ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory

Wonbae Kim† Chanyeol Park§,¶ Dongui Kim$^{\S,\pounds}$ Hyeongjun Park§
Young-ri Choi† Alan Sussman‡ Beomseok Nam§

UNIST† Naver¶ Line$^{\pounds}$ University of Maryland, College Park‡ Sungkyunkwan University§

Abstract

Due to the latency difference between DRAM and non-volatile main memory (NVMM) and the limited capacity of DRAM, incoming writes are often stalled in LSM tree-based key-value stores. This paper presents *ListDB*, a write-optimized key-value store for NVMM to overcome the gap between DRAM and NVMM write latencies and thereby, resolve the write stall problem. The contribution of ListDB consists of three novel techniques: (i) byte-addressable *Index-Unified Logging*, which incrementally converts write-ahead logs into SkipLists, (ii) *Braided SkipList*, a simple NUMA-aware SkipList that effectively reduces the NUMA effects of NVMM, and (iii) *Zipper Compaction*, which moves down the LSM-tree levels without copying key-value objects, but by merging SkipLists in place without blocking concurrent reads. Using the three techniques, ListDB makes background compaction fast enough to resolve the infamous write stall problem and shows 1.6x and 25x higher write throughputs than PACTree and Intel Pmem-RocksDB, respectively.

1 Introduction

Non-Volatile Main Memory (NVMM) is a new tier in the memory/storage hierarchy. NVMM has latency comparable to DRAM, but ensures non-volatility of data, similarly to secondary storage. Because NVMM is installed in the memory slot, it is byte-addressable and operates at memory bus speeds.

In order to locate and retrieve data from large datasets in NVMM, an efficient persistent index that takes into account the characteristics of NVMM is required. In the past few years, various NVMM-only persistent indexing structures [11,13,24,35,45,49,56,63] and hybrid DRAM+NVMM persistent indexing structures [38,40,49,63] have been proposed. In addition, several key-value stores that manage large datasets using such persistent indexes and background worker threads have been developed [12, 29, 30, 57, 60]. While NVMM-only indexing structures such as Fast and Fair B+tree [24], CCEH [45], and PACTree [32] provide orders of magnitude higher performance than their disk-based counterparts, their performance is still lower than a DRAM index because commercial NVMM products, e.g., Intel's Optane DC Persistent Memory Module, a.k.a., DCPMM [26], fall short of the performance of DRAM. Specifically, DCPMM has (i) latency higher than DRAM, (ii) bandwidth lower than DRAM, (iii) high sensitivity to NUMA effects, and (iv) a larger media access granularity (i.e., 256-byte *XPLine*) [62], which transforms a small write into a larger read-modify-write operation.

To benefit from DRAM performance and avoid the shortcomings of NVMM, the hybrid DRAM+NVMM indexing structures and key-value stores proposed in previous studies [12,49,57,60] place the complexity of indexing in volatile DRAM. In this work, we question whether such a hybrid approach that ignores the byte-addressability, keeps the entire index in DRAM, and uses NVMM only as log space is desirable, because it has two major limitations. First, the capacity of DRAM is small. If a dataset's index does not fit in small DRAM, or if DRAM is shared with the working sets of other processes, the existing hybrid DRAM+NVMM approaches may suffer from memory swapping of large indexes. Second, a volatile DRAM index needs to be reconstructed from scratch when recovering from a system failure. If a large number of key-value objects are stored without a persistent index that can survive system crashes, the recovery time can be significant. To improve the recovery performance, a volatile index can be periodically checkpointed [12]. However, such a periodic synchronous checkpointing results in very high tail latency because it blocks concurrent writes.

We advocate *asynchronous incremental checkpointing*, merging small, high-performance DRAM indexes into a persistent index in the background for data recovery. *ListDB* is a write-optimized LSM (log-structured merge) tree-based key-value store for NVMM. ListDB achieves high performance comparable to DRAM indexes, and prevents a DRAM index from growing indefinitely by flushing to NVMM at high throughput exceeding that of a DRAM index. ListDB buffers bulk insertions in a small DRAM index, and runs background *compaction* threads to incrementally checkpoint the buffered writes to NVMM without data copy. Instead, ListDB restructures log entries as a SkipList rather than flushing the entire volatile index to NVMM. Simultaneously, such SkipLists are merged in place, reducing NUMA effects, without blocking concurrent read queries.

Specifically, ListDB proposes the following three novel techniques - *Index-Unified Logging*, *Zipper Compaction*, and *Braided SkipList*. Our contributions are as follows.

- **Fast Write Buffer Flush:** ListDB unifies the write-ahead log with SkipList. Using *Index-Unified Logging* (IUL), ListDB writes each key-value object to NVMM only once, as a log entry. Taking advantage of NVMM's byte addressability, IUL converts a log entry into a SkipList element in a lazy manner, which masks the logging and MemTable flush overhead. Therefore, it makes the MemTable flush throughput higher than the write throughput of the DRAM index, thus resolving the write stall problem.
- **Reducing NUMA Effects:** *Braided SkipList* effectively reduces the number of remote NUMA node accesses by making the upper layer pointers point only to the SkipList elements on the same NUMA node.
- **Fast Compaction with In-Place Merge-Sort:** *Zipper compaction* merge-sorts two SkipLists in-place without blocking read operations. By avoiding copy, Zipper compaction alleviates the *write amplification* [21, 41, 53] problem and reduces the number of SkipLists fast and efficiently to improve read and recovery performance.

Our performance study shows that the write performance of ListDB outperforms state-of-the-art NVMM-based key-value stores. For read performance, ListDB relies on classic caching techniques.

The rest of the paper is organized as follows. In Section 2, we present the background and motivation. In Section 3, we present the design of ListDB. In Section 4, we compare the performance of ListDB against state-of-the-art key-value stores. Finally, we conclude the paper in Section 5.

2 Background and Motivation

2.1 Hybrid DRAM+NVMM Key-Value Store

Intel's Optane DCPMM is much faster than block device storage. However, its performance is still worse than that of DRAM in terms of latency, bandwidth, NUMA sensitivity, and access granularity [62]. Furthermore, byte-addressable persistency complicates failure-atomicity (i.e., reusability after a system crash) because the CPU cache replacement mechanism may evict dirty cachelines that are not ready to be persisted. When a system recovers, such prematurely written cachelines may corrupt data structures. To guarantee failure-atomicity despite such unexpected cacheline flushes, NVMM-only data structures carefully order machine instructions using memory fence instructions and call expensive `clflush` instructions frequently to persist dirty cachelines, which incurs significant overhead in NVMM [24, 39, 56]. To avoid this, several hybrid DRAM+NVMM indexing structures and key-value stores have been proposed. For example, NV-tree [63] and FP-tree [49] are variants of B+tree that store internal tree nodes in DRAM and leaf nodes in NVMM. The internal nodes are lost upon a system crash but can be reconstructed from persistent leaf nodes. With this approach, writes to internal nodes do not need to be failure-atomic.

FlatStore [12] takes a rather radical approach, i.e., NVMM is used only as a log space where key-value objects are appended in insert order rather than key order, whereas the index resides in DRAM. Therefore, FlatStore has to reconstruct a volatile index from persistent log entries after a system crash. To mitigate the expensive recovery overhead, FlatStore proposes to checkpoint the DRAM index onto NVMM periodically. However, a naive synchronous checkpointing, as in FlatStore, takes a global snapshot while blocking incoming writes, leading to unacceptably high tail latency.

2.2 Log-Structured Merge Tree

2.2.1 Asynchronous Incremental Checkpointing

A better approach is *asynchronous incremental checkpointing* [28], which checkpoints only the difference between the current checkpoint and the last checkpoint state. Log-Structured Merge (LSM) tree [47] is a classic index that consolidates checkpoint data over time [10, 17, 20, 33, 36, 48, 54].

2.2.2 Write in LSM Tree

An LSM tree buffers multiple write operations in an in-memory buffer space called *MemTable*, which sorts key-value objects using an ordered index such as SkipList [10, 17, 20, 33, 36, 48, 54]. Since a MemTable is volatile, a key-value object is written to a write-ahead log (WAL) for crash consistency before it is inserted into the MemTable. If the MemTable size exceeds a certain threshold, it is marked as immutable and a new MemTable is created so that the new MemTable can serve incoming clients' requests while a background thread transforms the immutable MemTable into a sorted array called SSTable (Sorted String Table), flushes it to disk, and then deletes the corresponding log entries. This design leverages the high performance of DRAM for random writes and the high sequential write bandwidth of block devices.

The key range of a MemTable is not disjoint with those of SSTables on disk. If a large number of MemTables are converted into SSTables and the overlap between SSTables increases, background threads merge-sort them to incrementally construct fewer, eventually into one large sorted array for fast search. This process, called *compaction*, is the most significant performance bottleneck because the same key-value object is repeatedly written to new SSTables [2, 9, 21, 29, 37, 41–43, 46, 53, 55].

2.2.3 Search in LSM Tree

For a read query, an LSM tree looks up a mutable MemTable, immutable MemTables, and then SSTables from level 0 to

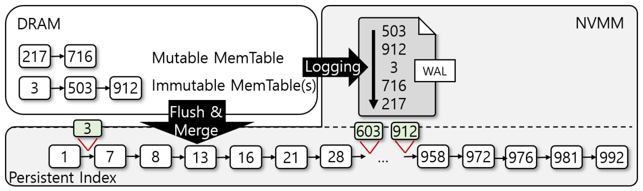


Figure 1: *Two-Level LSM Tree without Level 0 Buffer Indexes*

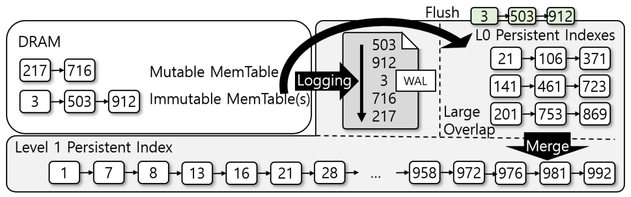


Figure 2: *Three-Level LSM Tree with Level 0 Buffer Indexes*

the upper levels, i.e., the recently stored objects are searched first. The search performance of LSM trees is affected by the degree of overlap between SSTables within and across levels because a read query searches all SSTables whose key range overlaps the search key until it finds a matched key. To reduce the overlap and improve the search performance, compaction threads merge-sort SSTables despite the high cost. Due to overlap and multiple levels, the read performance of LSM trees is worse than B+trees [27]. Nevertheless, LSM trees are more popular than B+trees in NoSQL systems because simple caching techniques can improve read performance. However, improving write performance is not easy.

