BonsaiKV: Towards Fast, Scalable, and Persistent Key-Value Stores with Tiered, Heterogeneous Memory System

Miao Cai
Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University
School of Computer and Information, Hohai University
mcai@hhu.edu.cn

Junru Shen
School of Computer and Information, Hohai University
jrshen@hhu.edu.cn

Yifan Yuan
Intel Labs
yifan.yuan@intel.com

Zhihao Qu
Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University
School of Computer and Information, Hohai University
quzhihao@hhu.edu.cn

Baoliu Ye
State Key Laboratory for Novel Software Technology, Nanjing University
Key Laboratory of Water Big Data Technology of Ministry of Water Resources, Hohai University
School of Computer and Information, Hohai University
yebl@nju.edu.cn

ABSTRACT

Emerging NUMA/CXL-based tiered memory systems with heterogeneous memory devices such as DRAM and NVMM deliver ultra-fast speed, large capacity, and data persistence all at once, offering great promise to high-performance in-memory key-value stores. To fully unleash the performance potential of such memory systems, this paper presents BonsaiKV, a key-value store that makes the best use of different components in a tiered memory system. The core of BonsaiKV is a tri-layer hierarchical storage architecture that separates data indexing, persistence, and scalability from each other and realizes each of them within a specialized software-hardware layer. We design BonsaiKV with a set of novel techniques, including collaborative tiered indexing, NVMM congestion control mechanisms, fine-grained data striping, and NUMA-aware data management, to leverage hardware strengths and tackle device deficiencies. We compare BonsaiKV with state-of-the-art NVMM-optimized key-value stores and persistent index structures using a variety of YCSB workloads. Evaluation results demonstrate that BonsaiKV outperforms others by up to 7.69×, 19.59×, and 12.86× in read-, write- and scan-intensive scenarios, respectively.

PVLDB Reference Format:
Miao Cai, Junru Shen, Yifan Yuan, Zhihao Qu, and Baoliu Ye. BonsaiKV: Towards Fast, Scalable, and Persistent Key-Value Stores with Tiered, Heterogeneous Memory System. PVLDB, 17(4): 726 - 739, 2023.
doi:10.14778/3636218.3636228

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636228

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/miaogecm/BonsaiKV.git.

1 INTRODUCTION

In-memory key-value store (KVS) has become an indispensable building block of today's cloud computing systems, serving in numerous scenarios like object caching [60], web indexing [65], and stream processing [10]. Conventional DRAM-based KVS, such as Memcached [60], Redis [3], and RAMCloud [62], reap high-performance DRAM to deliver orders of magnitude performance speedup over HDD- or SSD-based counterparts [11, 50, 53].

However, DRAM inefficiencies in cost, density, and scaling limitation raise severe concerns for today's in-memory key-value stores [58, 64]. Cloud service providers still heavily rely on cheap, slow storage devices (e.g., flash-memory-based SSDs) to compensate for DRAM shortcomings [23, 43]. Unfortunately, such workarounds often cause slow and unpredictable key-value store performance [82], failing to meet the strict performance requirements driven by big data applications [25, 40, 48, 51, 82].

The advent of heterogeneous memory systems [55, 67, 68] with emerging non-volatile memories (e.g., 3D XPoint [81], PCM [66], Z-NAND [1]) and cache coherent interconnect protocols (e.g., CXL [28], OpenCAPI [73], NVLink [47]) promise a solution to address this tension. For example, non-volatile main memory (NVMM) systems offer nano-seconds access latency, maximum 512GB capacity per DIMM slot, storage-like data persistence, byte-addressability via memory controllers [30, 81]. Also, the NUMA architecture and CXL-based memory expansion [16] enable low-cost and application-transparent memory capacity scale-up. All these technologies exhibit distinctive and attractive characteristics, together forming a tiered memory hierarchy. Given the current device/technology

availability, this paper focuses on a NUMA system with DRAM-NVMM hybrid memory.

With such exciting and revolutionary characteristics and features, however, how to fully unleash the performance potential of such memory systems for key-value stores is still nontrivial and poorly explored. Specifically, this paper identifies three major challenges to this end. **(1)** DRAM is fast and has small latency variations against different access patterns. A consensus derived from current research is to use DRAM data structures for fast key-value indexing [5, 14, 31, 34]. Unfortunately, the small capacity and high price of DRAM modules constrain such use in indexing large volumes of data for the production environment [9, 19, 60]. **(2)** NVMM exhibits relatively small bandwidth [17, 30, 81]. Data writes issued by increasing parallel executing thread easily saturate the small bandwidth, causing severe NVMM traffic congestion [84] or even leading to performance collapse under high contention [86]. **(3)** A tiered DRAM-NVMM memory system incorporates multiple memory nodes consisting of channel-isolated memory modules [81]. How to scale key-value stores by leveraging hardware-provided parallelism and mitigating remote memory access overheads is not well understood.

With these challenges in mind, we present BonsaiKV, a fast, scalable, persistent key-value store on tiered, heterogeneous memory systems. BonsaiKV features a DRAM-NVMM tri-layer hierarchical storage architecture that spans across a hybrid, large-scale memory system consisting of distinct memory devices.

BonsaiKV aims to *leverage hardware strengths and tackle device deficiencies to fulfill efficient indexing, fast persistence, and high scalability in a key-value store simultaneously.* Targeting these goals, it incorporates a set of novel key-value data indexing, persistence, and scalability techniques:

- We propose collaborative tiered indexing by exploiting both DRAM's fast speed and NVMM's large capacity to reduce the data index memory consumption without compromising indexing performance §4.1.
- We develop an analytical model to understand NVMM congestion and then propose two congestion control mechanisms to reduce traffic interference and mitigate bandwidth contention §4.2.
- We devise a scalable fine-grained data stripe scheme by fully leveraging memory channel-level parallelism §4.3.
- We propose an adaptive KV data replication mechanism and a write-optimal data coherence protocol to alleviate remote memory access overheads §4.3.

We implement BonsaiKV prototype from scratch on a commodity DRAM-NVMM platform. Functionality separation and layer specialization of BonsaiKV allows its three layers to be independently designed, implemented, and optimized, which greatly simplifies its development and enables easy adoption on other heterogeneous memory systems §4.4.

We compare BonsaiKV with three state-of-the-art DRAM-NVMM key-value stores (LSM-tree-based ListDB [41], Log-structured FlatStore [14, 77], Hash-based Viper [5]) and three persistent index structures (trie-based PACTree [42], B^+-tree-based FAST-FAIR [35], hybrid-index DPTree [87]). We also use a variety of YCSB workloads [15] to evaluate BonsaiKV performance. Evaluation results demonstrate that BonsaiKV significantly outperforms state-of-the-art works under different data-intensive scenarios.

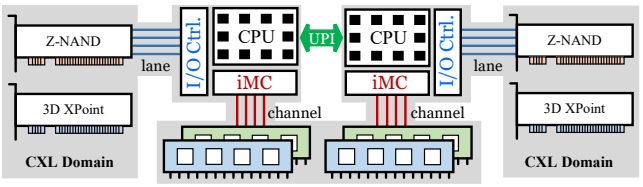


Figure 1: Modern Tiered Heterogeneous Memory System

The rest of this paper is organized as follows. Section §2 demonstrates the background and motivation of our work. Section §3 gives an overview of BonsaiKV. Section §4 elaborates on the design and implementation. Section §5 thoroughly evaluates the effectiveness of techniques and the overall performance of BonsaiKV. Finally, Section §6 gives a conclusion.

2 BACKGROUND AND MOTIVATION

2.1 Modern Tiered Heterogeneous Memory

With the explosive demand for big data storage and the advance of memory technologies, modern computer systems in data centers have embraced tiered memory systems with heterogeneous memory mediums for different trade-offs [49, 55]. As depicted in Figure 1, in addition to DRAM DIMM, programmers are able to leverage a wide range of emerging technologies to expand their memory space. For example, NVMM-based persistent DIMM has been commercialized to serve as compensation for DRAM [2]. In a typical DRAM-NVMM hybrid system, where both are attached to the memory bus and addressed by the integrated memory controller (iMC), it provides fast speed, large capacity, and data persistence [81]. Furthermore, inter-CPU coherent interconnects (e.g., Intel UPI) also scale the memory out of a single CPU to a non-uniform memory access (NUMA) architecture.

Recently, such efforts have been extended to CXL memory expansion [28, 49, 55], which defines the standard interface exposed to the host CPU, and has no constraints on memory/storage types. In CXL memory expansion, devices are treated as a regular NUMA node without CPU cores, enabling software-transparent memory expansion and offloading with load/store memory semantics, and all memory management software for regular NUMA systems can be seamlessly applied. Last but not least, advanced networking technologies have brought disaggregated memory [22, 29, 68] to the horizon for even larger memory capacity and higher resource utilization. All these innovations provide extensive design space for data center applications [70, 75, 79], exerting significant influence on system and architecture exploration.

2.2 Challenges and Related Works

We summarize and discuss three major technical challenges of building key-value stores on DRAM-NVMM. Moreover, we analyze related works and present a technical comparison of various state-of-the-art key-value stores in Table 1.

