GC thread appends all valid log entries to the corresponding local active file (using the latest version number and *invalid h_addr*). During this process, the corresponding key of each valid entry in the volatile index will be locked until its value address is updated to the address of the new log entry. After all valid log entries are appended to the active file, the GC thread reclaims the GC file space.

Concurrency of garbage collection. For optimal performance, NStore creates one local GC thread for each socket to minimize remote PM access. For optimal space efficiency, NStore can create one local GC thread for each Nlog file (excluding active files) in theory to quickly reclaim PM space. However, excessive thread counts might diminish performance due to remote PM writes during GC operations, as depicted in Fig. 1(a). Therefore, limiting the number of GC threads to 8 or below is recommended. Note that no communication is required among GC threads.

G. Multi-Threaded Recovery

Before a normal shutdown, NStore copies the volatile index to a predefined location of PM and persists a flag beside it to indicate a normal shutdown. On system restart, the recovery thread first checks and resets this flag. If the flag is marked (indicates a normal shutdown), the recovery thread then loads the volatile index to DRAM. If the flag is unmarked, the recovery thread needs to rebuild the volatile index by scanning Nlog files from beginning to end, and most of the recovery time is spent on scanning. To reduce the recovery time, NStore adopts

a multi-thread scanning scheme. It assigns a scanning thread for each hash cluster. Within each hash cluster, if a duplicate key is encountered, the thread keeps the log entry with the latest version number and provides (*key*, *v_addr*) to rebuild the volatile index. For the RAF cache, we only initialize it in the recovery phase, and its reconstruction is completed by subsequent read operations, regardless of the normal or abnormal shutdown.

IV. Implementation

NStore can use any of the existing indexes as its volatile index. In our implementation, we deploy a learned-model-based index ALEX [27] in DRAM to work as the volatile index. The learned-model-based indexes, also known as learned indexes, use small and efficient machine-learning models for fast lookups, thus providing higher performance and a smaller index footprint than B+-Trees. The most prominent of the existing learning indexes is ALEX [28], which can contain more data (16 MB) per node (including leaf node and non-leaf node) by designing model-based insertion and lookup. However, the original ALEX does not support concurrency, which can be a performance bottleneck in dense access scenarios. To solve this problem, we implement *c-ALEX*, a concurrent ALEX based on node-level locks. Specifically, using node-level locks directly will incur significant lock contention for leaf nodes with large sizes and frequent writes. So, we reduce the size of leaf nodes to 512 KB to use node-level locks efficiently. For non-leaf nodes with few writes, we use the node-level lock to control concurrency but do not reduce the node size to avoid losing the advantage of large nodes.

V. Evaluation

A. Experimental Setup

Environment. All the experiments are performed on a Linux server (kernel version 5.4.0) with the persistent memory - Intel Optane DCPMMs. The server is equipped with four 128 GB Optane DCPMMs, 64 GB DRAM, and two Intel Xeon Gold 5218 2.30 GHz (16-core, 32-hyperthread) CPUs. Each CPU core has 32 KB L1 instruction cache, 32 KB L1 data cache, 1024 KB L2 cache, and 22 MB last-level cache. The four Optane DCPMMs and 64 GB DRAM are evenly attached to two CPU sockets (i.e., each CPU socket owns two Optane DCPMMs and 32 GB DRAM). The Optane DCPMMs are configured in the App Direct mode. In our evaluation, applications can access PM by first using the ext4-DAX file system to manage the pmem device, and then relying on PMDK (version 1.8.1) to allocate the PM space.

Comparisons. NStore is implemented with C++. We evaluate and compare NStore with seven available and open-source state-of-the-art PM KV stores: Viper [4], PACTree [23], FAST+FAIR [10], APEX [11], uTree [11], DPTree [12] and Nap [17]. Viper is a PM-DRAM KV store that includes a volatile hash table and a persistent DIMM-aligned log structure. It does not support scan operations. PACTree is a persistent trie-tree that uses a NUMA-aware persistent memory manager. FAST+FAIR is a PM-only B+-tree and does not use DRAM. APEX is a hybrid PM-DRAM learned index that places the metadata, lock, bitmaps, and fingerprints of keys in DRAM. uTree is a hybrid PM-DRAM B+-tree that puts non-leaf and leaf nodes in DRAM and stores a linked list of KVs in PM. DPTree first batches the writes in a DRAM B+-tree (buffer tree), then asynchronously merges these writes into a PM-DRAM trie-tree (base tree). The source code of DPTree does not support delete operations, so we do not include it in the corresponding experiments. Nap is a generic component that converts persistent indexes to be NUMA-aware. The authors of Nap provide a modified FAST+FAIR code that is compatible with Nap, called Nap-FF, which we use in our evaluation. Since Nap is only effective at skewed workloads and frequently crashes at large data volumes (e.g., 200M), we evaluate Nap and NStore independently in Section V-D using the dataset and workload configuration of the Nap paper [17]. We modify the original open-sourced codes for all comparison systems to make each thread allocate PM from its local PMDK pool and configure them using the default parameters stated in their papers. All codes are compiled using g++ 9.4.0 with -O3. Unless otherwise mentioned, we set the number of hash clusters to 256, set the size of the RAF cache to 64 MB, and configure all structures to run on all two NUMA sockets (i.e., each of the two sockets executes half of the threads).