2.2.4 Side Effect of Write Buffer: Write Stall

The in-memory MemTable is effective in buffering writes. However, despite buffering write bursts in the MemTable, tail latency can be very high if the workload is write-intensive because incoming writes can be blocked by *artificial governors* [31]. For instance, if compaction is slow, immutable MemTables will not be flushed to storage fast enough and the number of immutable MemTables will increase. Similarly, if SSTables are not merge-sorted quickly, the number of overlapping SSTables will increase, and search performance will degrade. Most LSM tree-based key-value stores [10, 17, 20, 33, 36, 48, 54] block clients from inserting new objects into the MemTable until compaction finishes and makes space for a new MemTable. This *write stall* problem occurs frequently in disk-based LSM tree-based key-value stores because of the high latency of the disk. If the write stall problem occurs, the insertion throughput is bounded by persistent storage performance, failing to benefit from the fast write buffer (DRAM) performance.

2.2.5 Write Amplification in LSM Trees

2.2.5.1 Multi-Level vs. Two-Level Compaction

As SSTables accumulate in storage, LSM trees perform compaction to merge-sort SSTables and reduce the overlap. Compaction is particularly expensive in disk-based key-value stores because they copy key-value objects between SSTable files. That is, compaction threads select a set of overlapping SSTables at level k and another set of SSTables that overlap at the next level $k+1$, and merge-sort them to create a new set of SSTables at level $k+1$. Such copy-based compaction allows concurrent read queries to access old SSTables while new SSTables are being created. However, copy-based compaction requires the same objects to be repeatedly copied to new SSTables. The number of times a key-value object is copied to a new file, called *write amplification* factor, has been reported to be as high as 40 [41, 53, 55]. The write amplification is particularly serious if key-value stores use *leveled compaction* and a large number of levels [53, 55]. The leveled compaction limits the number of SSTables per level and prevents any overlap between the SSTables at the same level.

NVMM allows byte-addressable updates. Therefore, there is an opportunity to avoid write amplification and improve compaction performance by replacing multiple levels of SSTables with a high-performance single-level persistent index. In particular, SLM-DB [29] uses two levels, i.e., MemTables and a single persistent B+tree in NVMM. Using the two-level design (shown in Figure 1), MemTables buffer multiple key-value objects and later insert them into a large persistent index in ascending order of keys, such that the large persistent index is traversed only once for multiple writes and it yields a higher write throughput than a single persistent index.

2.2.5.2 Decoupling Merge-Sort from Flush

The main problem with the two-level design is that the size of the persistent index affects the performance of merging volatile indexes into a persistent index, i.e., it fails to make write performance independent of NVMM performance. This is because MemTables are not flushed[1] *as-is*, but merge-sorted into the large, slow persistent index. Because NVMM has higher latency than DRAM, merge-sort throughput is much lower than insert throughput of volatile indexes, especially when the persistent index is large.

To alleviate this problem, most key-value stores including LevelDB [36] and RocksDB [54] employ an intermediate persistent buffer level (level 0, $L0$) in storage. That is, they flush MemTables to the intermediate buffer level without doing merge-sort. Figure 2 shows such a three-level design. By separating merge-sort from flush, MemTables can be flushed to NVMM faster; the flush throughput becomes independent of the database size.

A drawback of this design is that it results in a large number of overlapping SSTables, which hurts search performance. Given its poor indexing performance, the intermediate persistent buffer level does not appear to be very different from write-ahead log. Furthermore, key-value objects are written

[1] To avoid confusion with the cacheline flush instruction (e.g., `clflush`), writing a MemTable to NVMM is henceforth referred to as *flush*, and the cacheline flush is referred to as *persist*.

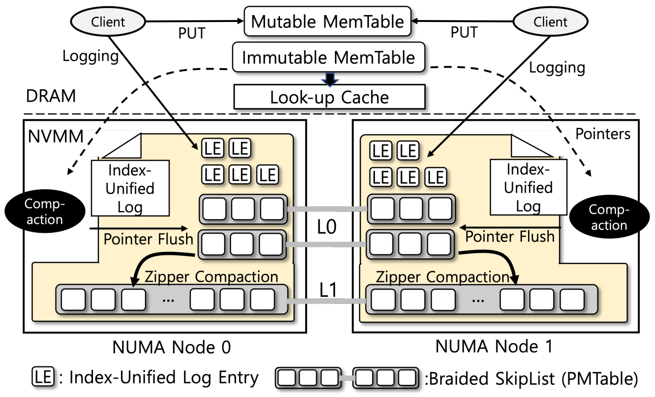


Figure 3: ListDB Architecture

to storage at least twice, i.e., once for WAL and once again for MemTable flush.

TRIAD [3], WiscKey [41], and FlatStore [12] prevent the same key-values (or just values) from being repeatedly written. TRIAD is particularly inspiring because it considers the commit log as an unsorted $L0$ SSTable. To enable efficient search in the unsorted $L0$ SSTables (the commit log), TRIAD creates a small index file for each $L0$ SSTable. The index file does not store keys and values, only the offsets for each object in sorted order of the keys. Although TRIAD reduces the I/O traffic, each MemTable flush creates an index file and calls the expensive `fsync()` to make it durable. However, given the high overlap between $L0$ SSTables and also the fact that $L0$ SSTables will be quickly merged into $L1$ SSTables, it is questionable whether a separate index file for each $L0$ SSTable should be created and persisted at a very high cost.

2.3 NUMA Effects

NVMM is more sensitive to NUMA effects than DRAM because of its lower bandwidth (1/6 for writes and 1/3 for reads) [16,57,62]. As such, state-of-the-art persistent indexes, such as FAST and FAIR B+tree [24] and CCEH [45] do not scale with the number of threads due to irregular cacheline accesses and NUMA effects [32,57].

To mitigate the NUMA effects, Daase et al. [16] suggest limiting the number of write threads to 4-6 per socket. Nap [57] hides NUMA effects by overlaying a DRAM index on top of NVMM-resident indexes such that the DRAM index can absorb remote NUMA node accesses. However, data stored in NVMM is already in the memory address space, and NVMM has latency comparable to DRAM. Therefore, using DRAM as a fast cache layer over NVMM and copying data between DRAM and NVMM back and forth can be wasteful. For example, NVMM file systems such as EXT4-DAX and NOVA [61] do not use the page cache but directly access NVMM.

To mitigate NUMA effects in DRAM, various approaches, including *Delegation* with hash-based sharding [4,6,7,44] and *Node Replication* (NR) [7] methods, have been investigated. In Delegation methods, a designated worker thread is assigned for all operations on a specific range of keys. Therefore, client threads have to communicate with worker threads and delegate operations using message passing. Due to the significant message passing overhead, Delegation performs sub-optimal, especially for lightweight tasks such as indexing operations [7]. Node Replication (NR) [7] implements a NUMA-aware shared log, which is used to replay the same operations for the data structures replicated across NUMA nodes. However, this consumes memory for replicating the same data structure across multiple NUMA nodes. Besides, the performance falters due to cross-node communication, as the number of NUMA nodes increases [7]. Considering that the bandwidth of Optane DCPMM is much lower than that of DRAM [62], replication can aggravate the low bandwidth problem.

3 Design of ListDB

ListDB is a write-optimized key-value store with an LSM tree structure that resolves the write stall problem. In this section, detailed descriptions of ListDB's key designs are provided. First, the overall architecture of ListDB is presented (§3.1). Then, its key designs, i.e., *Index-Unified Logging* (§3.2), NUMA-aware *Braided SkipList* (§3.3), in-place *Zipper Compaction* (§3.4), *lookup cache* (§3.5), and recovery algorithm (§3.6) are presented.

3.1 Three-Level Architecture

Figure 3 shows the three-level architecture of ListDB-volatile MemTables, and $L0$ and $L1$ Persistent MemTables (*PMTables*). MemTables and PMTables are essentially the same SkipLists, but the node structure of PMTable has additional metadata that MemTable does not need because PMTable is a data structure transformed from the write-ahead log. ListDB uses SkipList as the core data structure for all levels because it enables byte-addressable in-place merge-sort and avoids the *write amplification* problem [21,41,53], as will be presented throughout the paper.

ListDB employs an intermediate persistent buffer level - $L0$ (level 0) in NVMM. With level 0, a MemTable is flushed to NVMM without being merge-sorted, making the flush throughput independent of the next level persistent index size. MemTables accumulated at $L0$ ($L0$ PMTables) are gradually merged into the large $L1$ PMTable by compaction. To manage multiple PMTables, ListDB uses a metadata object called MANIFEST to point to the beginning of each SkipList.

3.2 Index-Unified Logging

ListDB aims to flush MemTables to NVMM without copying key-value objects. As discussed in Section 2.2.5.2, all key-value objects in MemTables are already persisted in the commit log in NVMM [3]. Besides, $L0$ indexes are known to have very poor indexing performance due to large overlap.