DRAM is fast but has limited capacity. DRAM outperforms NVMM for lower latencies and smaller variance for different access patterns. As a result, a line of research works [5, 13, 14, 52, 61, 80, 87] use volatile data structures to index data stored in the NVMM. Using

Table 1: Comparison of Various Key-Value Stores. *Most key-value stores excessively use DRAM for fast indexing, which causes high DRAM consumption. Second, NVMM congestion is a common problem for key-value stores, yet no effective solutions exist. Finally, current key-value stores support multi-DIMM, multi-node architecture poorly. They only provide page-level data striping and NUMA-aware memory allocation.*

	Indexing		Persistence		Scalability	
Project	Speed	DRAM Usage	Interf-erence	Conte-ntion	Inter-leave	NUMA-aware
HiKV [80]	Fast	High	High	High	✗	○
FAST-FAIR [35]	Slow	✗	High	High	✗	○
DPTree [87]	Fast	Medium	High	High	✗	○
FlatStore [14]	Fast	High	High	High	4KB	○
Viper [5]	Fast	High	High	High	4KB	○
LB$^+$-tree [52]	Medium	Medium	High	High	4KB	○
PACTree [42]	Slow	✗	High	High	4KB	◐
ListDB [41]	Slow	Low	High	High	4KB	●
BonsaiKV	Fast	Low	Low	Low	256B	●

○: NUMA-unaware; ◐: NUMA-aware NVMM Memory Allocation; ●: NUMA-aware NVMM Memory Allocation and Data Access.

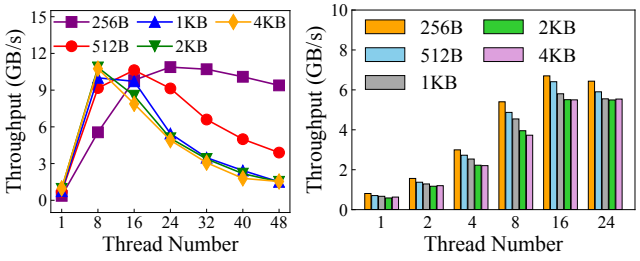
fast DRAM improves data indexing speed. However, the current DDR4 DRAM module capacity ranges from 8GB to 32GB, which is far less than the NVMM capacity (128-512GB for Intel DCPMM [2]). Indexing large volumes of data causes heavy DRAM pressure. We find this data indexing solution even becomes infeasible in the production environment [9].

HiKV [80], Viper [5], and FlatStore [14] cause high DRAM consumption. For every key-value pair in the NVMM, they create a volatile indexing entry [key, address] and manage entries using tree-like or hash table-based DRAM data structures. A recent RocksDB study from Facebook reports that a database node stores nearly eight hundred gigabytes of social graph data [57]. Assume the average key and value sizes are 27 bytes and 126 bytes, respectively [9]. Indexing these key-value data consumes around 183GB of precious DRAM resources. It occupies over 95% of DRAM space for our testbed machine §5.

To reduce the heavy DRAM pressure, hybrid DRAM-NVMM tree structures (e.g., FPTree [61] and LB$^+$-tree [52]) store leaf nodes in the large-sized NVMM. However, the overall indexing performance is compromised due to low NVMM node lookup performance. The reasons are twofold. The first one is expensive NVMM access to leaf node metadata and key-value data. Moreover, they usually store a leaf node in a device. The poor NVMM bandwidth limits the entry search efficiency in the same node.

In addition, ListDB [41] only places a small MemTable in DRAM. Most of its lookups need to search the multi-level LSM tree in NVMM. Therefore, its indexing speed is quite slow. *For fast but small DRAM, how to reduce the memory consumption without degrading the data indexing performance is the first challenge.*

NVMM congestion preventing fast persistence. In a tiered memory system, the device bandwidth varies across tiers. A remote NUMA memory node or a CXL-connected device has smaller bandwidth than the local memory bandwidth [49, 55]. Similarly, NVMM and DRAM also have a large bandwidth gap (3-10×). The small NVMM bandwidth is easily saturated with a burst of data requests. This phenomenon resembles the network congestion [20]. For data



(a) Poor NVMM Bandwidth **(b) Channel-Level Parallelism**

Figure 2: NVMM Bandwidth Performance Characteristics. *(a) shows that NVMM bandwidth is easily saturated with a large data persistence granularity or an increasing thread number. (b) demonstrates that a smaller data striping size delivers better hardware parallelism and higher throughputs.*

persistence, NVMM congestion leads to two severe consequences: *traffic interference* and *bandwidth contention*.

In NVMM congestion, memory traffic of different data flows has strong interference, causing unpredictable client performance. This problem is especially serious for log-structured key-value stores (e.g., ListDB [41], FlatStore [14]). A log-structured key-value store is equipped with background garbage collection (GC) threads. When GC threads clean stale logs, they compete for limited bandwidth resources with foreground user threads. Consequently, high-prioritized user requests are unable to be served in time. Furthermore, storing the entire index in the NVMM like FAST-FAIR [35] and PACTree [42] exacerbates this issue.

Besides, NVMM congestion also leads to the heavy bandwidth contention. The value size could be quite large in modern key-value data storage. For instance, UP2X, an AI/ML service database in Facebook, has an average value size of 3.6KB. Its maximum value size even exceeds 100KB [9]. A recent cache study also suggests that user requests are often bursty [6]. These massive, large-sized data writes stress the limited NVMM bandwidth.

We study the bandwidth contention with an experiment. This experiment creates multiple memory regions. A region is interleaved across six NVMM devices. Threads persist data in their private regions. Our experimental results in Figure 2a show that bandwidth contention easily happens for a large data persistence size or increasing threads. The root cause of the high bandwidth contention is the mismatch between data request transmission rate and NVMM device service rate [86]. This rate disparity is proportional to the data persistence size and the thread number. Section §4.2.1 gives a detailed analysis.

Heavy NVMM congestion is a serious problem for data persistence. Table 1 investigates that all existing key-value stores suffer from this issue. Unfortunately, none of them have effective congestion control mechanisms to deal with it. Other works require special hardware mechanisms, e.g., MBA [84] and DVFS [36]. For example, MT^2 [84] reduces the bandwidth contention by regulating the memory bandwidth with Intel MBA. FairHym [36] throttles CPU frequency using DVFS to alleviate interference among tasks running on DRAM and NVMM. *In summary, for data persistence techniques, how to address NVMM congestion is the second challenge.*

Scaling to large-scale memories. To achieve the memory capacity scaling, DRAM-NVMM systems adopt the NUMA architecture. Each memory node incorporates multiple memory DIMMs connecting to the iMC through independent channels. Existing key-value stores lack sufficient support for multi-DIMM, multi-node memory architecture.

First, multi-DIMM allows channel-level parallelism (CLP). Intel DCPMM utilizes CLP to interleave data across memory devices in a page granularity [81], which is widely used in current key-value stores [5, 14, 41, 42]. However, this coarse-grained data striping mode cannot fully exploit the channel-level parallelism. We perform an experiment to confirm it. For every thread, we create a 24GB memory region spread across six NVMMs ($NVMM_0$-$NVMM_5$) with a specific striping size. Threads read data from devices in a round-robin manner ($NVMM_0 \rightarrow \cdots \rightarrow NVMM_5 \rightarrow NVMM_0 \cdots$). The experiment varies the striping size and the thread number. Figure 2b suggests that the total throughput is higher for a small data striping size.

We explain such parallelism gain as follows. Modern CPUs support having multiple independent memory requests in-flight [12], which makes room for utilizing CLP. Yet, only a small number of consecutive memory access can be served in parallel due to re-order buffer (ROB) and line fill buffer (LFB) capacity limitations in the processor [69, 76, 85]. Thus, smaller striping size makes outstanding successive instructions more likely to be sent on multiple channels, resulting in higher bandwidth utilization. When the thread number is above sixteen, threads with different striping sizes deliver similar latency due to the small NVMM bandwidth.

Second, previous research works report that remote NVMM access is quite expensive [30, 42]. The performance gap between remote and local memory access exceeds over 40% [42]. Currently, only ListDB [41] and PACTree [42] provide NUMA-aware memory allocation. The braided skiplist in ListDB spans list nodes across NUMA nodes. However, it ignores remote list nodes during data lookup. As list nodes are unevenly distributed, ListDB cannot achieve balanced search performance.

For DRAM key-value stores, delegation [7, 33] and replication [8, 56] are two widely used methods to mitigate the NUMA impact. The delegation method uses a dedicated thread for handling data requests issued from clients. However, the centralized delegation design easily becomes a bottleneck in the large-scale memory environment [7]. The replication method requires data coherence protocols to achieve coherent, NUMA-aware data access. Existing protocols are often write-intensive as they introduce many data writes for invalidating or synchronizing remote copies when data update happens [8, 27, 56]. These additional memory writes are unfriendly to NVMM.

Finally, how to scale to multi-DIMM, multi-node memory systems by exploiting the hardware parallelism and minimizing the remote memory access impact is the third challenge.