Datasets. We use YCSB [29] and longlat (LLT) to test all the systems. Longlat is a realistic dataset from Open Street Maps (OSM)[1]. Unlike YCSB, longlat is transformed to become highly non-linear, consisting of compound keys that combine longitudes and latitudes from Open Street Maps. No duplicate elements are contained in these datasets. Each dataset has 200M key-value pairs. By default, the key and value sizes in all the datasets are 8 bytes. We randomly shuffle these two datasets to simulate real-world scenarios.

Workloads. We stress test all KV stores mainly using 7 workloads: *put100*, *put95 get5*, *put 50 get 50*, *get100*, *get50 upd50*, *upd100*, *delete100*. Among them, *put100* represents 100% of put operations (i.e., put workload), *upd100* represents update workload, and *put95 get5* is a mixed workload of 95% put operations and 5% get operations. For all runs except the put workload, we first warm the KV stores with 200M KVs and then perform 10M operations from different workloads; we treat the first warming step as a put workload.

B. Overall Performance

Put performance. Figs. 7(a) and 8(a) show the average throughput of put operations for KV stores on YCSB and LLT, respectively, and the performance of NStore increases steadily with the number of threads. Benefiting from local active files, NStore has significantly higher throughput than other structures even with only 2 threads; this advantage becomes increasingly apparent as the number of threads increases. With 32 threads, NStore's put throughput in the LLT dataset is 7.1×, 13.9 ×, 2.2×, 4.5×, 11.1×, and 1.8× higher than that of PACTree, FAST+FAIR, APEX, uTree, DPTree, and Viper,

[1] https://registry.opendata.aws/osm

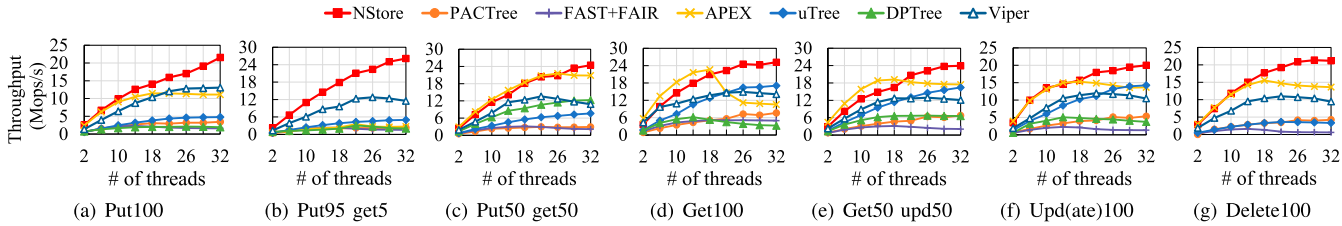


Fig. 7. Throughput on the YCSB dataset (200 million KVs).

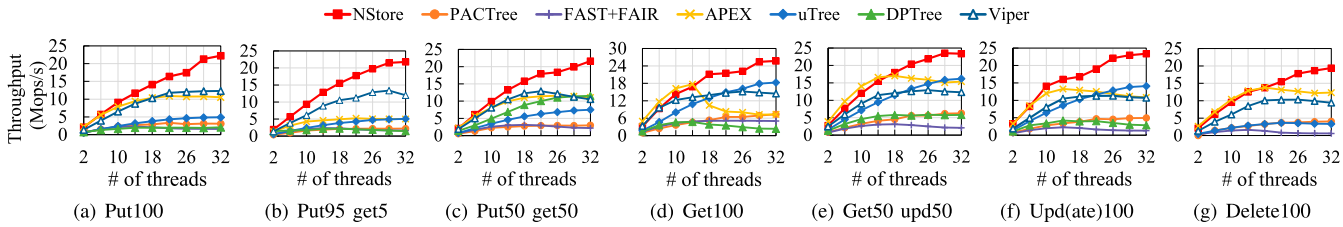
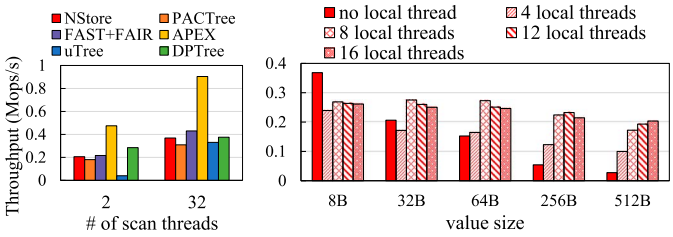


Fig. 8. Throughput on the LLT dataset (200 million KVs).

respectively. As the number of threads grows, FAST+FAIR, APEX, PACTree, and DPTree show similar throughput trends: the throughput decreases slightly with the number of threads. This is because they experience more frequent structure modifications, remote writes, and lock contention in PM. At high thread counts, the throughput of uTree and Viper does not drop but the growth rate slows down. The reason is that they both put the index in DRAM (so there are no structural modifications in PM), but as the thread counts grow, the number of remote PM writes will increase, slowing the throughput growth rate. In addition, the one reason Viper has higher throughput than uTree is that it assigns a thread to a log page, avoiding lock contention. Figs. 7(b) and 8(b) show the KV store's performance on a mixed workload of 95% put and 5% get operations. With 32 threads, NStore's *put95 get5* throughput is 9.9×, 13.6×, 4.3×, 4.4×, 5.6×, and 1.8× higher than that of PACTree, FAST+FAIR, APEX, uTree, DPTree, and Viper, respectively (on LLT dataset). The throughput of APEX and DPTree in *put95 get5* workload is severely degraded compared to *put100*. The throughput of APEX is degraded because it experiences structure modification operations, while DPTree is because it encounters merge operations.