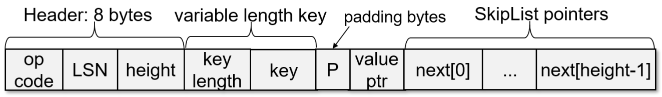


Figure 4: Index-Unified Log Entry Layout

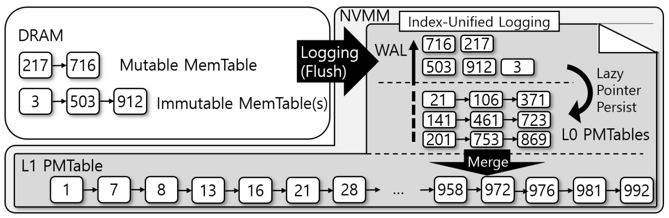


Figure 5: Index-Unified Logging

3.2.1 Conversion of IUL into SkipList

Index-Unified Logging (IUL) unifies write-ahead log entries and SkipList elements by allocating and writing log entries in the form of SkipList elements. Figure 4 shows the structure of an IUL entry, which serves both as a log entry and as a SkipList element. When a key-value object is inserted into a MemTable, the object and its metadata (i.e., operation code `op_code` and log sequence number `LSN`) are written and persisted as a log entry in NVMM with SkipList pointers initialized to NULL (Algorithm 1). Later, when a compaction thread flushes its corresponding MemTable from DRAM, the log entry is converted into a SkipList ($L0$ PMTable) element, reusing the key and value stored in the log entry.

The information that $L0$ PMTable needs, but the log does not have, is the sorted order of keys, which is managed as SkipList pointers in MemTables. When converting the log into an $L0$ PMTable, the addresses of the corresponding MemTable elements are simply translated into NVMM addresses, i.e., the log entry offsets, as shown in Algorithm 2. When the SkipList pointers in IUL entries are set to NVMM addresses, the IUL entries become SkipList elements.

Finally, the MANIFEST is updated to validate the new $L0$ PMTable and invalidate the immutable MemTable in a failure-atomic transaction.

3.2.2 MemTable Flush without `clflush`

When writing SkipList pointers to log entries, there is no need to call persist instructions (e.g.,`clflush`) because the key-value objects are already persistent in the log, and because the order of keys can be recovered without difficulty in case of a crash. Instead of explicitly persisting cachelines for updated pointers, Index-Unified Logging leaves that to the CPU cache replacement mechanism, i.e., it waits until the CPU evicts updated pointers from its cache. Through the CPU cache replacement mechanism, multiple pointer updates to the same 256-byte XPLine can be buffered and batched. That is, each 8-byte small write is not eagerly transformed into a 256-byte read-modify-write operation. Not only does it defer the read-modify-write problem, but also prevents background

Algorithm 1 *Put(kvObject)*

```
 1: mutex.lock();
 2: iul_entry ← iul_tail;
 3: iul_entry.LSN ← GetNextLSN(); /* log sequence number */
 4: iul_entry.height ← RandomHeight(); /* SkipList element height */
 5: iul_tail ← iul_tail + sizeof(kvObject) + height∗8 + 8;
 6: mutex.unlock();
 7: iul_entry.op_code ← OP_INSERT; /* operation type (insert, delete) */
 8: iul_entry.kvObject ← kvObject;
 9: iul_entry.next[0..height] ← NULL; /* initialize pointers */
10: pmem_persist(iul_entry, sizeof(iul_entry)); /* calls clwb */
11: memTable.Insert((SkipListElement)iul_entry); // classic SkipList insert
```

Algorithm 2 *FlushImmutableMemTable(memTable)*

```
 1: element ← memTable.head[0].next[0]; // smallest MemTable element
 2: while element≠NULL do
 3:     L0_element ← element.iul_address;
 4:     lookup_cache.Insert(L0_element);
 5:     for layer ← 0; layer < element.height; layer++ do
 6:         L0_element.next[layer] ← element.next[layer].iul_address;
 7:         /* no need to persist */
 8:     end for
 9: end while
10: new_L0.iul_address ← memTable.head[0].next[0].iul_address;
11: new_L0.next ← MANIFEST.L0List().GetFront();
12: MANIFEST.L0List().PushFront(new_L0); /* CAS */
13: freeMemTable(memTable);
```

compaction threads from being affected by the read-modify-write problem and high NVMM write latency.

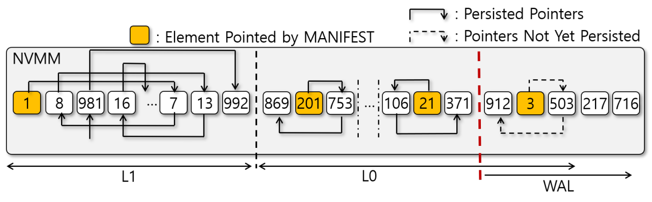
3.2.3 Walk-Through Example

Let us walk through MemTable flush illustrated in Figure 5. Suppose foreground client threads insert keys into the currently mutable MemTable in the order of 503, 912, and 3. Each client thread persists the object, its metadata, and NULL pointers in the log before it commits. Then, a background thread marks the MemTable as immutable and creates a new MemTable. Client threads insert two more keys, 716 and 217, into the new mutable MemTable.

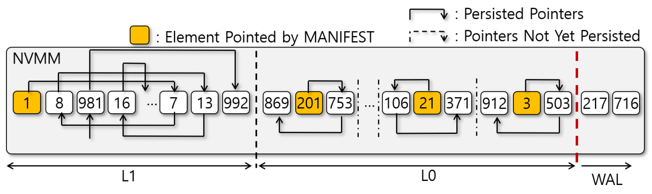
When a background compaction thread flushes the immutable MemTable, i.e., (3, 503, 912), the pointers of each MemTable element are simply translated into the IUL offsets of the corresponding log entries and the NULL pointers are replaced with the IUL offsets so that the log entries become a SkipList, as shown in Figure 6(a). As described in Section 3.2.2, the updated pointers in the new $L0$ PMTable may remain in the CPU cache and may be lost upon a system crash, but the pointers are not required for crash consistency.

3.2.4 Checkpointing $L0$ PMTable

Although the log entries are now converted to $L0$ PMTable elements, the boundary between logging space and $L0$ PMTable space (denoted as a thick dotted line in Figure 6(a)) has not moved, because it is not guaranteed that the pointers of the new $L0$ PMTable are persistent. The boundary can only move if `clflush` instructions are explicitly called for the updated pointers. In our implementation, a background thread persists dirty cachelines for $L0$ PMTables in batches. This op-



(a) NVMM Layout Before Checkpointing



(b) NVMM Layout After Checkpointing

Figure 6: NVMM Layout of Index-Unified Logging

eration is referred to as *checkpointing*. Figure 6(b) shows the NVMM layout after the pointers are explicitly persisted. Once a PMTable is checkpointed, it is possible to move the boundary of the logging space to reduce the number of log entries to recover, as shown in Figure 6(b).

3.2.4.1 Lazy Group Checkpointing

Checkpointing reduces recovery time. However, ListDB defers checkpointing as much as possible, because calling `clflush` instructions is very expensive. Even if $L0$ PMTables are not persisted at all, it does not affect crash consistency because all the elements in all $L0$ PMTables will be treated as log entries if the system crashes, and the key order of $L0$ PMTable elements can be reconstructed from the log.

In our implementation, multiple $L0$ PMTables are grouped and dirty cachelines for them are persisted in batches. We call this *lazy group checkpointing*. Note that there is a trade-off between lazy group checkpointing and recovery time. Infrequent checkpointing increases the log size and it takes longer to recover. In contrast, if checkpointing frequency is high, recovery will be fast, but flush throughput degrades.

Zipper compaction, which will be described in Section 3.4, persists pointers fast enough to prevent the number of $L0$ PMTables from increasing. That is, even if IUL does not persist any $L0$ PMTable, Zipper compaction persists pointers fast when merging an $L0$ PMTable into the $L1$ PMTable, and the recovery time of IUL is much shorter than synchronous checkpointing, as will be shown in Section 4.

3.3 NUMA Effects for SkipList

ListDB employs a NUMA-aware data structure, which is more scalable and effective in minimizing NUMA interconnect contention than Delegation and Node Replication [7].

3.3.1 NUMA-aware Braided SkipList

A SkipList has the invariant that the list at each layer[2] is a sorted sub-list of the bottom layer [52]. Unless this invariant

[2]To avoid confusion with the *level* of LSM trees, the level of SkipList will be referred to as *layer*.

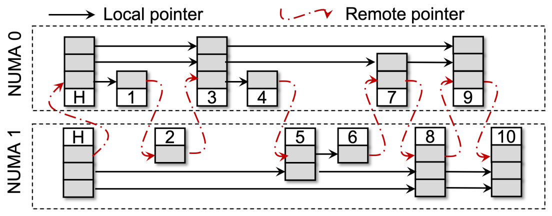


Figure 7: NUMA-aware Braided SkipList

is violated, correct search results are guaranteed because the upper layer pointers are probabilistic shortcuts, which do not affect the correctness of search results. However, an upper layer does not need to be a sub-list of the next layer, as long as it is a sub-list of the bottom layer. Even if a search does not find a key closer to the search key in an upper layer, the search falls back to a lower layer and eventually to the bottom layer which contains all sorted keys.

The Braided SkipList of ListDB leverages this property to mitigate NUMA effects in a simple and effective way. Upper layer pointers ignore SkipList elements in remote NUMA nodes; i.e., upper layer pointers of each element point to an element with a larger key in the same NUMA node. Compared to NUMA-oblivious conventional SkipLists, Braided SkipList reduces the number of remote memory accesses to $1/N$, where N is the number of NUMA nodes, as will be shown in Section 4.