3 BONSAIKV OVERVIEW

We propose BonsaiKV, a fast, scalable, persistent key-value store on DRAM-NVMM memory systems. Figure 3 presents BonsaiKV architecture. It exhibits a tri-layer hierarchical system architecture

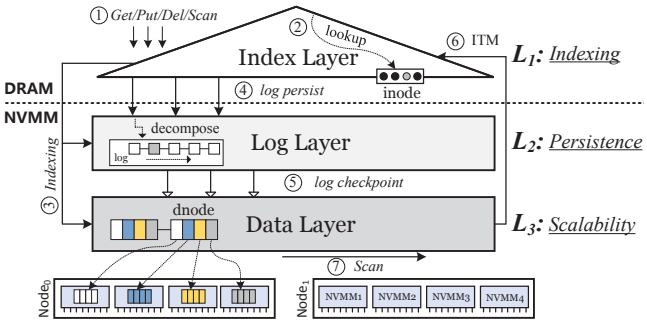


Figure 3: BonsaiKV Tri-Layer Hierarchical Architecture

that spans a tiered, heterogeneous memory system equipped with multiple channel-independent DRAM and NVMM DIMMs.

BonsaiKV carefully tailors the KV data storage format in each layer. The basic KV data storage units in these layers are namely index node (*inode*), persistent log, and data node (*dnode*). These three layers are specialized in *efficient key-value indexing, fast data persistence, and large-memory scaling*. To achieve these aims, BonsaiKV introduces four novel design principles.

P1. Combining DRAM's high speed and NVMM's large capacity to achieve fast and memory-efficient indexing. BonsaiKV splits the indexing over two distinct memory tiers. It compensates for the DRAM's small capacity by offloading part of the indexing into the large NVMM and improves indexing performance by uploading small, lookup-critical metadata onto fast DRAM. Moreover, these two indexing tiers are optimized separately with hardware characteristics.

P2. Regulating data persistence granularity to address the NVMM congestion. We uncover a strong correlation between data persistence size and NVMM congestion with an analytical model. Then, BonsaiKV proposes NVMM congestion control mechanisms that regulate data persistence granularity to effectively address memory traffic interference and bandwidth contention challenges.

P3. Full utilization of CLP to parallelize data reading. BonsaiKV proposes a novel key-value data stripe design by exploiting the channel-level parallelism. Logically, it aggregates a number of key-value pairs to ease data management. Physically, strips of key-value pairs are spread across memory devices. It greatly benefits *scan* since data can be fetched in parallel through independent memory channels.

P4. Scaling to multi-node architecture (NUMA) by localizing data access. NUMA architecture poses severe challenges for key-value stores due to the non-uniform memory access property. BonsaiKV always localizes data writes. Also, it adopts an adaptive data replication mechanism to alleviate remote memory read costs. The data coherence is achieved with a write-optimal, self-invalidation-based protocol.

Data Flow across Layers. These three layers cooperate to serve four types of key-value data requests: *get/put/del/scan*. For all requests, they are first issued to the index layer (①). The index layer uses the requested key to perform a collaborative tiered lookup. It first finds the associated indexing entry in the DRAM tier (②). Then, the remaining key-value data indexing is performed in the

NVMM tier (③). It searches for the target data which are stored in either log or dnode.

For *put/del*, they create new logs in the log layer in ④. A *del* request has a tombstone value in the log. The log checkpoint thread flushes logs to the data layer in ⑤. After key-value data are moved from logs to dnodes, BonsaiKV inserts, updates, or removes associated indexing entries in the DRAM tier, i.e., indexing tier modification (ITM in ⑥). Finally, key-value pairs stored in the dnode deliver superior access parallelism with its data striping scheme, supporting fast *scan* (⑦).

4 DESIGN AND IMPLEMENTATION

4.1 Index Layer Design

4.1.1 Approach Overview. We propose collaborative tiered indexing to resolve the trade-off between DRAM consumption and indexing performance. We design a low-cost shim layer between DRAM and NVMM indexing tiers. It reduces the DRAM consumption by offloading part of the DRAM index to the NVMM tier in a transparent and incremental way. Due to index offloading, the whole indexing task is split into two parts. Most of the lookup path executes in the fast DRAM tier. To improve NVMM tier indexing performance, the indexing metadata are uploaded onto DRAM. Also, we parallelize data lookup with multiple NVMM devices.

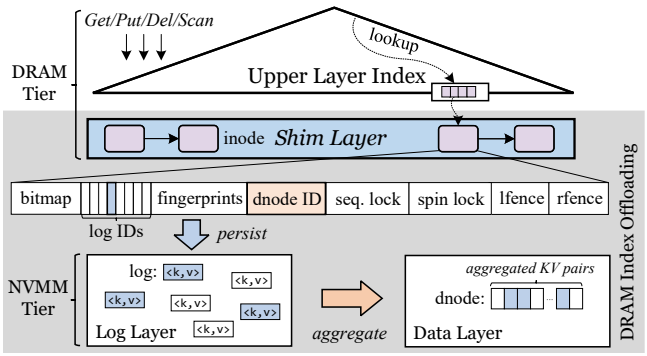


Figure 4: Transparent, Incremental DRAM Index Offloading

4.1.2 Offloading DRAM Index to NVMM Tier. In BonsaiKV, a key-value item *put/del* event generates a log at first. A volatile indexing entry is also created for this log. BonsaiKV uses an in-DRAM Masstree [54] to manage all indexing entries. Logs are transient. Log checkpoint migrates key-value pairs stored in distributed logs and aggregates them into dnodes. Aggregated key-value pairs in the dnode only need one DRAM indexing entry. Thus, the DRAM index size is reduced. Nevertheless, this index offloading is intrusive as tracking key-value pairs from logs to dnodes requires laborious modification to the DRAM index. To overcome this, we design a shim layer to realize transparent, efficient index offloading.

The shim layer is interposed underneath the DRAM index. It consists of a list of volatile inodes. The inode is responsible for tracking key-value pairs stored either in log or dnode. Figure 4 shows that an inode includes an unsorted array containing sixteen log IDs and a dnode ID. The bitmap is used to denote valid log IDs. It also contains a lfence and a rfence which are the minimal and maximal keys of this inode. BonsaiKV does not duplicate keys in the inode but leverages NVMM's large capacity to store keys inside logs. We set the inode size to two cache lines to utilize CPU adjacent cache line prefetch [12]. Every inode requires a pair [lfence, inode_addr] in the plug-in index atop the shim layer.

Initially, there is an empty inode in the shim layer whose lfence/rfence is $-\infty/+\infty$. Note that an inode insert does not alter lfence/rfence. When the inode is full, it is split into two inodes. Suppose the middle key of the full inode is γ. The key space of two inodes are $[-\infty,\gamma]$ and $[\gamma, +\infty]$, respectively.

After the log checkpoint, several key-value pairs are stored inside an m-fanout dnode in NVMM. When logs are flushed, associated log IDs in the inode are cleaned. If all logs of an inode are checkpointed, this inode could be freed to reduce DRAM pressure. The associated pair [lfence, inode_addr] in the upper layer DRAM index also can be removed. However, if the dnode's lfence is within this inode key range, we cannot free this inode. Otherwise, the DRAM index is unable to find this dnode anymore.

As the log checkpoint aggregates key-value pairs in NVMM dnodes, the DRAM index size is reducing increasingly. Moreover, *because inodes in the shim layer are responsible for tracking logs and dnodes dynamically, this incremental index offloading is transparent to the upper layer DRAM index.* In the best case, m key-value pairs only need an indexing entry in the inode, thereby reducing the DRAM space usage to $\frac{1}{m}$.

4.1.3 Collaborative Tiered Lookup. A *get* involves both DRAM and NVMM tier lookup. These two indexing tiers collaborate to serve the data lookup request.

DRAM indexing. The thread first searches the plug-in DRAM index for the last indexing entry whose key is less than or equal to the requested key. Then it reads the inode. An inode lookup could search either log or dnode. The inode uses the fingerprint mechanism for fast log lookup [61]. A fingerprint is a one-byte hash value of a key. An inode lookup computes the key fingerprint, checks the bitmap, and finds those log IDs with the same fingerprint. Then it passes log IDs to the NVMM indexing tier.

The plug-in index and shim layer can be viewed as a whole index. The concurrency control is split into two parts. The plug-in index uses its synchronization approach to search inodes. The inode lookup is lock-free. Coordination between readers and writers uses version lock for optimistic concurrency control [42]. We use a quiescent-state-based reclamation technique [32] to achieve safe inode access. In addition, writer-writer conflicts are resolved with a spin lock.

NVMM indexing. The log region is organized as a linear array. Given valid log IDs, NVMM indexing uses them to index logs in the array and compares keys with the requested key. If there is a match, it returns the key-value item. As fingerprint collision is rare [61], it guarantees that there is only one NVMM read to the key during lookup in common cases. For a collision, it reads those keys with the same fingerprint to determine which one is the target key. If there are no matched keys in logs, the remaining indexing is performed in the associated dnode. The dnode provides a striped data layout. It uses DRAM-resident metadata to parallelize entry lookup with multiple devices. Details are in §4.3.1.

4.2 Log Layer Design

The log layer uses a log-structured memory method. However, the small NVMM bandwidth prevents fast data persistence. This section first presents an analytical model to understand NVMM congestion. Then, we propose two effective mechanisms to address the NVMM congestion efficiently.

Figure 6 shows that every CPU has a log region. A log contains a 4-byte log size, an 8-byte value, or a 6-byte logical value offset and a 2-byte value size, an 8-byte timestamp, and a 1-byte log type. The variable-sized key is stored at the tail of the log. Large, variable-sized values are striped, which is described in §4.3.1.