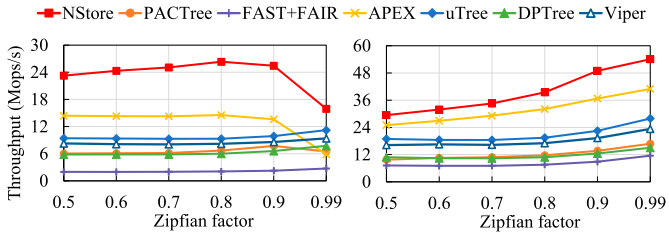
Get performance. Figs. 7(d) and 8(d) show the throughput of KV stores in get workload. When the number of threads is less than 14 (on the LLT dataset), the read throughput of NStore is slightly lower than APEX. However, as the number of threads grows, NStore exhibits significantly better performance than APEX and other structures. With 32 threads, the get throughput of NStore in the LLT dataset is 3.5×, 5.1×, 3.5×, 1.4×, 11.2×, and 2.1× better than that of PACTree, FAST+FAIR, APEX, uTree, DPTree, and Viper, respectively. It is worth noting that the get throughputs of APEX, DPTree, and FAST+FAIR drop significantly due to the remote PM reads at high thread counts, while NStore and uTree are unaffected. This is because NStore has local log files, and uTree allocates space for each KV, resulting in an even distribution of data in both sockets; in contrast, APEX, DPTree, and FAST+FAIR use large data nodes (especially for APEX), which cause a large amount of data to be written into one data node, leading to uneven data distribution. PACTree is not affected by remote PM access because it uses NUMA-aware space management, so the data is evenly distributed in the two sockets. The throughput gap between NStore and uTree is because we implement an efficient concurrent learned index as the volatile index. Viper uses a hash table in DRAM, so its throughput is higher than uTree at low thread counts. However, Viper requires three PM accesses per get operation (read lock, read data, and read lock again), which makes it experience more remote PM accesses at high thread counts, resulting in lower throughput. Figs. 7(c) and 8(c) show the throughput of indexes in a mixed workload of 50% put and 50% get operations. NStore, PACTree, FAST+FAIR, APEX, Viper, and uTree all have throughputs between the put workload (Figs. 7(a) and 8(a)) and the get workload (Figs. 7(d) and 8(d)), which is consistent with performance trends of other experiments.

Update performance. Figs. 7(f) and 8(f) show the throughput of KV stores in update workload. With 32 threads, the throughput of NStore in LLT dataset is 4.8×, 18×, 2.2×, 1.7×, 8.3×, and 2.3× larger than that of PACTree, FAST+FAIR, APEX, uTree, DPTree, and Viper, respectively. For NStore, PACTree, APEX, FAST+FAIR, and Viper, an update can be approximated as a combination of a get and a put operation, so their update throughput is very similar to the throughput of the *put50 get50*. DPTree only searches the buffer tree when updating and updates the value if the targeted key is found; if it is not found, it is directly written to the buffer tree as new data. Therefore, the update throughput of DPTree is similar to the put throughput. uTree requires only one PM write to update the value, while its put operation not only needs to allocate space for the new KV, but also requires at least two PM writes to add



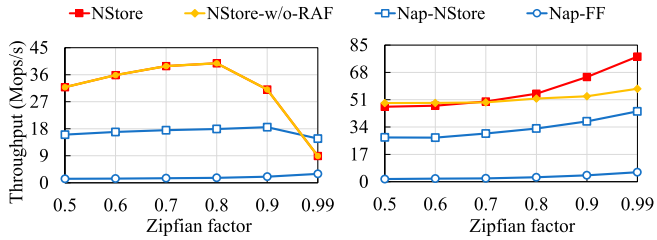
(a) 8B value *(NStore without local threads)* (b) Effect of different numbers of local PM threads at different value sizes (32 scan threads)

Fig. 9. Throughput of scan operation on the YCSB dataset.



(a) Update (b) Get

Fig. 10. Throughput under different skewness of Zipfian distribution (32 threads).



(a) Update (b) Get

Fig. 11. Throughput comparison with Nap on the YCSB dataset.

the new KV to the PM list. Therefore, the update throughput of uTree is better than the *put50 get50* throughput. Figs. 7(e) and 8(e) show the throughput of indexes in a mixed workload of 50% get and 50% update operations, and their throughput in *get50 upd50* is between *get100* and *upd100*.

Delete performance. Figs. 7(g) and 8(g) show the index throughput in a workload consisting of 100% delete operations. With 32 threads, NStore's delete throughput in the LLT dataset is 4.8×, 32.3×, 2.2×, 1.6×, 5.8×, and 2×, higher than PACTree, FAST+FAIR, APEX, uTree, and Viper, respectively. NStore deletes KVs by writing logs, so its delete throughput is similar to the put operation's. APEX completes deletion by updating the key to an invalid key, so its delete throughput is similar to its update throughput. The delete operation of uTree removes the targeted KV from the PM list and reclaims the space, so its delete throughput is similar to that of the put operation. Viper deletes a KV by updating the bitmap, so its delete throughput is similar to the update throughput. Among all the index structures, the delete operation of FAST+FAIR is the most complex and requires moving other KVs to avoid gaps, so when the number of threads increases, a large number of remote PM writes causes the delete throughput to drop.