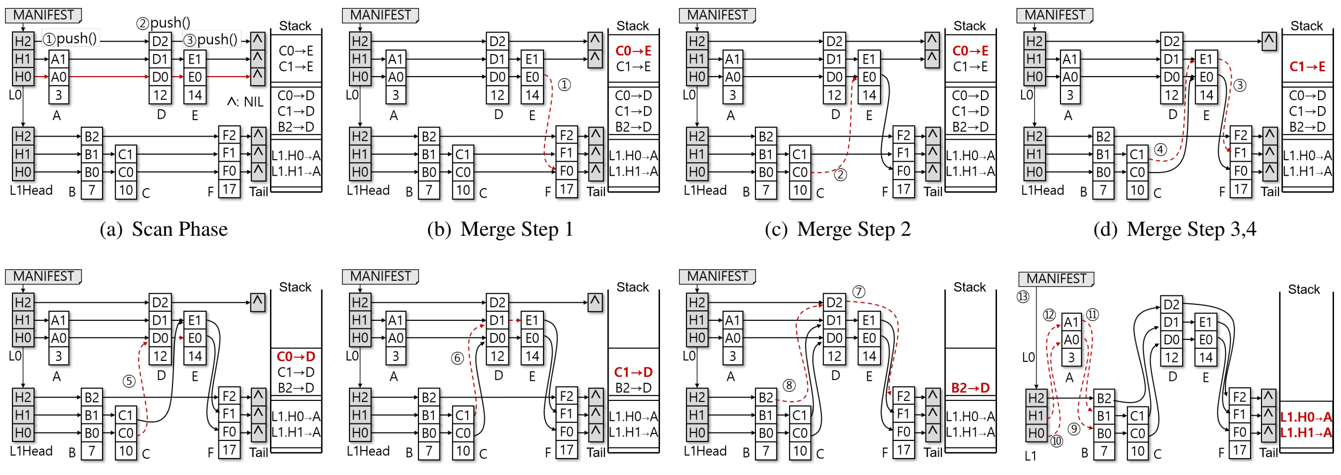
Figure 7 illustrates an example (The upper layers in NUMA node 1 are illustrated upside down for ease of presentation). Observe that the second layer pointer of element `3` on NUMA node 0 points to element `7` on the same NUMA node, instead of element `5` on NUMA node 1. Nonetheless, a correct search is guaranteed. For example, suppose a client thread running on NUMA node 0 searches for element `5`. It will follow the top layer to element `3`, then `9`. Since `9` is greater, the thread moves down one layer in element `3`, and then the search visits element `7`. Since `7` is greater than `5`, the thread moves down again and follows the bottom layer pointer to element `4`. Since the search key is greater than `4`, it follows the bottom layer to a remote SkipList element `5`. The search then completes.

In our implementation of Braided SkipList, a NUMA ID is embedded in the extra 16 bits of the 64-bit virtual address, as in *pointer swizzling* [59], such that it can use 8-byte atomic instructions instead of expensive PMDK transactions [50]. For direct reference, Braided SkipList restores the virtual memory address of a SkipList element by masking the extra 16 bits.

3.4 Zipper Compaction

With byte-addressable NVMM, Zipper compaction merge-sorts $L0$ and $L1$ PMTables in-place by only updating pointers, but without blocking concurrent read queries. The in-place merge-sort avoids write amplification, thus it improves the compaction throughput.

Leveraging the SkipList invariant (§3.3.1), various lock-free SkipLists have been studied in the literature [22, 23], and the Java™ SE `ConcurrentSkipListMap` class has been



(a) Scan Phase (b) Merge Step 1 (c) Merge Step 2 (d) Merge Step 3,4

(e) Merge Step 5 (f) Merge Step 6 (g) Merge Step 7,8 (h) Compaction Done

Figure 8: *Zipper Compaction: Merging SkipLists from Tail to Head*

shown to perform well in practice [34]. Zipper compaction algorithm allows concurrent read operations to access $L0$ and $L1$ PMTables while merging them without violating the invariant of SkipList.

A classic lock-free SkipList avoids locks for multiple writers. In contrast, ListDB does not perform concurrent writes; the only writers are the compaction threads, and ListDB coordinates them to avoid write-write conflicts. For parallelism, multiple compaction threads write to disjoint *shards*. A shard is a disjoint key range from an element with the maximum height to the next element with the maximum height in $L1$ PMTable. To merge $L0$ elements into $L1$, a compaction thread must acquire a lock on the corresponding shard.

Zipper compaction proceeds in two phases; (i) a forward *scan* from head to tail and (ii) a backward *merge* from tail to head, hence the name. To guarantee correct search results without blocking concurrent readers, $L0$ PMTable elements are merged into $L1$ PMTable from tail to head while concurrent read operations are traversing them from head to tail.

3.4.1 Scan Phase

In the forward scan phase, a compaction thread traverses $L0$ and $L1$ PMTables from head to tail and determines where each $L0$ PMTable element should be inserted in the $L1$ PMTable. However, in this phase, it does not make any change to the PMTables but pushes necessary pointer updates on a stack. The backward merge phase pops the stack to apply and persist the updates to the $L1$ PMTable.

The scan phase follows the bottom layer of $L0$ PMTable. For each $L0$ element, it searches the $L1$ PMTable to find where to insert the $L0$ element. For this, it keeps track of the rightmost element smaller than the current search key ($L0$ element) in each layer to avoid repeatedly traversing $L1$ PMTable. Since keys are sorted in both PMTables, the next larger key in $L0$ PMTable can reuse the previous rightmost elements, and backtrack to the top-layer rightmost element for the next search. Therefore, the complexity of the scan phase is $O(n_0 + \log n_1)$ where n_0 and n_1 are the sizes of $L0$ and $L1$ PMTables, respectively.

Algorithm 3 shows the pseudo-code of Zipper compaction. For NUMA-aware Braided SkipLists, Zipper compaction requires a two-dimensional array - `rightmost[numa_id][layer]` to keep as many rightmost elements in each layer as the number of NUMA nodes for Braided SkipList. But, note that a Braided SkipList element does not need more pointers than a conventional SkipList element as it embeds NUMA node ID in the 8-byte address.

Figure 8(a) shows an example of Zipper scan. For ease of presentation, all SkipList elements are assumed to be on the same NUMA node. The first element `A` in $L0$ will be placed in the first position in $L1$. Hence, `H0` and `H1` of the head element in $L1$ are the current rightmost pointers that need to be updated for `A`. This information is stored on the stack. Note that `A0` and `A1` need to point to `B`, but they are not pushed onto the stack because `B` is pointed by the current rightmost elements that are already pushed on the stack. Each $L0$ element is inserted between two $L1$ elements and only the previous (i.e., rightmost) element in each layer needs to be pushed on the stack because the next element can be found from the previous elements. Next, the scan phase visits the second element `D` in $L0$ and searches $L1$. Inserting `D` requires updating `B2`, `C1`, and `C0`. Again, they are pushed onto the stack. Finally, it visits the last element `E` in $L0$ and searches $L1$. Note that $L1$ PMTable has not changed and the current rightmost pointers are still `B2`, `C1`, and `C0`. Thus, the scan phase pushes `C1` and `C0` on the stack to make them point to `E`.

3.4.2 Merge Phase

The merge phase applies pointer updates from tail to head. When a compaction thread pops a pointer update $X_N \rightarrow Y$ from the stack, the Nth layer pointer in element Y is updated to the current value of X_N. Then, X_N is set to the address of Y.

Algorithm 3 *BraidedZipperCompaction(L0SkipList, L1SkipList)*

```
 1: LogZipperCompactionBegin(L0SkipList); // micro-logging
 2: L0_element ← L0SkipList.head[0].next[0]; // smallest L0 element
 3: local_numa_id ← DecodeNumaId(L0_element); // not always 0
 4: for i ← 0; i < NumNUMA; i++ do
 5:     rightmost[i][] ← L1SkipList.head[i].next[]; // array copy
 6: end for
 7: bottom_L1_element ← L1SkipList.head[0].next[0];
 8: L1_element ← L1SkipList.head[local_numa_id]; // local head
 9: // I. scan phase: from head to tail
10: while L0_element≠NULL do
11:     // NUMA-aware local search for upper layer pointers
12:     for layer ← L0_element.height−1; layer > 0; layer−− do
13:         while L1_element.next[layer] ≠ NULL &&
                    L1_element.next[layer].key < L0_element.key do
14:             L1_element ← L1_element.next[layer];
15:             // update the rightmost for the current layer
16:             rightmost[local_numa_id][layer] ← L1_element;
17:         end while
18:     end for
19:     // NUMA-oblivious search for bottom-layer pointer, i.e., layer = 0
20:     L1_element ← bottom_L1_element;
21:     <<same while loop with line 13–17 >>
22:     // push an array of NUMA local upper-layer pointers and a NUMA-
        oblivious bottom layer pointer
23:     stack.push(L0_element, rightmost[local_numa_id][]);
24:     // fetch the next L0_element and update local NUMA ID
25:     L0_element ← L0_element.next[0];
26:     local_numa_id ← DecodeNumaId(L0_element);
27:     bottom_L1_element ← L1_element;
28:     L1_element ← rightmost[local_numa_id][L0_element.height−1];
29: end while
30: // II. merge phase: from tail to head
31: while stack is not empty do
32:     (L0_element, rightmost2update[]) ← stack.pop();
33:     for layer ← 0; layer < L0_element.height; layer++ do
34:         // Pop and apply the updates without worries about NUMA IDs
35:         L0_element.next[layer] ← rightmost2update[layer].next[layer];
36:         if layer = 0 then
37:             persist(L0_element.next[layer]);
38:         end if
39:         rightmost2update[layer].next[layer] ← L0_element;
40:         if layer = 0 then
41:             persist(rightmost2update[layer].next[layer]);
42:         end if
43:     end for
44:     second_chance_cache.Insert(L0_element);
45: end while
46: MANIFEST.L0List().PopBack(); /* CAS */
47: LogZipperCompactionDone(L0SkipList); // micro-logging
```

In the example, shown in Figure 8(b), the compaction thread pops `C0`→`E` and sets `E0` to `F`, which is the current value of `C0`. At this point, the upper layer pointer of element `E` (`E1`) is not pointing to element `F`. However, as described earlier, upper layer pointers are probabilistic shortcuts, which do not affect the correctness of search. Therefore, there is no need to update `E0` and `E1` atomically. In the next step, shown in Figure 8(c), the compaction thread sets `C0` to the address of `E`. In the next step, shown in Figure 8(d), the compaction thread pops `C1`→`E`, sets `E1` to `F`, and makes `C1` point to `E`. Each pointer update is removed from the stack one by one, and is applied in order, as shown in Figures 8(e), 8(f), 8(g), and 8(h). Zipper compaction assumes 8-byte pointer updates are atomic. To make the updates failure-atomic, it persists each bottom layer update immediately using memory fence and cacheline flush instructions. In the final step, the compaction thread deletes the head element of $L0$ PMTable from the MANIFEST object, thus completing compaction.