4.2.1 NVMM Congestion Analytical Model. Let N and G denote the thread number and the data persistence granularity in bytes. Modern out-of-order execution instruction pipeline supports issuing multiple memory requests simultaneously. A memory request first queries the L1-D cache. If there is a miss, it puts this request in a microarchitectural buffer called line-fill buffer (LFB) [69, 76] and consults the lower cache hierarchy and the main memory.

Assume the CPU allows maximal k outstanding memory requests. For example, k equals 12 for Intel skylake microarchitecture [69, 76]. The cache line size is C. We define the data transmission rate (DTR) without any granularity restriction as v. The DTR v_G of data persistence granularity G is $\frac{\min(G, k \cdot C)}{k \cdot C} \cdot v$. The CPU only allows maximal k in-flight memory requests. The total DTR v_{total} of N threads is $\frac{\min(G, k \cdot C)}{k \cdot C} \cdot v \cdot N$. We define the data service rate of the NVMM medium as v_S.

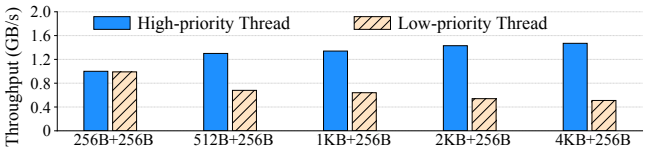


Figure 5: Thread Throughput with Different Data Persistence Granularities

From this model, we draw two important conclusions. First, the data transmission rate is proportional to the data persistence granularity G. The thread with a larger data persistence granularity delivers a higher data transmission rate. Consequently, hardware units like LFB, WPQ, and NVMM buffer are likely populated with its data. Hence, its memory requests are first served and the thread exhibits a higher priority.

We also conduct an experiment to confirm it. The experiment creates two concurrent running threads. They persist data to the same NVMM device. We fix the data persistence granularity of one thread as 256B and vary the other from 256B to 4KB. Figure 5 shows that the thread with a large persistence granularity yields higher throughputs. Second, the congestion is proportional to the total data transmission rate v_{total}. If v_{total} is greater than the NVMM service rate v_S, the NVMM device buffer is heavily contended and the data thrash begins [86].

4.2.2 NVMM Congestion Control Mechanism. This analytical model inspires two techniques that reduce traffic interference and mitigate bandwidth contention.

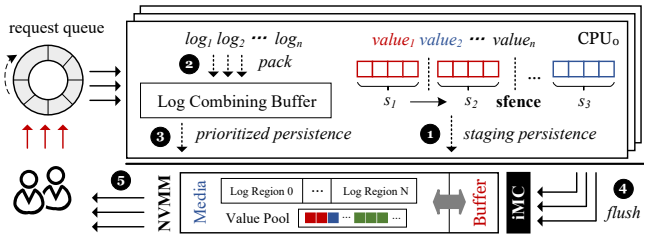


Figure 6: NVMM Congestion Control Mechanisms

Reducing NVMM traffic interference. There are two persistent data flows in BonsaiKV: (i) foreground log persistence caused by *put/del*; (ii) background log flush to the data layer. Log flush incurs unexpected memory interference for co-running foreground threads. We propose prioritized persistence technique to reduce NVMM traffic interference.

From the analytical model, we find that a larger data persistence granularity yields a higher priority. We design a private 2KB volatile log combining buffer (LCB) for every worker thread. The LCB is used to pack logs in ❷. The non-log-embedded key data are also stored in the LCB. Packed data are persisted at once in ❸. The background thread adopts a small persistence size to flush logs. Increasing the data persistence granularity for foreground threads reduces traffic interference from background threads. Another advantage of using log packing is reducing log metadata *read-after-persist*, which amortizes performance costs induced by *clwb* inefficiency [18].

Mitigating NVMM bandwidth contention. As stated in §2.2, large-sized value persistence causes heavy bandwidth contention. We propose staging persistence to address this.

From the analytical model, the NVMM bandwidth contention is positively correlated to the total data transmission rate $v_G \cdot N$. Reducing the thread DTR v_G is a useful method to avoid device buffer contention, thereby reducing the bandwidth contention. According to $v_G = \frac{\min(G, k \cdot C)}{k \cdot C} \cdot v$, thread DTR is proportional to the persistence granularity G. To reduce the data transmission rate v_G, the staging persistence technique uses *sfence* to divide the value data flow into a series of small, fixed-sized *stages* and persist data in each stage at a time (❶).

Every stage has a small persistence granularity. Moreover, the memory fence forces two consecutive stages s_1 and s_2 to obey a strict order. The next stage s_2 cannot begin until data in s_1 finishes persisting. Data from multiple threads' stages are flushed to the NVMM device in ❹. We deliberately slow down the data transmission. Reducing DTR prevents data bursts from happening in the device buffer and thus avoids data thrash.

Another contention mitigation method is throttling parallel executing threads (i.e., reducing N in v_{total}) [86]. Section §5.3 shows that this method is heavy-weight for key-value stores due to non-negligible communication overheads.

Crash consistency. BonsaiKV supports durable linearizability (DL) [21, 26, 37], which is an acknowledged correctness condition for NVMM data structures [26, 35, 42] and key-value stores [14, 41]. In a client-server model, multiple clients issue requests to the server and wait for responses. A worker fetches requests and puts logs

into the LCB. To ensure the correct persist order, values are made durable before keys. Only after logs in LCB are persisted atomically, the worker notifies clients in ❺. These operations become visible and complete. If a crash happens before log persistence, these in-flight requests will be abandoned. Although unfinished requests lose their data, the *all-or-nothing* condition in durable linearizability is still ensured.

Discussion. In realistic workloads [4, 9, 82], key sizes are usually small, whereas value sizes have a wide distribution. For large-sized value writes, the major performance bottleneck is bandwidth contention. Thus, the staging persistence is effective. On the other side, when value sizes are small, write persistence priority is more important. We store values along with keys in the LCB to ensure the high priority of data persistence.

4.3 Data Layer Design

The data layer scales out to multi-DIMM, multi-node memory systems with fine-grained data striping exploiting channel-level parallelism and NUMA-aware data management.

4.3.1 Exploiting Channel-Level Parallelism. As shown in Figure 7, we partition a dnode into multiple fixed-sized *strips* (256B for our platform) including a meta strip and several data strips. The total number of strips is aligned with the number of memory channels per CPU (six in this paper). The meta strip contains a 40-element fingerprint array, a bitmap, and other metadata. A data strip contains eight entry slots. A dnode entry contains a 24-byte key, or an inline 16-byte key and an 8-byte pointer to the remaining key data, an 8-byte value, or a 6-byte logical value offset and a 2-byte value size, and a volatile 2-byte epoch used in data coherence protocol. The order array records the dnode entry order.

Each NVMM device has a memory pool which is partitioned into a number of 256-byte blocks to store either meta or data strips. Figure 7 depicts that six strips belonging to a dnode reside at the same offset within every device's memory pool. Strips belonging to a dnode are spread randomly across all NVMM devices within a NUMA node, which avoids bottlenecked memory devices due to hot data access.

Large values are also striped. Values are organized linearly in a logical address space. Their logical address offsets and sizes are kept in the logs or dnode entries. Physically, the value is partitioned

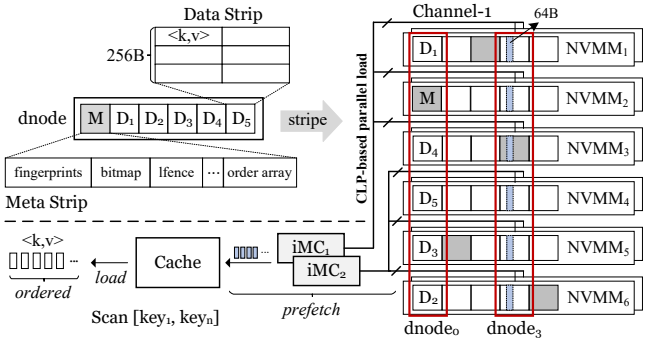


Figure 7: Fine-grained Data Striping by Exploiting Channel-level Parallelism

into *m*-byte units, which are spread across *N* NVMMs in a round-robin manner. Every device has a value pool for storing fixed-sized units. *m* and *N* are 256 and 6 in our paper. Besides, small-sized keys dominate in realistic workloads [4, 9, 82]. Hence, most of the keys are dnode-inline. Unlike values, We manage overflow key data using a slab-based allocator instead of striping them.

Distributed dnode lookup. A dnode lookup needs to find both meta and data strips in the different devices. We fully utilize available NVMM devices during entry lookup. Each dnode has a 4-byte ID in the inode, as shown in Figure 4. This ID is split into two parts: a 3-bit NUMA memory node ID and a 29-bit strip index. Shown in Figure 8, the thread locates the memory node (❶) and uses the hashed strip index to find the strip permutation array entry (❷). The global strip permutation array is read-only, non-volatile, and replicated by all memory nodes. It records the device IDs of meta and data strips.

Figure 8 shows that the meta strip is stored in the sixth NVMM of NUMA node$_1$. Suppose a node has six NVMMs. The memory device of the meta strip is NVMM$_{11}$ (❸). The dnode strips are organized as a linear array. The thread uses the strip index to read the metadata strip (❹). This NVMM meta strip access procedure only happens once. We duplicate the fingerprint and the bitmap in the DRAM. As a consequence, the subsequent dnode metadata accesses happen in the DRAM.