Scan performance. Fig. 9(a) shows the scan performance of all systems. NStore performs moderately compared to other schemes, because the values are not stored in sorted order. With 32 threads, the scan throughput of NStore is 1.2× and 1.1× higher than that of PACTree and uTree, respectively, but lower than APEX, DPTree, and FAST+FAIR. APEX has the highest scan throughput due to its large data nodes (256 KB). Each data node in DPTree can hold 256 KVs, while FAST+FAIR only has 15 KVs. Therefore, the scan throughput of DPTree is higher in 2 threads. However, DPTree must read the data within the scan range from the buffer tree, which may bring remote PM reads. Consequently, the scan throughput of DPTree is lower than that of FAST+FAIR in 32 threads. Each leaf node in PACTree can hold 64 KVs, but it needs to construct an additional data array in DRAM when executing a scan (i.e., all the keys in the scan range are written to DRAM), causing its scan performance to be relatively low. With 2 threads, uTree performs the worst among all the indexes since traversing the PM list incurs a great number of CPU cache misses. However, with increased thread count, cache misses notably drop, leading to a substantial rise in scan throughput.

C. Performance With Skewed Access Patterns

We evaluate the performance of update and get operations with 32 threads on the YCSB datasets with different skewness (Zipfian factor varying from 0.5 to 0.99). Fig. 10(b) shows that with higher skewness, all KV stores perform better because accesses are focused on a smaller set of hot keys, making better use of the CPU cache and are less impacted by PM's high latency. Nevertheless, the effect of the RAF cache is still very significant: the more the skewness, the more improvement NStore attains in get throughputs. Update operations (in Fig. 10(a)) are different. When a large number of update operations are concentrated in a smaller set of hot keys, lock contention becomes a major bottleneck because the granularity of APEX and NStore write locks are relatively large. For other indexes, the CPU cache reduces PM accesses, resulting in higher update throughput at higher skewness.

D. Comparison With Nap

Nap, detailed in Section II-B, is a component that transforms existing PM KV stores into NUMA-aware stores. It primarily improves the performance of skewed workloads. In this experiment, we evaluate Nap and NStore based on the dataset and workload configuration of the Nap paper [17]. Specifically, we first load a 16M YSCB dataset, then execute 64M operations. Besides, we set the maximum number of KVs that the RAF cache can hold to 100 K, which is the same as for Nap.

Fig. 11 illustrates the update and get throughputs of NStore and Nap with a varies of skewness (32 threads). In Fig. 11, *NStore-w/o-RAF* indicates NStore without RAF cache. *Nap-FF* uses FAST+FAIR as the raw PM index for Nap, and *Nap-NStore* uses NStore-w/o-RAF as the raw PM index. For

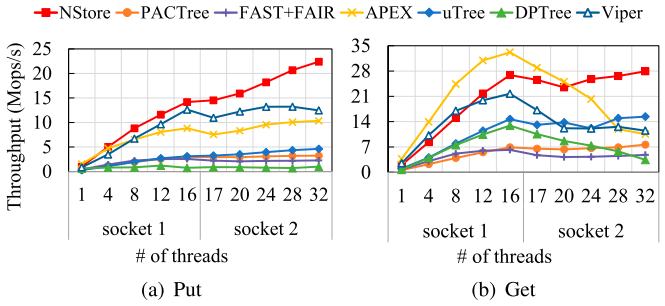


Fig. 12. Scalability with varying NUMA nodes (sockets) on the YCSB dataset.

a fair comparison, we cancel the creation of PM-views when testing the get throughput.

We observe that the update and get throughputs of NStore are significantly higher than those of Nap-NStore and Nap-FF. When the Zipfian factor (α) is 0.9, the update and get throughputs of NStore are 15× and 20× higher than those of Nap-FF. It is worth noting that the throughput of Nap-NStore is lower than that of NStore-w/o-RAF, which means that Nap has negatively optimized the update and get performance of NStore-w/o-RAF. The reason is that NStore implements complete local updates. However, Nap pointlessly introduces a large number of local PM writes when creating per-socket PM views during each hot dataset switch, consuming a lot of PM bandwidth. Additionally, in an update workload with $\alpha = 0.99$, the throughput of NStore drops due to the severe contention caused by coarse-grained write locks in the volatile index. When Nap shares some of the hot updates, the lock contention of NStore is mitigated. There are three reasons for Nap's negative optimization on get throughput: First, it switches hot keys at the coarser granularity of the set, hence the high cache misses. In more tests, we observe that the cache miss rate of RAF cache is below 10% when $\alpha = 0.99$, while that of nap is over 20%; Second, there are a lot of slower remote accesses in the missed requests. As shown in Fig. 3(a), remote reads account for up to 50% of missed reads (i.e., the total number of PM reads). Third, the performance of NStore is much higher than that of PM-only KV stores. This makes it difficult for Nap to improve performance on NStore.

E. Scalability With Number of Sockets

In this experiment, we present a performance test of each structure as the number of sockets grows. The first 16 threads are bound to socket 1, while the remaining 16 are bound to socket 2; each thread allocates PM from its local PMDK pool. For different numbers of threads, we first insert 200M KVs (put operations) and then perform 10M get operations.