3.4.3 Lock-Free Search

Zipper compaction does not violate the correctness of concurrent search, i.e., a read thread will not miss its target SkipList element without acquiring a lock. This is because a read thread accesses PMTables from head to tail and from $L0$ to $L1$, whereas a compaction thread merges them from tail to head. During Zipper compaction, every element is guaranteed to be pointed by at least one head. Consider the example shown in Figure 8, which shows how a sequence of atomic store instructions merges the two example SkipLists. Even if a concurrent read thread accesses the PMTables in any state shown in Figure 8, it returns a correct result.

The algorithm remains correct even if a read thread is suspended during compaction thread is making changes to SkipLists. For example, suppose a read is suspended while accessing an $L0$ element. When it resumes, the element might have been merged into $L1$. When the read thread wakes up, it will continue traversing to the tail if it does not find the search key. Once it reaches the tail, it is done with $L0$ and will start searching $L1$, into which $L0$ elements have been merged. Consequently, the read thread might visit the same elements multiple times, but it will never miss the element it is searching. Multiple visits might hurt search performance. To avoid this, a read stops searching the $L0$ if it detects the level of the current element is $L1$.

3.4.4 Updates and Deletes

An update in LSM trees duplicates the same key because writes are buffered in MemTables and gradually flushed to the last level. ListDB does not eagerly delete the older version in $L1$. Instead, when a compaction thread scans $L0$ and $L1$ levels for Zipper compaction, it marks the older version in $L1$ obsolete. Similarly, a delete in ListDB does not physically delete an object but inserts a *key-delete* object into the MemTable. If an LSM tree physically deletes the most recent version of a key from MemTables or $L0$ PMTables, older versions of the key will come back to life. Zipper compaction places a more recent key-value or key-delete object before its corresponding old objects. Therefore, a read query always accesses the more recent object before older ones, and thus returns a correct search result.

3.4.5 Fragmentation and Garbage Collection

Using `libpmemobj` library [50], ListDB allocates and deallocates a *memory chunk* (e.g., 8 MB) for multiple IUL entries in PMDK's failure-atomic transaction so that the number of calls to expensive PMDK transactions can be reduced. ListDB deallocates a memory chunk if all elements in the chunk are marked obsolete or deleted. Note that ListDB does not relocate SkipList elements for garbage collection. To address the lasting fragmentation, a compaction thread may perform CoW-based garbage collection. We leave this optimization for our future work.

Memory management for lock-free data structures is a hard problem because there is no easy way to detect whether deallocated memory space is still being accessed by concurrent reads [5, 14, 19]. ListDB employs a simple epoch-based reclamation [14]; ListDB does not deallocate memory chunk immediately but waits long enough for short-lived read queries to finish accessing the deallocated memory chunk. A background garbage collection thread periodically checks and reclaims a memory chunk if all objects in the memory chunk are obsolete or deleted. For obsolete objects, the garbage collection thread checks its newer version's LSN. If it is also old enough, it considers the obsolete objects are not accessed by any reads, removes them from $L1$ PMTable, and physically deallocates the memory chunk.

3.4.6 Linearizability

Theorem 1. *Zipper compaction is linearizable with a single writer and multiple readers.*

Proof. For some element e, there is a single *linearization point* [23] for a writer when its level changes from $L0$ to $L1$, by atomic update of the bottom-layer next pointer. We denote this linearization point as $e.merge\_to\_L1$.

There are two linearization points $e.search\_L0$ and $e.search\_L1$ for a reader, as it searches $L0$ and then $L1$ PMTable in order. Let $a \rightarrow b$ if an event a happens before another event b. There are three cases to consider.

1. $e.merge\_to\_L1 \rightarrow e.search\_L0 \rightarrow e.search\_L1$
2. $e.search\_L0 \rightarrow e.merge\_to\_L1 \rightarrow e.search\_L1$
3. $e.search\_L0 \rightarrow e.search\_L1 \rightarrow e.merge\_to\_L1$

In case 1, $e.search\_L1$ will find e in $L1$. In case 2, $e.search\_L0$ will find e in $L0$. If the search does not stop after finding e in $L0$, $e.search\_L1$ will also find e in $L1$. In case 3, similarly, $e.search\_L0$ will find e. Since all three cases succeed in finding e, Zipper compaction is linearizable, meaning a read always succeeds in finding an element if the element was inserted by a committed write transaction, regardless of whether the element is in $L0$ or $L1$ PMTable. □

3.5 Look-up Cache

ListDB requires that a read query accesses at least two indexes, i.e., a mutable MemTable and $L1$ PMTable. Therefore, the read throughput of ListDB is significantly lower than a highly-optimized persistent B+tree, as we show in Section 4.

To mitigate this problem, ListDB uses a *lookup cache* in DRAM. Flushing a MemTable hashes each element into a fixed-sized static hash table. Unlike disk-based designs, the lookup cache does not duplicate the element in it, but only stores its NVMM address because the element in NVMM is already in the memory address space and its address never changes. Hence, regardless of the level at which the PMTable element is present, the lookup cache can locate the PMTable element. SkipList pointers are frequently updated by compaction threads in ListDB. By caching immutable addresses, not mutable content, the lookup cache can avoid frequent cache invalidation. If a hash collision occurs on a bucket, the old address is overwritten (i.e., FIFO replacement policy).

Algorithm 4 *Get(key)*

```
1: iter ← MANIFEST.GetTableIterator();
2: table ← iter.GetTable(); // get mutable MemTable
3: while table ≠ NULL && table.IsPMTable() = false do
4:     value ← table.Search(key); // SkipList lookup
5:     if value ≠ NULL then
6:         return value; // Found: return value
7:     end if
8:     table ← (++iter).GetTable(); // immutable MemTables
9: end while
10: /* L0 Cache Lookup */
11: cached ← lookup_cache.Lookup(key);
12: if cached ≠ NULL && cached.GetElement().key = key then
13:     return cached.GetElement().value;
14: end if
15: /* L0 Search */
16: while table ≠ NULL && table.Level() = 0 do
17:     value ← table.Search(key); // SkipList lookup
18:     if value ≠ NULL then
19:         return value; // Found: return value
20:     end if
21:     table ← (++iter).GetTable(); // L0 PMTables
22: end while
23: /* L1 Search */
24: rightmost ← second_chance_cache.Lookup(key);
25: value ← table.SearchFromElement(key, rightmost);
26: if value ≠ NULL then
27:     return value; // Found: return value
28: end if
29: return NOT_FOUND;
```

ListDB constructs a SkipList in DRAM as a second chance lookup cache for tall elements evicted from the hash table. The purpose of the second chance lookup cache is to accelerate PMTable search. Even if a key is not found in the second chance cache, a query can start the search from the closest PMTable element found in the cache. Algorithm 4 shows how a read query uses the lookup caches. Suppose a read searches for key 100 but finds element 85 is the closest smaller element in $L1$. Then, the search continues from element 85 in $L1$ PMTable instead of the beginning of $L1$. ListDB does not use the second chance lookup cache for $L0$ search because small $L0$ PMTables are merged into $L1$ fast, and $L0$ elements are mostly cached in the lookup hash table, The second chance lookup cache uses the SIZE replacement policy [58], i.e., it compares heights and evicts elements with shorter heights.

3.6 Recovery

A system may crash while $L0$ and $L1$ PMTables are being merged by Zipper compaction. To recover from such failures, a compaction thread performs micro-logging to keep track of which $L0$ PMTable is being merged into $L1$ PMTable. When a system restarts, ListDB checks the compaction log to redo unfinished compactions. For redo operations, Zipper compaction has to check duplicate entries since many entries in the tail of

Algorithm 5 *RecoverDB()*

```
 1: ScheduleUnfinishedZipperCompactionJob();
 2: curr_table ← NULL;
 3: while log_iter.Valid() do
 4:     iul_entry ← log_iter.GetIULEntry();
 5:     table_id ← GetTableIdByLSN(iul_entry.LSN);
 6:     if curr_table = NULL || table_id ≠ curr_table.Id() then
 7:         curr_table ← MANIFEST.GetTableById(table_id);
 8:         curr_table.ResetSkipListHead();
 9:     end if
10:     curr_table.InsertEntry(iul_entry); /* SkipList Insert */
11:     log_iter.Next(); /* from old to latest */
12: end while
```

$L0$ PMTable can be shared with $L1$ PMTable.

The recovery algorithm of ListDB, shown in Algorithm 5, is similar to that of conventional LSM trees. First, a recovery process locates the boundary of WAL, which is recorded by compaction threads in the compaction log. Then, it sorts log entries and restores $L0$ PMTables. At this point, the system returns to the normal execution mode and starts processing clients' queries. Compaction between $L0$ and $L1$ will be done in the background as normal.

As for the lookup cache, ListDB can process clients' queries without restoring the cache although the search performance will be poor until the cache is populated. By avoiding the reconstruction of DRAM cache and index, the recovery performance of ListDB is superior to synchronous checkpointing [12], as we show in Section 4.