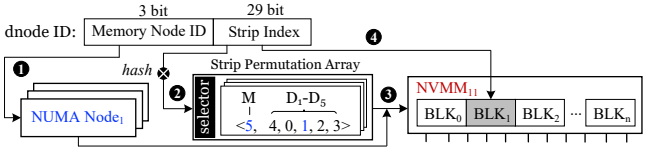


Figure 8: Dnode Meta Strip Access Procedure

Key fingerprints are stored in the meta strip. For the requested key-value pair, the thread calculates its fingerprint to find the fingerprint array index *i*. The data strip number is i/n (n is the number of data strip entries). It uses i/n as the index to read the strip permutation array entry to get the device ID. Then, similar to step ❹, it locates the data strip in the device with the NUMA node ID and the strip index encoded in the dnode ID. Finally, it fetches the $(i\%n)$th entry in the data strip which contains the target key-value pair.

If there are fingerprint collisions, candidate dnode entries are fetched in parallel if they belong to different data strips. If the dnode entry contains a value offset, we use this value offset to fetch all units from the value pool. Given a logical value offset and the value pool base address, the device number and the start memory address of the kth value unit are $((offset + mk)/m)\%N$ and $base + (offset + mk)/N$, respectively.

Parallel range query. Dnode entries are unordered. However, a *scan* should return a sorted entry array. To address this, we propose a *two-phase* parallel strip scan technique. This technique maximizes the memory channel utilization during range queries.

Suppose a strip has m 64-byte data (m = 4 in our 256B-striping setting). As shown in Figure 7, reading a dnode uses prefetch instructions to load kth 64-byte data of all data strips into the CPU cache from different channels first. It repeats this with the next 64-byte data in all strips until k reaches m. This phase preloads strip entries

from different memory devices into the CPU cache. Because the NVMM access latency is much longer than the CPU cache access, we remove this bottleneck by parallelizing long NVMM medium access using the channel-level parallelism. Then, it reads sorted key-value pairs from the cache using the order array. Besides, a *scan* also reads all valid logs of this inode. It continues reading until the key exceeds the given range.

4.3.2 NUMA-aware Data Management. The data layer provides adaptive data replication to localize data read. If there is a remote read to a dnode entry, readers perform a hot key identification using the count-min sketch method [38]. If it is a hot entry, readers copy the dnode entry to their local memory nodes. As a result, a dnode entry could have $N-1$ replicas at most for N memory nodes. Accessing these $N-1$ replicas requires $N-1$ memory addresses. To avoid the extra memory usage, the data layer provides a uniform way for threads running on different memory nodes to access the dnode entries.

Section §4.3.1 presents a striped dnode layout across devices within a memory node. Further, every memory node uses the same dnode striping setting, and every dnode strip resides at the same memory offset in each NVMM. Therefore, wherever a dnode entry is copied to which memory node, readers can use the same procedure in the Figure 8 to access a dnode entry except using a different NUMA node ID.

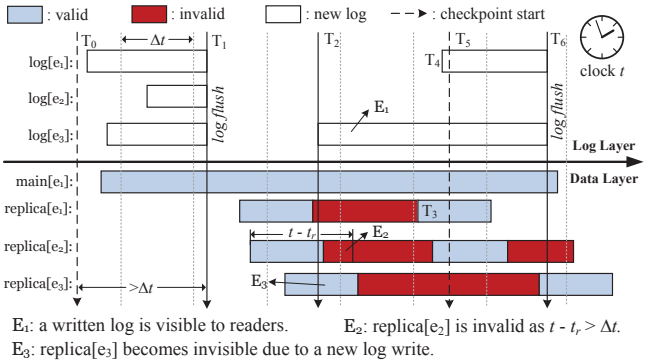


Figure 9: Illustrative Examples of WoSI Protocol

Data coherence. Data replication requires maintaining coherence between the main data and its replicas. Existing protocols incur excessive data writes for maintaining the replica coherence. We propose Writer-optimized Self-Invalidation (WoSI) protocol to achieve write-optimal data coherence.

Analogously to self-invalidation techniques [44, 45], WoSI requires each memory node to maintain data validity by itself without any cross-node communication or global coordination [8, 56, 63]. Figure 9 illustrates the protocol. The main version and the replica of entry e are denoted as $main[e]$ and $replica[e]$, respectively. There is a global clock t. Each replica has a local timestamp t_r recording its fetch time. Readers first check the replica timestamp before accessing the data. If $t - t_r > \Delta t$ where Δt is the valid time period threshold, it indicates that the replica expires and readers should re-fetch the data (E_2).

Nevertheless, self-invalidation techniques suffer from low writer performance [71], i.e., writers have to stall until all replicas expire. WoSI solves this by *leveraging the log layer to separate new data writes from old data versions*. For example, at T_2, although there is a $replica[e_3]$ for e_3 in the data layer, the thread still can write the new data into a log. BonsaiKV updates the inode in the shim layer to make the new data visible to readers (E_1) and causes $replica[e_3]$ to become invisible (E_3). Thus, writers proceed without waiting for all replicas to become invalid, and no coherence is violated.

Additionally, log checkpoints may raise data coherence violations. For instance, a $replica[e_1]$ is created at T_3. Then, a new $log[e_1]$ for e_1 is created at T_4. The log checkpoint begins at T_5. If we flush the log to $main[e_1]$ but $replica[e_1]$ is still valid, it causes a data version conflict between $main[e_1]$ and $replica[e_1]$. To deal with it, we set the time interval between the log checkpoint start time and the log flush start time greater than Δt.

4.3.3 Log Checkpoint and Recovery. A checkpoint starts when the log number exceeds the threshold. The worker threads fetch logs into DRAM and merge them with timestamps. Worker threads group logs into clusters. A cluster has a dnode and several logs whose keys are within this dnode key range. The master thread balances dnodes among all memory nodes and assigns clusters to workers. Workers perform *conflict-free*, *localized* data writes on disjoint dnodes in the log flush.

During a log checkpoint, only log flush affects system states. Its crash consistency is achieved with three key points. First, logs are removed only when they are flushed. Second, the dnode entry insert and remove are idempotent. Inserting or removing an entry multiple times has no side effect. Third, the log flush order makes no difference during recovery since a key-value pair only has a newest data version after the log merging. During recovery, BonsaiKV rebuilds the volatile shim layer and the DRAM index and redoes the log checkpoint.

4.4 Discussion and Future Usage

Currently, BonsaiKV is designed and implemented on a regular NUMA machine with both DRAM and NVMM. However, its techniques are generalized and applicable to other tier memory systems like CXL-connected heterogeneous memories [55].

Specifically, DRAM cell scalability almost reaches the physical limitation [64], whereas surging storage class memories, such as Z-NAND [1] and PCM [66], are much denser and scale up to terabytes. Hence, the performance and capacity gaps still exist, which are tackled by our collaborative tiered indexing.

Second, the rate improvement of memory is less than that of microprocessor [83]. More severely, growing data-intensive applications stress the memory performance [49]. This hardware evolution trend and application demand exacerbate the memory congestion issue. BonsaiKV uses data packing and *sfence*-based congestion control techniques to address this challenge. Because the real CXL-based memory system exposes the same interface as the existing DRAM-NVMM system [74], our persistence techniques are generalized to such systems.

Finally, PCIe-connected CXL devices deliver superior bandwidth than DRAM via lane-level parallelism (~40GB/s for DDR5 DRAM vs. ~64GB/s for PCIe Gen5 ×16 [39]). Similar to the DIMM-based NUMA

architecture, the I/O controller is integrated into the processor chip. It also causes non-uniform access speed to PCIe devices plugged in different physical domains [72]. Exploiting hardware-provided parallelism and alleviating non-uniform access impact is still important. Therefore, our scalable techniques are also applicable.

5 EVALUATION

5.1 Experiment Methodology

Testbed machine. Experiments are performed on a dual-socket Intel Optane DC Persistent Memory machine. There are two Intel Xeon Gold 5220R processors. Each processor has twenty-four physical cores running at 2.20GHz with hyper-threading disabled. Each chip has two memory controllers and six memory channels. This machine has a total of 1.5TB (12×128GB) DCPMM in *fsdax* mode and 192GB (12×16GB) DDR4 DRAM. Besides, it also has a 1TB Samsung solid-state drive. We use the directory-based cache coherence protocol in the experiment. The operating system is Ubuntu 18.04 with Linux kernel version 5.4.0. The compiler is GCC-8 with -O3 optimization. We use jemalloc [24] for DRAM allocation for all key-value stores.

Compared solutions. We compare BonsaiKV with (1) three NVMM key-value stores: LSM-tree-based ListDB [41], log-structured FlatStore [14, 77], hash-based Viper [5] and (2) three persistent index structures: trie-based PACTree [42], B^+-tree-based FAST-FAIR [35], and hybrid index DPTree [87]. We do not compare BonsaiKV with LB$^+$-tree. LB$^+$-tree is unstable due to incomplete software fallback mechanism [42, 52].

We create a 4KB-interleaved NVMM region across six devices in a socket for all key-value stores except BonsaiKV. For NUMA-unaware key-value stores and indexes, we use the device mapper tool to create a striped device across all interleaved NVMMs [41]. The lookup cache in ListDB is set to 1GB. We also configure Masstree as the DRAM index for FlatStore. We set the user/worker thread number ratio to two for ListDB, DPTree, and BonsaiKV.