Fig. 12 shows the throughput of the structures at different numbers of threads/sockets. We observe that the scalability of NStore is much better than that of the other structures. In Fig. 12(a), NStore is hardly affected by the increase in the number of threads/sockets because NStore creates a local Nlog file in socket 2 to receive writes from threads 17–32. However, the throughput growth rate of other schemes slows significantly when the number of threads increases from 17 to 32 due to remote PM accesses. In Fig. 12(b), the throughputs of all schemes drop when socket 2 is added, which is due to the remote PM accesses. However, as the number of threads further rises, the get throughputs of NStore, uTree, PACTree, and FAST+FAIR improve. This, however, is not true for APEX, DPTree, and Viper. The data node size of APEX and DPTree is large and it is difficult to distribute the data evenly in all sockets, so the proportion of remote PM reads is high. Interestingly, one reason for the better read performance of APEX than NStore on a single socket is that the super large nodes make the structure of APEX flatter and the access paths shorter. Viper requires at least three PM accesses per get operation, which leads to more remote accesses after adding socket 2, so it is difficult to recover the performance.

F. In-Depth Analysis for NStore

Effect of RAF cache. Fig. 11 also shows the read performance improvement of the RAF cache. Since RAF cache is only enabled under skewed get workloads, the update throughput of NStore and NStore-w/o-RAF is the same. From Fig. 11(b), we observe that the RAF cache can significantly improve the throughput of NStore as the Zipfian factor (α) increases. NStore slightly underperforms NStore-w/o-RAF under low-skewed workloads due to the cache miss penalties, which mainly come from the overhead of looking up cache, replacing cache items, and looking up missed requests in the volatile index and Nlog. RAF cache introduces a hash table structure to speed up in-cache lookups, uses a sampling strategy to minimize the performance fluctuations from cache replacement, and designs a remote-access-first replacement policy to minimize the misses on slower remote requests. In contrast, Nap cache has a negative optimization for NStore-w/o-RAF, because it replaces hot data in units of sets, causing more cache misses and higher replacement overhead. In addition, there are more remote requests in Nap's missed requests, so the miss penalties are more severe (Section V-D).

Effect of volatile index. We design a simple but effective concurrent learned index (i.e., *c-ALEX*) as the volatile index of NStore for fast indexing (Section IV). To demonstrate the impact of c-ALEX on NStore's performance, we implement *NStore-B* that uses B+-tree as the volatile index and Nlog as the PM storage. Fig. 13 shows that the put/get throughput of NStore is significantly higher than that of NStore-B at high thread counts. Especially, the put throughput of NStore grows faster than that of NStore-B as the number of threads increases. This is because we preserve large nodes in learned indexes (so fewer structural modifications caused by put operations) but also successfully mitigate the issue of lock contention under conditions of high concurrency.

Effect of Nlog. Nlog eliminates remote PM writes in NUMA using separate per-socket active files. Fig. 13 illustrates the performance of Nlog with other persistent storages, where *NStore* is c-ALEX (as the volatile index) + Nlog (as the persistent storage), *Viper-A* is c-ALEX + its log structure, *NStore-B* is B+-tree + Nlog, and *uTree* is B+-tree + PM list.

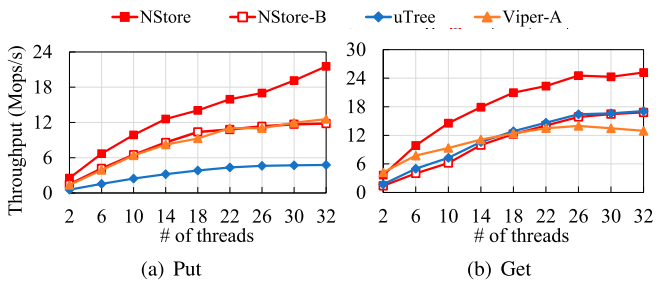


Fig. 13. Performance sensitivity of volatile index and Nlog on the YCSB dataset.

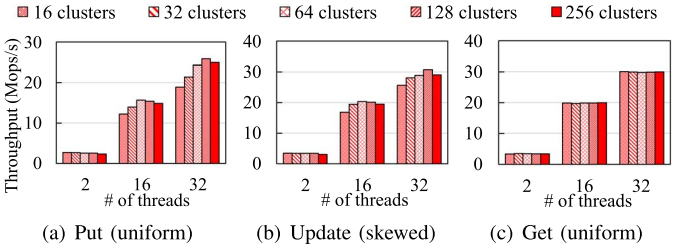


Fig. 14. Throughput of NStore under different numbers of hash clusters with 2, 16, and 32 threads on the YCSB dataset.

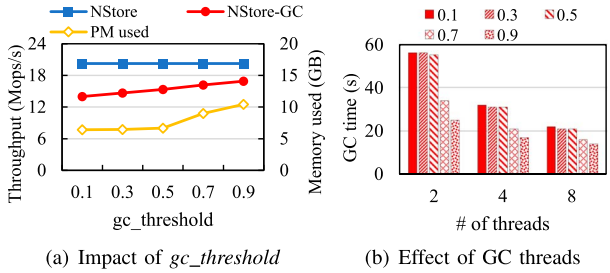


Fig. 15. Garbage collection overhead of NStore on the YCSB dataset.

The impact of Nlog can be demonstrated from two aspects: 1) Comparison with the existing log structures. Comparing NStore and Viper-A, we observe that Nlog's put/get throughput is higher than the log structure of Viper's. The reason is that NStore eliminates remote PM writes and need only one local PM write per put operation. In addition, Viper-A's get throughput is lower because it uses in-place updates, and thus requires additional PM reads for concurrency. 2) Comparison with other persistent storage. Comparing NStore-B and uTree, we observe that Nlog's put throughput is higher than PM list. This is because uTree needs to allocate space for new KVs in each put operation. NStore-B and uTree have similar get throughput because they both require only one PM read per get operation.