4 Evaluation

4.1 Experimental Setup

Experiments are conducted on a four-socket NUMA server with Intel Xeon Gold 5215 CPU (2.50 GHz, 20 vCPUs) per socket, 256 GB of DDR4 DRAM (16x 16 GB), and 2 TB (16x 128 GB) Optane DCPMM (4 DCPMM's and 4 DRAM's per each socket) in app-direct mode. Our testbed server supports only the *directory coherence* protocol, but not *snoop* protocol, despite its known NUMA bandwidth meltdown issues [32].

All implementations are compiled using gcc 7.5.0 with `-O3` optimization. Using PMDK [50], ListDB creates an auto-growing directory-based persistent memory poolset (`pmempool`) on each NUMA node [50]. For NUMA-oblivious designs, we use the device mapper to create a single persistent memory poolset interleaved on four sockets.

We evaluate the performance of ListDB[3] using two individual sets of experiments. First, the performance effects of each part of the design of ListDB are quantified. Second, the performance of ListDB is compared against that of state-of-the-art persistent indexes, including FAST and FAIR B+tree [24] and PACTree [32], and LSM tree-based key-value stores for NVMM, i.e., NoveLSM [30], SLM-DB [29], and Intel's industry-optimized Pmem-RocksDB [51]. Pmem-RocksDB [51] is a variant of RocksDB for NVMM that Intel has optimized in two respects. First, Pmem-RocksDB separates keys and values to mitigate write amplification issues, as in WiscKey [41]. Second, Pmem-RocksDB mmaps SSTables and writes directly to NVMM by using non-temporal stores (i.e., `ntstore`) to bypass the cache hierarchy and eliminate context switching.

[3]Source code is available at http://github.com/DICL/listdb.

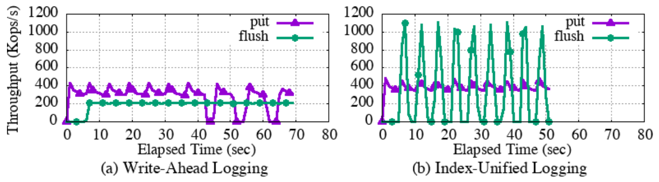


Figure 9: *Low Flush Throughput Results in Write Stalls*

Our experiments use YCSB [15] and the Facebook benchmark [8]. The Facebook benchmark generates more realistic workloads than YCSB as it emulates real-world RocksDB workloads in Facebook/Meta datacenters. Specifically, the Facebook benchmark adds mathematical models (e.g., sine distribution) to db_bench [18] such that it can vary key sizes, value sizes, and query arrival rates over time.

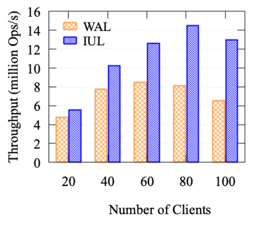
4.2 Evaluation of Index-Unified Logging

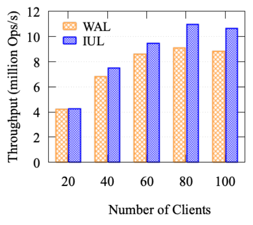
4.2.1 IUL vs. WAL: Flush Throughput

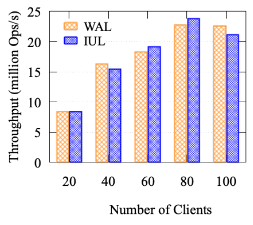
This section compares the performance effect of IUL to standard WAL with respect to write stalls. For the experiments shown in Figure 9, a single YCSB [15] client thread (Load A) and a single compaction thread are used to evaluate how fast a MemTable absorbs bursts of 20 million writes (8-byte key and 8-byte value objects), and how fast a single background compaction thread flushes MemTables to NVMM. To prevent memory usage from increasing indefinitely, the maximum number of immutable MemTables is set to 4. Zipper compaction threads are disabled to evaluate only the effect of IUL, i.e., $L1$ is not used. `put` denotes the client's query processing throughput over time (i.e., the number of records inserted into MemTables per second), and `flush` denotes how many records are flushed from MemTables to NVMM by the compaction thread.

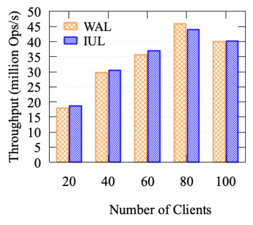
Figure 9 (a) shows that with standard WAL, `put` throughput is higher than `flush` throughput because inserting key-value objects into a SkipList in DRAM is much faster than flushing (i.e., copying key-value objects from DRAM to NVMM) and persisting a SkipList in NVMM. Each spike in `put` throughput indicates that a new empty mutable MemTable was created; it takes about 5 seconds to fill a 64 MB MemTable. In 40 seconds, the number of MemTables exceeds the threshold, and subsequent writes are blocked. Even if the threshold is set to a higher value than four, it is only a matter of time before a write is stalled, because `flush` throughput is lower than `put` throughput.

In contrast, Figure 9 (b) shows that with IUL, `flush` throughput is much higher than `put` throughput. `flush` throughput of IUL fluctuates because each flush takes less time than filling a MemTable, i.e., the compaction thread be-


(a) Load A


(b) Workload A


(c) Workload B


(d) Workload C

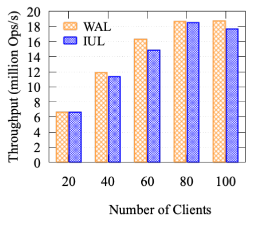

(e) Workload D

Figure 10: Performance Effect of Index-Unified Logging

comes idle. This high `flush` throughput is because IUL does not copy key-value objects from DRAM to NVMM and does not call cacheline flush instructions. So, write stalls do not occur and the compaction thread often becomes idle, allowing the CPU to perform other work.

4.2.2 Evaluation of IUL using YCSB

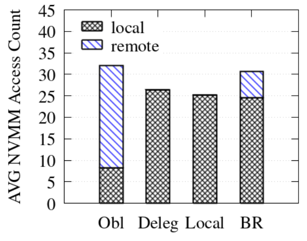
The experiments shown in Figure 10 compare the performance of IUL to standard WAL with varying the number of client threads for YCSB workloads. The number of background compaction threads is set to half the number of client threads. The MemTable size and the maximum memory usage for MemTables are set to 256 MB and 1 GB, respectively, i.e., a maximum of 4 MemTables are allowed. We set the lookup cache size to 1 GB (979 MB hash-based lookup cache and 45 MB for the second chance lookup cache). For the experiments, Braided SkipList and Zipper compaction are enabled so that $L0$ PMTables are merged into $L1$ PMTable and read queries can run faster. The `Load A` workload populates the database with 100 million records (8-byte keys and 8-byte values). All other workloads submit 100 million queries each.

Figure 10(a) shows that increasing the number of client threads increases the write throughput of both logging methods, up to 80 threads. With 80 client threads, the throughput of IUL (14.513 million ops/sec) is approximately 1.8x higher than that of WAL (8.101 million ops/sec). However, when the number of client threads exceeds the number of logical cores throughput degrades due to the high overcommit rate. That is, 100 client threads and 50 background compaction threads compete for 80 logical cores. Still, the throughput of IUL is 99% higher than WAL.

For Workload B (95% reads), Workload C (100% reads), and workload D (read latest), WAL has similar or slightly better performance than IUL because WAL does copy-on-writes to store records in ascending order of keys, and read operations benefit from higher memory access locality than IUL. Nevertheless, IUL outperforms WAL in Workload A (50:50 Read:Write) due to its better write performance.

4.3 Evaluation of Braided SkipList

This section evaluates NUMA effects in NVMM using a single PMTable. The performance of the NUMA-aware Braided SkipList (denoted as `BR`) is compared with three other methods that were discussed in Section 2.3; i.e., (i) NUMA-oblivious SkipList (denoted as `Obl`), (ii) delegating client


(a) Memory Access Count

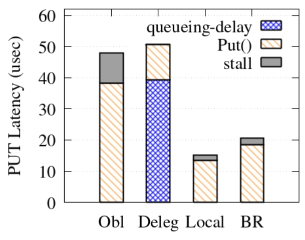
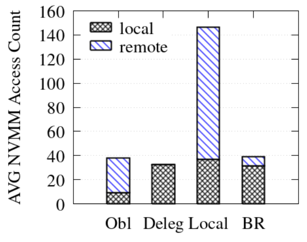

(b) Response Time Breakdown

Figure 11: PUT Performance (80 Clients)


(a) Memory Access Count

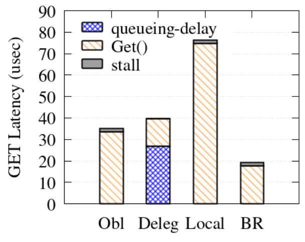
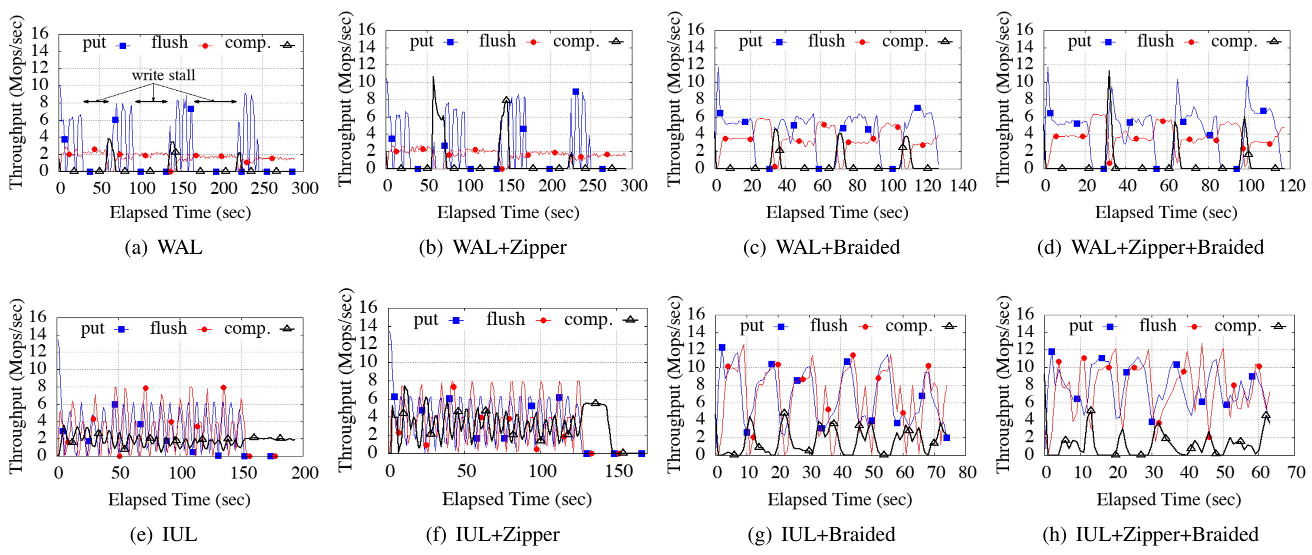