Table 2: YCSB Workload Configuration

	Workload	K/V Size	Distribution	Preload	Ops
Integer	Load, A, C, D, E	8B/8B	Uniform	5M	2.5M
String	Load	24B/16KB	Uniform	240K	120K
	A	24B/16KB	Zipfian	240K	120K
	C, D	24B/1KB	Zipfian	0.5M	5M
	E	24B/8B	Uniform	5M	0.5M

Benchmarks. We create a set of microbenchmarks to evaluate BonsaiKV core techniques. We also use the synthetic YCSB benchmark suit [15] for deep experimental analysis. Table 2 lists the configurations of various YCSB workloads. The operation number in Table 2 is per-thread. Every key-value store is populated with a number of key-value records before the experiment. We use the index-microbench to generate YCSB workload traces [15, 78]. To stress the NUMA impact, we evenly distribute all threads across two sockets. BonsaiKV enables data replication for skewed access distribution experiments. For BonsaiKV, its data validity threshold in the WoSI protocol is set to five seconds.

5.2 Indexing Technique Evaluation

We evaluate the performance and the DRAM consumption of three variants of collaborative tiered indexing. All variants use the same Masstree implementation. The first approach (BonsaiKV+DRAM-only) is similar to FlatStore [14]. It creates a DRAM indexing entry for each key-value pair. The second approach (BonsaiKV+data offload) offloads the data storage and the part of indexing into NVMM. The last approach uploads lookup-related metadata onto DRAM. This experiment uses YCSB-C workload with 100% of *get*. Each thread inserts 150 millions of 16-byte records.

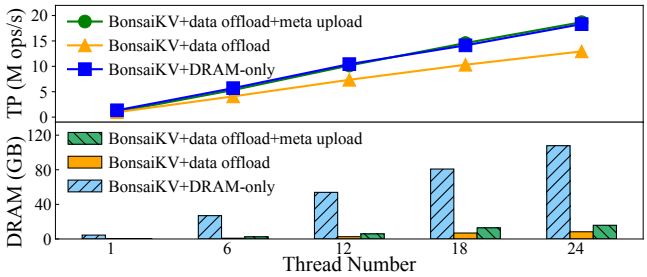


Figure 10: Evaluating Indexing Techniques

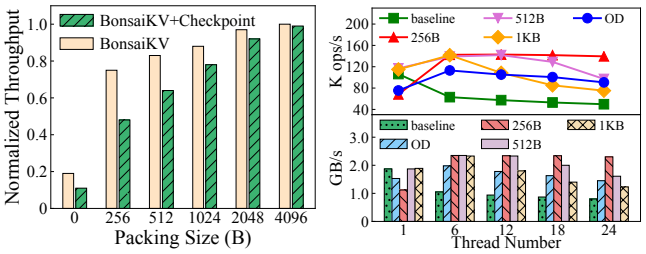
Figure 10 shows that offloading indexing from DRAM to NVMM degrades the throughput by up to 44%. Retaining lookup-related metadata in DRAM closes the performance gap and delivers similar performance as the DRAM-only approach. It even slightly outperforms BonsaiKV+DRAM-only. The reason is the lookup-friendly dnode design. Both of them only induce an essential NVMM read to the entry during lookup. Differently, Masstree uses an ordered leaf node layout. A node lookup requires an expensive linear search, whereas our dnode design performs $O(1)$ entry lookup.

DRAM-only indexing causes heavy memory pressure. Figure 10 shows that indexing 3.6 billion eight-byte key-value pairs (~53GB) consumes nearly 107GB DRAM. In contrast, BonsaiKV+data offload solution only costs 8.3GB DRAM. The DRAM-resident metadata consumes 7GB more memories. Overall, our indexing technique saves 6.9× DRAM space while achieving comparable performance as the DRAM indexing.

5.3 Persistence Technique Evaluation

This section evaluates performance and effectiveness of our persistence techniques. Every thread preloads five million eight-byte key-value records. Afterward, every thread performs five million update operations. Experiments run twelve user threads and six checkpoint threads in a socket. For experimental comparison, we also disable the log checkpoint and vary the data packing size to study its effect.

Figure 11a shows that throughput increases and gradually approaches that without checkpoint when the packing size becomes large. It suggests that a large packing size in the prioritized persistence technique decreases the interference from the log checkpoint and increases the log persistence priority. Figure 11a also demonstrates that the log packing improves throughput by up to 5.15× over no packing setting due to zero write amplification and the reduced number of cache line *read-after-persist*.



(a) Prioritized Persistence Effect **(b) Staging Persistence Size Effect**

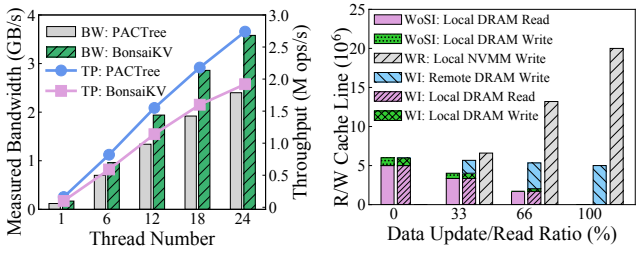
Figure 11: Evaluating Persistence Technique

Then, we analyze the performance of different staging persistence sizes. We use the YCSB-Load workload in this experiment. Every thread inserts 240K records with 16KB value. We vary both thread number and staging persistence size. The baseline disables the staging persistence technique. For small thread numbers, the NVMM bandwidth contention is low. Thus all configurations achieve the similar performance in Figure 11b. The baseline throughput decreases dramatically when the thread number increases. For large thread numbers, the 256-byte staging persistence size performs best. As the contention level is proportional to the DTR, a small persistence size reduces the transmission rate effectively and prevents device buffer contention.

OdinFS uses opportunistic delegation (OD) to limit the parallel thread number [86]. We implement it in BonsaiKV. Figure 11b shows that the opportunistic delegation outperforms the baseline. However, profiling results show that this technique suffers from severe communication overheads for the ring buffer cache thrash. This technique is more applicable to file systems instead of key-value stores. It is because file system calls have lengthy execution paths which can amortize extra overheads.

5.4 Scalability Technique Evaluation

We evaluate the parallelism benefits of our data striping. We compare BonsaiKV with PACTree. PACTree organizes entries in a tree node as a linear array. We use the YCSB-E workload with 24-byte keys and 8-byte values. Figure 12a shows sequential entry scan in PACTree is unable to fully utilize all NVMM bandwidth. BonsaiKV yields a 49.04% better bandwidth utilization and 42.71% higher throughputs than that of PACTree.



(a) Stripe and Non-stripe Performance Comparison **(b) Coherence Protocol Overhead Comparison**

Figure 12: Evaluating Scalability Technique

We compare WoSI with another two typical data coherence protocols: Write-Invalidation (WI) in ccKVS [27] and Write-Update (WR) in HydraList [56]. We run a thread per socket and vary its read/update operation number ratio. Each thread performs five million operations. We use the number of R/W cache lines to measure the protocol overhead and break it down into sub-categories. Figure 12b shows that the WoSI protocol introduces local timestamp reads and few writes for timestamp updates. WI protocol also incurs local DRAM access for maintaining the state transition for each entry like MESI protocol [63]. Besides, remote DRAM writes also increase for propagating messages to invalidate remote copies. The WI protocol induces heavy NVMM writes because it synchronizes all replicas when a data update occurs.

5.5 Sensitivity Study

Key length. In this experiment, we run YCSB-Load, YCSB-C, and YCSB-E workloads and vary the key length. The value size is eight bytes. The variable-sized keys are efficiently supported by our log-structured method. Particularly, a key-value *put* in the YCSB-Load workload creates a log and writes it sequentially to the device. As shown in Figure 13, for different key lengths, sequential writes have no performance impact on the *put* performance. When the key size is larger than 48 bytes, the throughput degrades due to small NVMM bandwidth limitations.

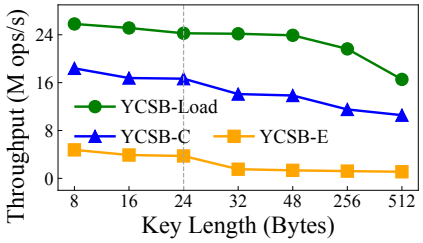


Figure 13: Key Length Sensitivity Study

YCSB-C and YCSB-E workloads issue massive data *get* and *scan* requests, respectively. Before this experiment, we flush all logs to dnodes. Therefore, for these two workloads, either a *get* or *scan* reads data from the dnode. When the key size exceeds 24 bytes, Figure 13 shows that workload throughputs decrease owing to extra NVMM reads to pointers and pointer deference costs. Figure 13 also shows that the *scan* performance is especially hurt due to more intensive pointer deferences.