Effect of local PM reads scheme on scan operation. To improve the performance of scan operations, we design the local PM reads scheme (mentioned in Section III-E), which not only eliminates remote PM reads but also takes advantage of the internal parallelism of the PM. Fig. 9(b) shows the performance impact of different numbers of local PM read threads with 32 scan threads. Each socket has half of the local PM threads; that is, when there are 8 local PM read threads, 4 are included in each socket. We observed that the larger the value size, the easier it is for the scan operations to benefit from concurrent local read threads. When the value size is 8B, the thread communication causes a significant drop in scan throughput, and when the value size is 512B, the advantage of local PM threads is very apparent.

Effect of hash cluster. NStore avoids contention through hash clusters and improves the concurrency of write operations. In this paper, we set the default number of hash clusters to 256 for testing. Now, we use multiple numbers of hash clusters to explore more options. Fig. 14(a) shows the variation of put performance with the number of hash clusters ranging from 16 to 256. At 2 threads, the write concurrency is small, and too many hash clusters lead to a large number of random writes; thus, the best performance is achieved with 16 hash clusters. However, with the increase in threads, too few hash clusters can lead to severe write contention. Therefore, the best performance is achieved when the number of hash clusters is 64 for 16 threads, and when the number of threads increases to 32, the optimal number of hash clusters is 128. Fig. 14(b) shows the throughput variation of the update with a Zipfian at 0.9 as the number of hash clusters increases from 16 to 256. Similar to the even put workload performance, the number of hash clusters with the best performance grows as the number of threads increases. Thus, changing the number of hash clusters does not affect the performance of the skewed workload. Fig. 14(c) shows the variation of read performance as the increase in the number of hash clusters. Changing the number of hash clusters does not affect the read performance.

Garbage collection. We evaluate the garbage collection overhead from two viewpoints: foreground workload priority (i.e., fewer garbage collection threads) and garbage collection priority (i.e., more garbage collection threads). For each run, we first load the YCSB dataset in NStore and then update 75% of the data. (a) To prioritize foreground requests, we assign only one GC thread per socket and analyze the GC's impact on write performance and space usage for various *gc_threshold* values. Fig. 15(a) shows the results. A larger *gc_threshold* means fewer Nlog files to reclaim, and both the write performance and space usage increase. Moreover, the increase in write performance with growing *gc_threshold* is small and linear, however, when *gc_threshold* exceeds 0.5, PM usage increases significantly, so the default *gc_threshold* is 0.5. (b) To prioritize garbage collection, we study the effect of the number of GC threads on GC time with varying *gc_threshold* values. Fig. 15(b) shows that the GC time is reduced by 61% as the GC thread counts increase from 2 to 8, when *gc_threshold* is 0.1. Hence, in scenarios with limited PM space or light foreground workloads, additional GC threads can expedite recycling.

G. Recovery Overhead

We evaluate the recovery time of NStore with varying numbers of KV pairs, value sizes, and recovery threads, and the results are shown in Fig. 16. Note that only the volatile index needs to be rebuilt during recovery, which does not

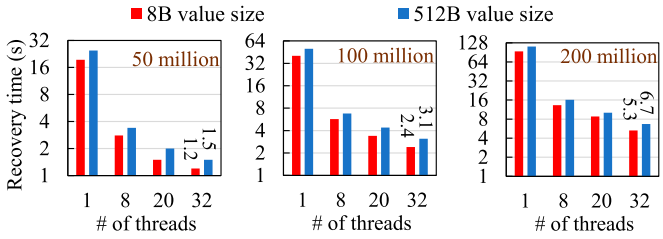


Fig. 16. Recovery overhead of NStore on the YCSB dataset.

incur any PM writes. When the value size is 8 B, the recovery process with 32 threads takes only 1.2/2.4/5.3 seconds to rebuild the volatile index containing 50/100/150 million KV pairs, respectively. In the test with 100M KVs and 8 B value size, the recovery time was reduced by 94% from 40.4 s with a single thread to 2.4 seconds with 32 threads. This reduction in recovery time is also achieved in the recovery tests with different value sizes and data volumes, demonstrating the effectiveness of our multi-threaded recovery strategy in reducing scanning costs and speeding up recovery in various scenarios. It is worth noting that we used 256 hash clusters in our tests, and with 32 threads, each recovery thread needs to scan 8 hash clusters. However, in our design, every hash cluster can have a standalone recovery thread, so the recovery time will be further reduced as the number of threads increases.

VI. Related Work

PM Indexes. Existing persistent indexes are optimized for crash consistency and performance [7], [8], [10], [11], [12], [14]. Here are a few representative ones. FAST+FAIR [10] is a PM-only B+-tree index designed to minimize crash consistency overhead by tolerating transient inconsistency. APEX [11] is a hybrid learned index placing internal nodes and leaf nodes on PM and maintaining sophisticated data structures on DRAM to improve performance. In addition, APEX has a large node size and flatter structure by using machine learning models in each node to accelerate the search. uTree [8] is a hybrid B+-tree index designed to reduce tail latency by placing all nodes of B+-tree in DRAM and coordinating concurrency control. DPTree [12] is a hybrid index designed to amortize crash consistency overhead by batch writes. Specifically, DPTree first appends data to the buffer tree, then merges the data in the buffer tree to the base tree when the buffer tree reaches the size threshold. However, almost all state-of-the-art persistent indexes forgo the effect of NUMA. Thus, they do not scale well with the number of sockets.