(b) Response Time Breakdown

Figure 12: GET Performance (80 Clients)

queries to a worker thread, using shared memory (denoted as `Deleg`), and (iii) a write-optimal local SkipList (denoted as `Local`), which manages a SkipList per NUMA node. `BR` and `Obl` manage one large PMTable, whereas `Deleg` and `Local` create four smaller PMTables. `Deleg` partitions key-value records according to hash keys, but `Local` allows a write client to insert data into the SkipList on its local NUMA node regardless of the key. Consequently, a read query has to search all four SkipLists. Even if a key is found in the local index, it must search remote indexes because a remote index may have a more recent update. Therefore, when there are n NUMA nodes, the ratio of local accesses is always $1/n$.

Our experiments, shown in Figures 11 and 12, run YCSB Load A (100 million inserts, 5-25 bytes string keys, 100-byte values) and Workload C (10 million queries).

PUT: Figure 11(b) shows that `Local` has the lowest write response time because it always inserts into the local PMTable. This eliminates remote NUMA node access for writes as shown in Figure 11(a). Braided SkipList (denoted as `BR`) has a higher write response time than `Local` because `BR` accesses remote NVMM via bottom layer pointers. Figure 11(a) shows that most NVMM accesses using `BR` are local, unlike NUMA-



(a) WAL (b) WAL+Zipper (c) WAL+Braided (d) WAL+Zipper+Braided (e) IUL (f) IUL+Zipper (g) IUL+Braided (h) IUL+Zipper+Braided

Figure 13: Put/Flush/Compaction Throughput over Time (YCSB Load A)

oblivious SkipList (20.1% vs. 74.1% remote accesses). `Obl` and `BR` access NVMM more than `Deleg` and `Local`.

Similar to `Local`, `Deleg` also completely removes remote NVMM access, but the write response time is significantly higher due to delegation overhead. That is, threads use slow atomic instructions to access the shared queue and make a memory copy for queries and results. Figure 11(b) shows that the queueing delay accounts for 77.1% of query response time with 80 client threads. Because put/get operations on a lock-free index are very lightweight, the synchronization overhead incurred by delegation dominates the overall response time.

GET: Figure 12(a) shows that the response time of `BR` for read queries is lower than the other methods. While `Local` outperforms `BR` for writes, the read response time of `Local` is about 4x higher than `BR` because `Local` must search all 4 PMTables. Although `BR` avoids visiting a more efficient search path that follows remote elements, Figures 11(a) and 12(a) show that it has almost no effect on the traversal length. `Deleg` shows the fewest memory accesses. However, due to synchronization overhead, its query response time is about 2x higher than `BR`, so its performance is even lower than `Obl`.

4.4 Putting It All Together

Figure 13 presents a factor analysis for ListDB. [4] We enable and disable each design feature of ListDB and measure write throughput (denoted `put`), flush throughput (MemTable $\rightarrow L0$ PMTable, denoted `flush`), and compaction throughput ($L0 \rightarrow L1$ PMTable, denoted `comp.`) over time. We run 80 client threads and 40 background compaction threads for YCSB Load A, inserting 500 million 8-byte keys and 8-byte values. Figure 13(a) shows that disabling all three optimizations causes client threads to stall for more than 50 seconds. Enabling Zipper compaction improves the $L0 \rightarrow L1$ compaction throughput as shown in Figure 13(b), but the write stall problem still occurs because of the memory copy overhead for flushing the MemTable. If Braided SkipList is used, accessing remote NUMA nodes can be avoided when flushing the MemTable. Therefore, flush throughput doubles, which results in less frequent write stalls, as shown in Figure 13(c). Enabling both Zipper compaction and Braided SkipList results in shorter write stall times, and the workload completes in less than 120 seconds (Figure 13(d))

If IUL is used instead of WAL, `flush` throughput becomes comparable to `put` throughput, as shown in Figure 13(e). By avoiding expensive memory copy, write stalls are less frequent than WAL. However, note that compaction throughput is much lower than flush throughput. This increases the number of $L0$ PMTables and degrades search performance. As shown in Figure 13(f), if additionally IUL and Zipper compaction are enabled, the NVMM bandwidth improves by reducing the number of memory copies. Thus, it improves compaction and flush throughput. Enabling IUL and Braided SkipList, as shown in Figure 13(g), avoids NUMA effects, which improves both compaction and flush throughput. Finally, with all three optimizations enabled, the workload completes in under 65 seconds with virtually no write stalls (Figure 13(h)) compared to 300 seconds in figure 13(a).

4.5 Recovery Performance

We evaluate the recovery performance of asynchronous incremental checkpointing for ListDB and periodic synchronous checkpointing. Using the Facebook benchmark, we populate a database with 100 million objects and measure the time to recover using a checkpoint and write-ahead log entries. Despite using the same workload, the recovery performance of synchronous checkpointing is affected by the checkpointing interval, whereas asynchronous checkpointing is only affected by the query arrival rate. This is because the number of the

[4] Note that the scale of the x axis differs between the subfigures.

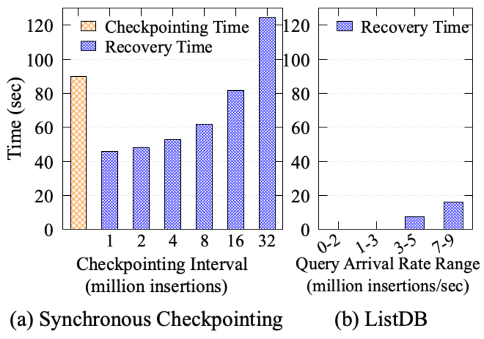


Figure 14: Recovery Performance

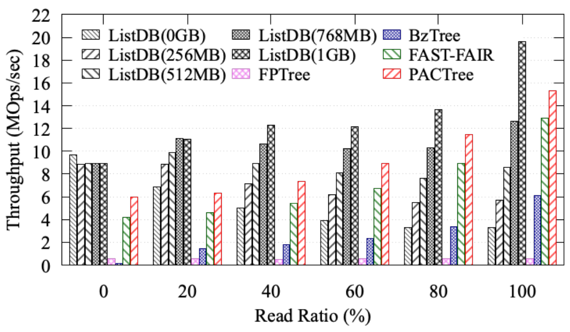


Figure 15: Comparison with Other Designs

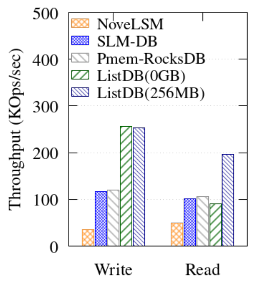


Figure 16: Comparison with NoveLSM and SLM-DB

log entries varies with asynchronous checkpointing, which background compaction threads have not yet merged into $L1$. If the query arrival rate is higher than the Zipper compaction throughput, the number of the IUL entries increases and the recovery process has to create a larger $L0$ PMTable with more log entries.

Figure 14 shows that a synchronous checkpointing takes about 90 seconds to serialize and flush the in-memory B+tree using the `binary_oarchive` class from the Boost library. This causes concurrent queries to block for 90 seconds while the checkpointing is being performed, resulting in unacceptably high tail latency. To alleviate the problem, checkpointing can be performed less frequently, but that increases the recovery time (i.e., the time to restore the checkpointed index and insert log entries to it) as more log entries accumulate.

In contrast, Figure 14 (b) shows that ListDB recovers instantly if it crashes when the write query arrival rate is lower than 3 million insertions/sec. If the query arrival rate varies between 7 and 9 million insertions/sec, ListDB takes about 19 seconds to recover. With a higher query arrival rate, the recovery time of ListDB increases.

4.6 Comparison with Other Designs

The experiments shown in Figure 15 compare the performance of ListDB with state-of-the-art persistent indexes; i.e., BzTree [1], FP-tree [49], FAST and FAIR B+tree [24], and PACTree [32], We run the experiments on a two-socket machine, because PACTree is hardcoded for two sockets. The two-socket machine has the same Intel Xeon Gold 5215 CPUs (40 logical cores in total), 128 GB DRAM (8x 16GB), and 1 TB (8x 128 GB) DCPMM. The database is pre-loaded with 100 million key-value records and then 40 clients submit 10 million queries with uniform distribution (generated from YCSB Workload A) with various read-write ratios. These tree-structured indexes are not optimized for (or do not support) large variable-length string keys and values. Therefore, we generated 8-byte numeric keys and 8-byte pointer values for the workload, which is favorable for tree-structured indexes with large fanouts.

Figure 15 shows that ListDB outperforms tree-structured persistent indexes for write-intensive workloads. For the write-only workload, `ListDB(0GB)` shows 79x, 17x, 2.3x and 1.6x higher throughput than BzTree, FPTree, FAST and FAIR B+tree, and PACTree, respectively. However, for the read-only workload, tree-structured indexes benefit from faster search performance. In particular, FAST and FAIR B+tree and PACTree show 3.88x and 4.61x higher search throughputs, respectively, than `ListDB(0GB)`. With the lookup cache enabled, ListDB outperforms or shows comparable performance to tree-structured indexes. The numbers in parentheses in the graph key show the lookup cache size. With a lookup cache larger than 768 MB, ListDB outperforms PACTree unless the read ratio is higher than 80%.