Table 3: DRAM and NVM Space Consumption (GB) of Five Key-Value Stores

# Thread	DPTree	ListDB	FlatStore	Viper	BonsaiKV
1	0.01/0.02	1.03/0.14	0.15/0.11	0.21/0.08	0.02/0.09
24	0.14/2.34	1.07/4.18	3.59/2.72	4.96/1.92	0.53/2.38
48	0.35/4.51	1.13/8.11	7.19/5.58	10.59/3.81	1.06/4.83

Memory consumption. We measure the DRAM and NVMM consumption of five key-value stores. Table 3 lists DRAM and NVMM consumption after loading five million 16-byte key-value

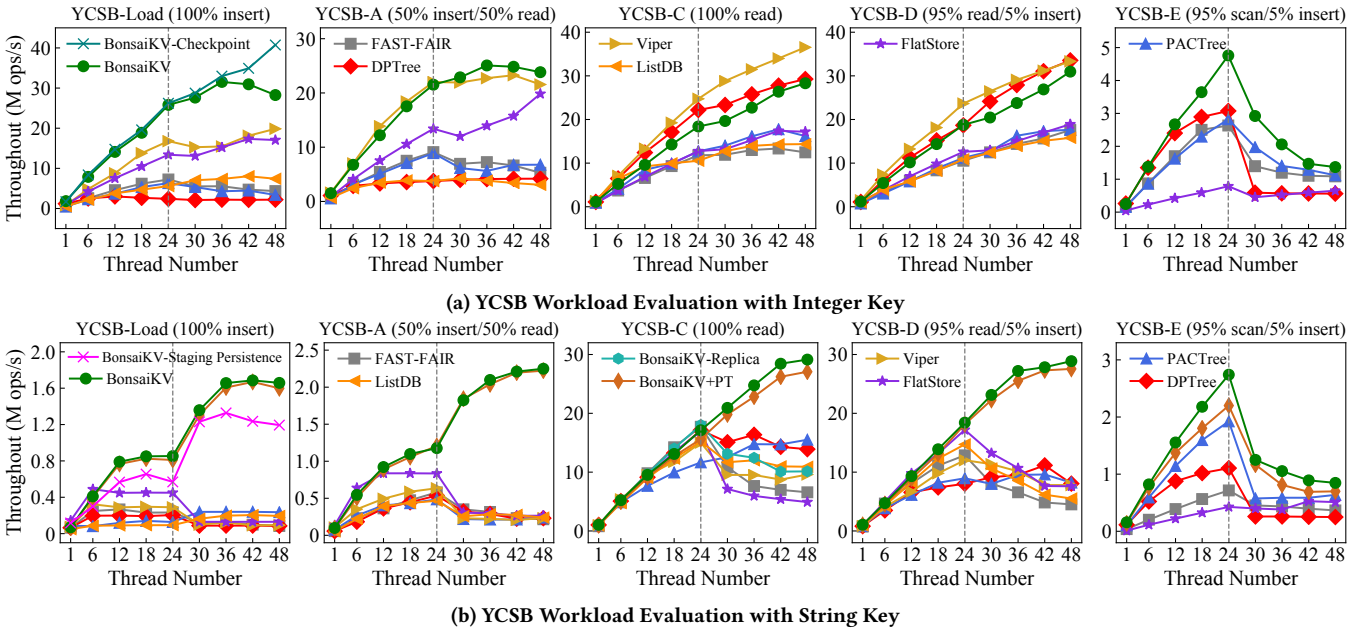


Figure 14: YCSB Workloads Evaluation Results. *BonsaiKV denotes the proposed full-fledged key-value store. BonsaiKV-Checkpoint, BonsaiKV-Staging Persistence, and BonsaiKV-Replica denotes a full-fledged BonsaiKV disables log checkpoint, staging persistence, and data replication, respectively. BonsaiKV+PT stores the entire string key outside the dnode and uses a pointer to access the key.*

pairs per thread into each key-value store. ListDB achieves constant DRAM consumption for its fixed-size lookup cache setting. FlatStore and Viper use 6.78× and 9.99× more DRAM space than BonsaiKV. DPTree is DRAM frugal due to its split tree structure and large leaf node fanout.

Besides key-value data, every log in ListDB has an eight-byte pointer and an eight-byte version. Thus, ListDB has the highest memory space amplification. Viper consumes the smallest NVMM space due to condensed data layout and per VPage metadata. Other key-value stores occupy a similar NVMM space.

5.6 Write-intensive Workload

We use YCSB-Load workload to stress the *put* performance. We first describe the integer key experiment results. Figure 14a shows that BonsaiKV achieves a linear speedup when the thread number increases and outperforms others significantly. As the worker thread number is twelve, its throughput reaches a peak at 36-thread for CPU competition between user and worker threads. We disable the log checkpoint in BonsaiKV-Checkpoint. When the user thread number is smaller than 36, its throughput is close to that of BonsaiKV. It demonstrates that our prioritized persistence technique effectively prevents the performance interference from background worker threads.

BonsaiKV performs 2× better than hash-based Viper. We find three bottlenecks in Viper. First, Viper data layout design is NUMA-unaware. Second, its VPage metadata are frequently updated, causing heavy cache line thrash due to *clwb* inefficiency. Third, its CCEH implementation [59] incurs severe segment lock contention. FAST-FAIR also suffers from similar performance issues in its tree node metadata management. Besides, FAST-FAIR adopts an in-order key design, which introduces costly key movements during node updates. Seven-phase data merging in DPTree causes high background performance costs. PACTree uses a trie [46] in its search layer. It incurs excessive small, random accesses, leading to heavy NVMM write amplification.

LSM-tree-based ListDB provides fast key-value *put* by just inserting an entry into the MemTable in the DRAM and creating a write-ahead log in the NVMM. ListDB is slower than log-structured FlatStore and Viper for its inefficient skip list walk during MemTable insert. Surprisingly, BonsaiKV performs much better than FlatStore. The *put* execution path in these two key-value stores is similar: creating a log, performing a Masstree tree walk, and inserting an indexing entry. Differently, BonsaiKV offloads the index to the NVMM tier, which effectively reduces the Masstree size. FlatStore has a higher Masstree, which leads to a relatively longer tree walk during *put*.

For string key experiments, most key-value stores' throughputs stop increasing at 6-thread and experience sharp decrement beyond 24 threads. Low NVMM bandwidth and remote value write are two major reasons. ListDB and PACTree are NUMA-aware. Their throughputs continue to increase slightly. BonsaiKV reaches a peak at 18-thread and delivers stable performance thanks to its staging persistence. Furthermore, its throughput still grows beyond one socket for its localized log writes.

We also disable the staging persistence technique. Experiment results show that high NVMM bandwidth contention degrades BonsaiKV's throughputs greatly. Furthermore, interleaving NVMM devices increases bandwidth contention as a single NVMM is accessed by all threads. In BonsaiKV, four log regions share an NVMM device,

which also helps alleviate the bandwidth contention. It accounts for the remaining performance gap. BonsaiKV+PT' throughputs are close to BonsaiKV's. Similar to fixed-sized key writes, variable-sized key persistence is also NVMM-friendly, sequential writes.

5.7 Read-intensive Workload

Both YCSB-C and YCSB-D are read-intensive workloads. Every thread issues 10× more *get* requests than the preload operation number to fully stress the data indexing performance. FlatStore is 1.65× slower than BonsaiKV. The reasons are twofold. First, BonsaiKV's dnode fanout is 3× wider than Masstree's leaf node fanout, resulting in a smaller tree height and shorter lookup path than that of FlatStore's Masstree. Second, BonsaiKV keeps fingerprints in the fast DRAM, which effectively overcomes NVMM indexing performance shortcomings.

FAST-FAIR has high linear search overheads in its node lookup caused by the large tree node read and unscalable NVMM. ListDB introduces a hash-based DRAM cache to avoid NVMM skip list lookup. Even so, *get* of ListDB suffers from expensive cache misses. The poor data locality and pointer-chasing accesses caused by the NVMM skip list degrade the lookup performance. Although PACTree uses fast ART [46] as the index structure, NVMM's poor performance impedes its data lookup efficiency.

The base tree in DPTree is a hybrid index that puts the upper layer in the DRAM. It has a wide leaf node fanout with a low tree and uses a hash table to manage leaf node entries. A *get* in DPTree only introduces one cache line fetch from the NVMM. BonsaiKV uses the volatile fingerprint design. A *get* incurs one more data read to the DRAM. Moreover, its index tree is higher than DPTree. These two reasons account for their small performance difference. Viper shows a 28.84% higher throughput than BonsaiKV. Because Viper uses a hash table for the DRAM index, its data indexing speed is higher than that of the Masstree-based BonsaiKV due to fewer DRAM accesses.

Next, we run workloads with a 1KB string value using Zipfian distribution. All systems are bottlenecked by the data reading and achieve similar throughputs. When the thread number exceeds 24, NUMA-unaware key-value store throughputs decrease dramatically because of costly remote data reads. BonsaiKV throughput scales linearly thanks to the adaptive data replication. BonsaiKV-Replica disables the adaptive data replication. When the thread number exceeds 24, massive remote reads significantly hurt its performance. BonsaiKV+PT performs slightly worse than BonsaiKV. Because the major bottleneck is large-sized data reads, extra pointer deference overheads are negligible.

5.8 Scan-intensive Workload

YCSB-E consists of 95% *scan* and 5% *put*. ListDB and Viper have no *scan* implementations. FlatStore is a log-structured key-value store, so it exhibits extremely poor data locality. Scanning a range of key-value pairs in FlatStore causes many small, random reads which are suboptimal NVMM access patterns. DPTree and FAST-FAIR use pointers to access string keys. PACTree and BonsaiKV adopt an inline key design. Reading a range of entries has a smaller memory footprint and better locality. Moreover, striped data layout in BonsaiKV further enables parallel data read for *scan*.