NUMA-aware Techniques. Researchers proposed many techniques to solve the NUMA problem on storage engines, including *delegation* [22], [25], [30] for file systems (or resources contention management) and *replication* [16], [31], [32] for DRAM indexes. The delegation technique uses local worker threads to perform tasks delegated by the application threads, which causes a huge thread communication overhead, especially for small-sized indexing operations. The replication technique replicates all or part of indexes in each socket, which causes every update operation to be executed at every socket, and thus is very unfriendly to PM with crash recovery overhead and low bandwidth. In addition, a few NUMA-aware PM indexes have been proposed. ListDB [33] designed a braided skiplist, which reduces remote access by building a separate upper layer in each socket. However, this solution increases the number of local PM access. PACTree [23] allocates local PM space for structure modification threads to reduce remote PM writes, but this scheme cannot eliminate remote PM writes. Nap [17] proposed a generic NUMA-aware component, which reduces remote PM access by caching hot data from the PM KV stores into DRAM. However, Nap's NUMA optimization for the PM KV stores is minimal.

Log-structured Key-Value Stores. The log-structured design is prevalent in KV stores. FASTER [19] is a hash-based KV store employing a volatile index and a hybrid log for disk storage with in-place updates and "roll-to-tail" garbage collection. However, in-place updates introduce remote access into the critical path of the operations. Moreover, the premise of "roll-to-tail" is that there is a single tail for new entries, leading to remote PM writes when appending entries. NStore maintains multiple tails (i.e., active files) to enable fully local PM writes and mitigate lock contention. Nibble [18] is a hash-based many-core KV store with a multi-head log structure. However, Nibble is not suitable for PM due to the lack of data consistency and crash recovery mechanisms. NStore assigns an active file to a hash cluster instead of a core in Nibble, thus enabling fast recovery and more balanced write operations. Furthermore, NStore designs an efficient garbage collection for this log structure that ensures data consistency. These measures allow NStore to fit pure DRAM and PM-DRAM hybrid memory. FlatStore [5], Viper [4], and RStore [6] are all PM-DRAM hybrid KV stores that utilize volatile indexes on DRAM and log-structured storage on PM. FlatStore proposes pipelined horizontal batching to batch small-sized requests, achieving low latency and high throughput. Viper assigns threads to different PM regions to minimize the thread-to-DIMM ratio, and stores data in DIMM-aligned segments to balance DIMM contention. RStore aims to reduce tail latency via the asynchronous programming model, message-passing communication, and log-structured storage. However, these approaches are not optimized for NUMA and they do not run efficiently in the NUMA architecture due to the unavoidable large number of remote PM accesses.

VII. Conclusion

In this work, we propose NStore, a high-performance NUMA-aware key-value store containing a read-optimized cache (RAF cache) designed for NUMA architectures, a persistent log-structured storage (Nlog) to localized PM writes, and a volatile index for fast indexing of Nlog. RAF cache minimizes remote PM reads by giving the latter higher cache priority. Further, with the independent local Nlog files, NStore completely eliminates remote PM accesses in write operations. In addition, we present several strategies for Nlog to optimize scan, garbage collection, and recovery performance. Extensive evaluations show that NStore achieves up to 13.9× and 11.2× improvements in put and get throughput, respectively, compared to recent PM systems.

ACKNOWLEDGMENT

We appreciate all reviewers and editors for their insightful comments and feedback.