These results confirm that standard caching techniques can easily improve read performance. However, the lookup cache that indexes the location of key-value records cannot be used for PACTree, FAST FAIR B+tree, FPTree, etc. because they frequently relocate key-value records to different tree nodes due to tree rebalancing operations. That is, employing a DRAM cache for a tree-structured persistent index is not as simple as our address-only lookup caching. For example, Nap [57] has a very complicated caching mechanism.

4.6.1 Write Amplification

Although LSM trees have better write performance than tree-structured indexes, they have higher write amplification, as a critical limitation in block device storage [21, 41, 53]. To compare write amplification, we used Intel PMwatch [25] to measure the total number of accessed bytes in the experiments shown in Figure 15. All indexing methods suffer from high write amplification. DCPMM's internal write combining buffer transforms a small write (8-byte key and 8-byte value) into a 256-byte read-modify-write operation, resulting in at least 16x write amplification. In ListDB, the writes are further amplified by merge-sort operations in $L0$ and $L1$ PMTables. However, the write amplification of ListDB (104.4) is lower than that of FAST and FAIR B+tree (126.789) and comparable to that of PACTree (91.5) because ListDB merge-sorts SkipLists in-place.

4.7 Comparison with NoveLSM and SLM-DB

Figure 16 shows the single-threaded read and write throughput of NoveLSM, SLM-DB, Pmem-RocksDB, and ListDB.

The experiments run a single client thread (`db_bench`, 100 million random 8-byte keys and 1 KB values) because NoveLSM crashes when multiple threads concurrently access the database. NoveLSM and SLM-DB were designed to use NVMM as an intermediate layer on top of the block device file system, but our experiments store all SSTables in NVMM formatted with EXT4-DAX for a fair comparison.

NoveLSM shows the worst performance, not because of its design but because it is implemented on top of LevelDB, which is known to have poor performance. SLM-DB is also implemented on top of LevelDB but shows better performance because it uses FAST and FAIR B+tree as its core index. Since SLM-DB is not yet ported to use PMDK, it has no overhead imposed by run-time flushing or transactional updates, i.e., it shows DRAM performance and does not survive a system crash. Nonetheless, SLM-DB does not show better performance than Pmem-RocksDB, a fully persistent key-value store. Compared to Pmem-RocksDB, `ListDB(0GB)` shows twice the write throughput, but read performance is slightly worse unless the lookup cache is enabled. This is because Pmem-RocksDB benefits from memory locality by storing keys contiguously in NVMM in sorted order, whereas ListDB does not relocate data.

4.8 Comparison with Pmem-RocksDB

Finally, we compare the performance of ListDB with Intel's Pmem-RocksDB using the *Prefix Dist* workload in the Facebook benchmark. The experiments shown in Figure 17 run 80 client threads and use the default key and value sizes of the benchmark (48-byte string keys and variable-length values ranging from 16 bytes to 10 KB). The workload submits queries according to a query arrival rate (QPS parameter) that follows a sine distribution with a noise factor of 0.5. The put/get ratio of the workload is 3 to 7.

For various parameter settings, ListDB consistently outperforms Pmem-RocksDB. The results of two different settings are presented in Figure 17 - an `idle` workload (0.1 $\sim$ 0.3 million write queries and 0.2 $\sim$ 0.7 million read queries arrive per second; 200 million queries in total), in which the throughput of Pmem-RocksDB is saturated, and a `heavy` workload (2.4 $\sim$ 7.2 million write queries and 5.6 $\sim$ 16.8 million queries arrive per second; 5 billion queries in total), in which the throughput of ListDB is saturated. The lookup cache is disabled for ListDB while setting the maximum DRAM usage for both key-value stores to 1 GB and allowing Pmem-RocksDB to use the default 8 MB block cache.

For the `idle` workload, Pmem-RocksDB suffers from excessive NVMM writes, so its `put` throughput saturates at 200 Kops. For the Facebook benchmark, a `get` query has to wait for its previous `put` query to commit. Therefore, the `get` throughput of Pmem-RocksDB saturates at 400 Kops in the experiment. In contrast, Figure 17(b) shows that the throughput of ListDB follows the sine distribution, i.e., the query arrival rate, without blocking queries.

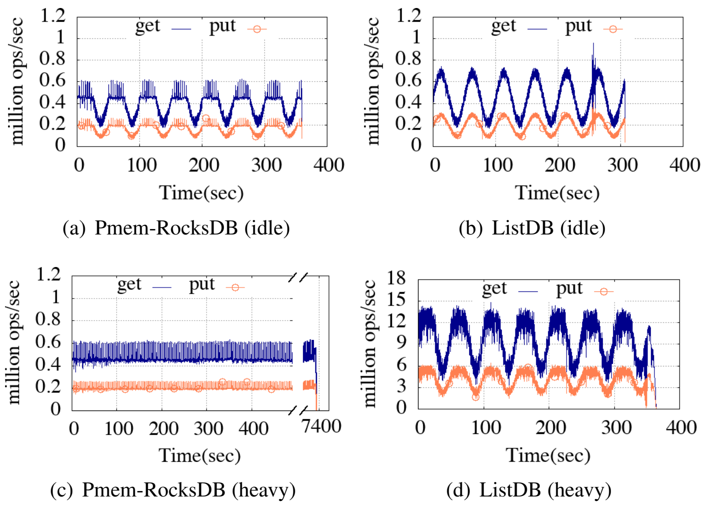


Figure 17: Throughput over Time (Facebook Benchmark)

For the `heavy` workload, Pmem-RocksDB's throughput is still saturated. On the other hand, the `put` throughput of ListDB is 25x higher than that of Pmem-RocksDB, i.e., 5 million ops. Similarly, the `get` throughput of ListDB is up to 22x higher than that of Pmem-RocksDB (i.e., 13 million vs. 0.6 million ops). As such, ListDB completes the workload 19.4x faster than Pmem-RocksDB (i.e., 380 vs. 7400 seconds).

5 Conclusion

In this work, we design and implement ListDB - a key-value store that leverages the byte-addressability to avoid data copies by restructuring data in-place and high-performance of NVMM to reduce the write amplification and avoid write stalls. We show that ListDB significantly improves write performance via asynchronous incremental checkpointing and in-place compaction. With its three-level structure, ListDB outperforms state-of-the-art persistent indexes and NVMM-based key-value stores in terms of write throughput. A standard lookup cache can help mitigate the problem of having multiple levels. For future work, we are exploring the possibility of improving search performance by introducing another level, namely L2 PMTable, to opportunistically rearrange L1 PMTable elements for spatial locality and garbage collection.

Acknowledgement

We would like to give special thanks to our shepherd Dr. Marc Shapiro and to the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by Samsung Electronics, and also by the R&D program of National Research Foundation of Korea (NRF) (grant No. NRF2018R1A2B3006681) and IITP (grant No. 2018-0-00549 and 2021-0-01817) and Electronics and Telecommunications Research Institute (ETRI) grant (grant No. 20ZS1310) funded by the Korean government. The corresponding author is Beomseok Nam.

References

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, jan 2018.

[2] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache storage for the next billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 123–136, 2009.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 363–375, 2017.

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[5] Trevor Brown. Reclaiming Memory for Lock-Free Data Structures: There has to be Better Way. In *Proceedings of the 34th ACM Symposium on the Principles of Distributed Computing (PODC'15)*, 2015.

[6] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*, June 2013.

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 207–221, 2017.

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

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 1007–1019, 2018.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[11] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[12] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 1077–1091, 2020.

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

[14] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures (SPAA'15)*, 2015.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

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

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[18] Facebook. db_bench. `https://github.com/facebook/rocksdb/wiki/Benchmarking-tools`.

[19] Thomas E. Harta, Paul E. McKenneyb, Angela Demke Brown, and Jonathan Walpole. Performance of Memory

Reclamation for Lockless Synchronization. *Journal of Parallel and Distributed Computing*, 67:1270–1285, 2007.

[20] HBase. https://hbase.apache.org/.

[21] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 127–144, 2017.

[22] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, 06 2007.

[23] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

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

[25] Intel. PMWatch. https://github.com/intel/intel-pmwatch.

[26] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[27] Varun Jain, James Lennon, and Harshita Gupta. LSM-Trees and B-Trees: The Best of Both Worlds. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, page 1829–1831, 2019.

[28] Ashok Joshi, William Bridge, Juan Loaiza, and Tirthankar Lahiri. Checkpointing in Oracle. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, page 665–668, 1998.

[29] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, pages 191–205, 2019.

[30] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 993–1005, 2018.

[31] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. BoLT: Barrier-Optimized LSM-Tree. In *Proceedings of the 21st International Middleware Conference (Middleware)*, page 119–133, 2020.

[32] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, page 424–439, 2021.

[33] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[34] D. Lea. Java Platform SE 8, java.util.concurrent.ConcurrentSkipListMap. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

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

[36] LevelDB. https://github.com/google/leveldb.

[37] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.

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

[39] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, 2018.

[40] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, apr 2020.

[41] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[42] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 207–219, 2015.

[43] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, pages 142–156, 2017.

[44] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 319–320, 2012.

[45] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*, pages 31–44, 2019.

[46] Suman Nath and Aman Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 410–419, 2007.

[47] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[48] Oracle Berkeley DB. https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

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

[50] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[51] PMEM-RocksDB. https://github.com/pmem/pmem-rocksdb.

[52] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[53] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[54] RocksDB. https://rocksdb.org/.

[55] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, 14(11):2216–2229, jul 2021.

[56] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[57] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–111, July 2021.

[58] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 1996.

[59] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *SIGARCH Computeer Architecture News*, 19(4):6–13, jul 1991.

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

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

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

[63] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: Reducing Consistency Const for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.