When the thread number exceeds 24, all key-value stores are bottlenecked by remote reads. The pointer-based key design has a bigger performance impact on YCSB-E. Other workloads spend more time on data indexing or persistence. Pointer deference time occupies a smaller portion of the total execution time.

5.9 Read-Write Mixed Workload

For YCSB-A workload, DPTree is slow due to poor insert performance. A *put* first inserts the key-value pair into the buffer tree, and then data are merged into the base tree. Its multi-phase merge causes high garbage collection costs, causing write stalls. FlatStore achieves balanced *get/put* performance for its log-structured data layout and volatile index design. BonsaiKV still outperforms FlatStore by up to 1.8× at peak performance due to its superior data indexing and persistence technique.

BonsaiKV achieves comparable throughputs as that of Viper. Even though its indexing performance lags behind Viper, its superior data persistence performance compensates for this weakness. In ListDB, searching the L_0 requires many PMTable lookups, which exacerbates its slow *get* performance.

When the experiment uses a large 16KB value, key-value store performance is dominated by the value persistence. DPTree, FAST-FAIR, ListDB, Viper, and PACTree achieve similar performance. FlatStore outperforms them for its lazy-persist allocator design. Its allocation policy avoids frequent metadata persistence overheads. Nevertheless, its throughput increment stops at 12-thread for NVMM bandwidth limitation. In contrast, staging persistence in BonsaiKV mitigates bandwidth contention, and its throughput increases continuously.

When the thread number is greater than 24, all key-value store throughputs experience a shrink beyond one socket. Again, BonsaiKV throughput scales irrespective of NUMA impact. Localized log writes and adaptive data replication effectively reduce remote data access. However, its throughput stops increasing at 36-thread for small NVMM bandwidth.

6 CONCLUSION

This paper pinpoints and analyzes three major challenges that prevent exploiting tiered heterogeneous memory systems for key-value stores. We describe the design and implementation of BonsaiKV, a fast, scalable, and persistent in-memory key-value store, with a range of novel and efficient techniques to design BonsaiKV hierarchical storage architecture. Extensive experiments demonstrate BonsaiKV reaps heterogeneous memory strength and yields significant performance gains.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We thank Andy Rudoff for his valuable discussion of Intel Optane Persistent Memory. This project is supported by the National Natural Science Foundation of China (Grant No. 61832005, Grant No. 62102131), the Natural Science Foundation of Jiangsu Province (Grant No. BK20220973, Grant No. BK20210361), and the Future Network Scientific Research Fund Project (Grant No. FNSRFP-2021-ZD-7). Baoliu Ye is the corresponding author.

REFERENCES

[1] 2017. Ultra-Low Latency with Samsung Z-NAND SSD. https://semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf.

[2] 2022. Intel Optane Persistent Memory 200 Series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[3] 2022. Redis. https://redis.io/.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-Value Store. In *International Conference on Measurement and Modeling of Computer Systems*. 53–64.

[5] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *USENIX Symposium on Operating Systems Design and Implementation*. 753–768.

[7] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *International Conference Distributed Systems*, Vol. 8304. 83–97.

[8] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 207–221.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies*. 209–223.

[10] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *International Conference on Management of Data*. 1087–1098.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *USENIX Conference on File and Storage Technologies*. 17–32.

[12] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance through Prefetching. In *International Conference on Data Engineering*. 116–127.

[13] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2634–2648.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*. 143–154.

[16] Ian Cutress. 2021. Using a PCIe Slot to Install DRAM: New Samsung CXL.mem Expansion Module. https://www.anandtech.com/show/16670/using-a-pcie-slot-to-install-dram-new-samsung-cxlmem-expansion-module.

[17] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *International Conference on Management of Data*. 339–351.

[18] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 115–127.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM Symposium on Operating Systems Principles*. 205–220.

[20] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX Symposium on Networked Systems Design and Implementation*. 395–408.

[21] Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2023. The Path to Durable Linearizability. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 748–774.

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation*. 401–414.

[23] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *European Conference on Computer Systems*. 1–13.

[24] Jason Evans. 2006. A Scalable Concurrent Malloc (3) Implementation for FreeBSD. In *Proceeding of the BSDCan Conference*.

[25] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation*. 371–384.

[26] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-Volatile Memory. In *ACM Symposium on Principles and Practice of Parallel Programming*. 28–40.

[27] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *European Conference on Computer Systems*. 1–15.

[28] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2021. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *USENIX Annual Technical Conference*. 287–294.

[29] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *USENIX Symposium on Networked Systems Design and Implementation*. 649–667.

[30] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639.

[31] Shashank Gugnani and Xiaoyi Lu. 2021. DStore: A Fast, Tailless, and Quiescent-Free Object Store for PMEM. In *International Symposium on High-Performance Parallel and Distributed Computing*. 31–43.

[32] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *Journal of Parallel Distributed Computing* 67, 12 (2007), 1270–1285.

[33] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization Parallelism Tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 355–364.

[34] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo I. Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX Annual Technical Conference*. 967–979.

[35] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *USENIX Conference on File and Storage Technologies*. 187–200.

[36] Satoshi Imamura and Eiji Yoshida. 2020. FairHym: Improving Inter-Process Fairness on Hybrid Memory Systems. In *Non-Volatile Memory Systems and Applications Symposium*. 1–6.

[37] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *International Symposium on Distributed Computing*, Vol. 9888. 313–327.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM Symposium on Operating Systems Principles*. 121–136.

[39] Myoungsoo Jung. 2022. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). In *ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.

[40] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John K. Ousterhout. 2016. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *USENIX Annual Technical Conference*. 57–70.

[41] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *USENIX Symposium on Operating Systems Design and Implementation*. 161–177.

[42] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *ACM Symposium on Operating Systems Principles*. 424–439.

[43] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the Performance of Fast NVM Storage with uDepot. In *USENIX Conference on File and Storage Technologies*. 1–15.

[44] An-Chow Lai and Babak Falsafi. 2000. Selective, Accurate, and Timely Self-invalidation using Last-touch Prediction. In *International Symposium on Computer Architecture*. 139–148.

[45] Alvin R. Lebeck and David A. Wood. 1995. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*. 48–59.

[46] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful Indexing for Main-Memory Databases. In *International Conference on Data Engineering*. 38–49.

[47] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel Distributed Systems* 31, 1 (2020), 94–110.

[48] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance

In-Memory Key-Value Store with Programmable NIC. In *ACM Symposium on Operating Systems Principles*. 137–152.

[49] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 574–587.

[50] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *USENIX Annual Technical Conference*. 673–687.

[51] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX Symposium on Networked Systems Design and Implementation*. 429–444.

[52] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[53] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies*. 133–148.

[54] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *European Conference on Computer Systems*. 183–196.

[55] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 742–755.

[56] Ajit Mathew and Changwoo Min. 2020. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1332–1345.

[57] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[58] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. 2013. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *International Symposium on Computer Architecture*. 72–83.

[59] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *USENIX Conference on File and Storage Technologies*. 31–44.

[60] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *USENIX Symposium on Networked Systems Design and Implementation*. 385–398.

[61] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *International Conference on Management of Data*. 371–386.

[62] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems* 33, 3 (2015), 1–55.

[63] Mark S. Papamarcos and Janak H. Patel. 1984. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *International Symposium on Computer Architecture*. 348–354.

[64] Kyriakos Paraskevas, Andrew Attwood, Mikel Luján, and John Goodacre. 2019. Scaling the Capacity of Memory Systems; Evolution and Key Approaches. In *International Symposium on Memory Systems*. 235–249.

[65] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *USENIX Symposium on Operating Systems Design and Implementation*. 251–264.

[66] Moinuddin Qureshi, Vijayalakshmi Srinivasan, and Jude Rivers. 2009. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In *International Symposium on Computer Architecture*. 24–33.

[67] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *ACM Symposium on Operating Systems Principles*. 392–407.

[68] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *USENIX Symposium on Operating Systems Design and Implementation*. 315–332.

[69] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security*. 753–768.

[70] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *ACM Symposium on Cloud Computing*. 323–337.

[71] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. 2011. *Library Cache Coherence*. Technical Report. Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory.

[72] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. 2020. IOctopus: Outsmarting Nonuniform DMA. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 101–115.

[73] Jeffrey Stuecheli, Bart Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (2015).

[74] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[75] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a Petabyte Scale Data Warehouse using Hadoop. In *International Conference on Data Engineering*. 996–1005.

[76] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE Symposium on Security and Privacy*. 339–354.

[77] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *USENIX Annual Technical Conference*. 773–787.

[78] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *International Conference on Management of Data*. 473–488.

[79] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon D. Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-performance Scientific Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–19.

[80] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX Annual Technical Conference*. 349–362.

[81] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX Conference on File and Storage Technologies*. 169–182.

[82] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *USENIX Symposium on Operating Systems Design and Implementation*. 191–208.

[83] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *USENIX Annual Technical Conference*. 601–617.

[84] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *USENIX Annual Technical Conference*. 199–215.

[85] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *ACM Symposium on Operating Systems Principles*. 147–162.

[86] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *USENIX Symposium on Operating Systems Design and Implementation*. 179–193.

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