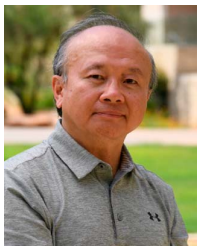
REFERENCES

[1] H.-S. P. Wong et al., "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[2] D. Apalkov et al., "Spin-transfer torque magnetic random access memory (stt-mram)," *ACM J. Emerg. Tech. Comput. Syst.*, vol. 9, no. 2, pp. 1–35, 2013.
[3] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Tech.*, 2020, pp. 169–182.
[4] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid pmem-dram key-value store," *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.
[5] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 1077–1091.
[6] L. Lersch, I. Schreter, I. Oukid, and W. Lehner, "Enabling low tail latency on multicore key-value stores," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1091–1104, 2020.
[7] Z. Wang et al., "ComboTree: A persistent indexing structure with universal operational efficiency and scalability," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2277–2290, Oct. 2022.
[8] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "uTree: A persistent B+-tree, with low tail latency," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.
[9] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "{WORT}: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Tech.*, 2017, pp. 257–270.
[10] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}," in *Proc. 16th USENIX Conf. File Storage Tech.*, 2018, pp. 187–200.
[11] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "APEX: A high-performance learned index on persistent memory," 2021, *arXiv:2105.00683*.
[12] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "DPTree: Differential indexing for persistent memory," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.
[13] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: The design and implementation of a fast persistent key-value store," in *Proc. 27th ACM Symp. Oper. Syst. Princ., (SOSP)*, New York, NY, USA: ACM, 2019, pp. 447–461. [Online]. Available: https://doi.org/10.1145/3341301.3359628
[14] F. Xia, D. Jiang, J. Xiong, and N. Sun, "{HiKV}: A hybrid index {Key-Value} store for {DRAM-NVM} memory systems," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 349–362.
[15] Y. He, D. Lu, K. Huang, and T. Wang, "Evaluating persistent memory range indexes: Part two," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 2477–2490, 2022.
[16] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for numa architectures," in *Proc. 22th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 'New York, NY, USA: ACM, 2017, pp. 207–221, doi: 10.1145/3037697.3037721.
[17] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A {Black-Box} approach to {NUMA-Aware} persistent memory indexes," in *Proc. 15th USENIX Symp. Oper. Syst. Des. Implementation*, 2021, pp. 93–111.
[18] A. Merritt, A. Gavrilovska, Y. Chen, and D. Milojicic, "Concurrent log-structured memory for many-core key-value stores," *Proc. VLDB Endowment*, vol. 11, no. 4, pp. 458–471, 2017.
[19] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 275–290.
[20] I. Corporation, "Intel reports second-quarter 2022 financial results," 2022. Accessed: Jul. 25, 2023. [Online]. Available: https://www.intc.com/news-events/press-releases/detail/1563/
[21] J. Handy and T. Coughlin, "Persistent memories without optane, where would we be?" Accessed: Aug. 01, 2023. [Online]. Available: https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memoryimplementing-nvdimm-n-architecture.
[22] D. Zhou, Y. Qian, V. Gupta, Z. Yang, C. Min, and S. Kashyap, "{ODINFS}: Scaling {PM} performance with opportunistic delegation," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implementation*, 2022, pp. 179–193.
[23] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Principles*, 2021, pp. 424–439.
[24] C. Lameter, "NUMA (non-uniform memory access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors." *Queue*, vol. 11, no. 7, pp. 40–51, Jul. 2013, doi: 10.1145/2508834.2513149.
[25] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf. (USENIXATC)*, Berkeley, CA, USA: USENIX Association, 2011, p. 1.
[26] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, 2017.
[27] J. Ding et al., "Alex: An updatable adaptive learned index," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 969–984.
[28] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3004–3017, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3004-wongkham.pdf
[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[30] I. Calciu, J. Gottschlich, and M. Herlihy, "Using elimination and delegation to implement a scalable numa-friendly stack," in *Proc. 5th {USENIX} Workshop Hot Topics Parallelism*, 2013, pp. 1–7.
[31] A. Mathew and C. Min, "HydraList: A scalable in-memory index using asynchronous updates and partial replication," *Proc. VLDB Endowment*, vol. 13, no. 9, pp. 1332–1345, 2020.
[32] L. Cui, K. Yang, Y. Li, G. Wang, and X. Liu, "DiffLex: A high-performance, memory-efficient and NUMA-aware learned index using differentiated management," in *Proc. 52nd Int. Conf. Parallel Process.*, 2023, pp. 62–71.
[33] W. Kim et al., "ListDB: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implementation*, 2022, pp. 161–177.

Zhonghua Wang received the bachelor's degree in computer science from Zhengzhou University, China, in 2016. She is currently working toward the Ph.D. degree with Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China. Her research interests include hybrid storage device (i.e., non-volatile memory, DRAM, etc.), learned index, and key-value systems.

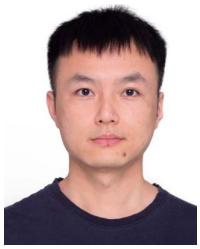
Kai Lu received the B.S. and Ph.D. degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2018 and 2023, respectively. Currently, he is a Postdoctoral Researcher with Wuhan National Laboratory for Optoelectronics (WNLO), HUST. His research interests include computer architecture, distributed storage systems, and key-value stores.

Jiguang Wan received the bachelor's degree in computer science from Zhengzhou University, China, in 1996, the M.S. and Ph.D. degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2003 and 2007, respectively. Currently, he is a Professor with Wuhan National Laboratory for Optoelectronics (WNLO), HUST. His research interests include computer architecture, networked storage system, key-value storage system, parallel, and distributed system.



Hong Jiang (Fellow, IEEE) received the B.Sc. and M.A.Sc. degrees in computer engineering from Huazhong University of Science and Technology, and the University of Toronto, respectively, and the Ph.D. degree in computer science from Texas A&M University. Currently, he is the Chair and Wendell H. Nedderman Endowed Professor of Computer Science and Engineering Department, The University of Texas at Arlington. Prior to joining UTA, he served as a Program Director at the National Science Foundation (2013–2015) and he was with the University of Nebraska-Lincoln since 1991, where he was Willa Cather Professor in computer science and engineering. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, and cloud and edge computing. He is a Topic Editor and an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERSand member of ACM.

Zeyang Zhao received the bachelor's degree in computer science and technology from Huazhong University of Science and Technology, China, in 2021. He is currently working toward the master's degree in storage systems and techniques from Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology. His research interests include systems optimization for emerging storage technologies, learned index, and key-value store.



Peng Xu received the bachelor's and Ph.D. degrees in computer science from Huazhong University of Science and Technology, China. Currently, he is with Zhejiang Lab. His research interests include computer architecture, key-value storage systems, and graph computing.

Biliang Lai received the bachelor's degree in computer science and technology from Huazhong University of Science and Technology, China, in 2023. Her research interests include computer architecture, memory index, and non-volatile memory.

Guokuan Li received the bachelor's, M.S., and Ph.D. degrees from Huazhong University of Science and Technology (HUST), in 1993, 1996, and 2000, respectively. Currently, he is an Associate Professor with Wuhan National Laboratory for Optoelectronics (WNLO), HUST. His research interests include digital image processing/analysis, computer vision, machine learning, and data storage system.

Changsheng Xie (Member, IEEE) received the B.S. and M.S. degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 1982 and 1988, respectively. Currently, he is a Professor with the Department of Computer Engineering, Huazhong University of Science and Technology. He is also the Director of the Data Storage Systems Laboratory, HUST and the Deputy Director of Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, disk I/O system, networked data storage system, and digital media technology.