Redesigning LSMs for Nonvolatile Memory with NoveLSM

Sudarsun Kannan, ***University of Wisconsin-Madison;***
Nitish Bhat and Ada Gavrilovska, ***Georgia Tech;***
Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, ***University of Wisconsin-Madison***

https://www.usenix.org/conference/atc18/presentation/kannan

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Redesigning LSMs for Nonvolatile Memory with NoveLSM

Sudarsun Kannan
University of Wisconsin-Madison

Nitish Bhat
Georgia Tech

Ada Gavrilovska
Georgia Tech

Andrea Arpaci-Dusseau
University of Wisconsin-Madison

Remzi Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

We present NoveLSM, a persistent LSM-based key-value storage system designed to exploit non-volatile memories and deliver low latency and high throughput to applications. We utilize three key techniques – a byte-addressable skip list, direct mutability of persistent state, and opportunistic read parallelism – to deliver high performance across a range of workload scenarios. Our analysis with popular benchmarks and real-world workload reveal up to a 3.8x and 2x reduction in write and read access latency compared to LevelDB. Storing all the data in a persistent skip list and avoiding block I/O provides more than 5x and 1.9x higher write throughput over LevelDB and RocksDB. Recovery time improves substantially with NoveLSM's persistent skip list.

1 Introduction

Persistent key-value stores based on log-structured merged trees (LSM), such as BigTable [13], LevelDB [4], HBase [2], Cassandra [1], and RocksDB [3], play a crucial role in modern systems for applications ranging from web-indexing, e-commerce, social networks, down to mobile applications. LSMs achieve high throughput by providing in-memory data accesses, buffering and batching writes to disk, and enforcing sequential disk access. These techniques improve LSM's I/O throughput but are accompanied with additional storage and software-level overheads related to logging and compaction costs. While logging updates to disk before writing them to memory is necessary to recover from application or power failure, compaction is required to restrict LSM's DRAM buffer size and importantly commit non-persistent in-memory buffer to storage. Both logging and compaction add software overheads in the critical path and contribute to LSM's read and write latency. Recent proposals have mostly focused on redesigning LSMs for SSD to improve throughput [23, 30, 40].

Adding byte-addressable, persistent, and fast non-volatile memory (NVM) technologies such as PCM in the storage stack creates opportunities to improve latency, throughput, and reduce failure-recovery cost. NVMs are expected to have near-DRAM read latency, 50x-100x faster writes, and 5x higher bandwidth compared to SSDs. These device technology improvements shift performance bottlenecks from the hardware to the software stack, making it critical to reduce and eliminate software overheads from the critical path of device accesses. When contrasting NVMs to current storage technologies, such as flash memory and hard-disks, NVMs exhibit the following properties which are not leveraged in current LSM designs: (1) random access to persistent storage can deliver high performance; (2) in-place update is low cost; and (3) the combination of low-latency and high bandwidth leads to new opportunities for improving application-level parallelism.

Given the characteristics of these new technologies, one might consider designing a new data structure from scratch to optimally exploit the device characteristics. However, we believe it worthwhile to explore how to redesign LSMs to work well with NVM for the following reasons. First, NVMs are expected to co-exist with large-capacity SSDs for the next few years [27] similar to the co-existence of SSDs and hard disks. Hence, it is important to redesign LSMs for heterogeneous storage in ways that can exploit the benefits of NVMs without losing SSD and hard disk optimizations. Second, redesigning LSMs provides backward compatibility to thousands of applications. Third, maintaining the benefits of batched, sequential writes is important even for NVMs, given the 5x-10x higher-than-DRAM write latency. Hence in this paper, we redesign existing LSM implementations.

Our redesign of LSM technology for NVM focuses on the following three critical problems. First, existing LSMs maintain different in-memory and persistent storage form of the data. As a result, moving data across storage and memory incurs significant serialization and deserialization cost, limiting the benefits of low latency NVM. Second, LSMs and other modern applications [1–4, 13] only allow changes to in-memory data structures and make the data in persistent storage immutable. However, memory buffers are limited in their capacity and must be frequently compacted, which increases stall time. Buffering data in memory can result in loss of data after a system failure, and hence updates

must be logged; this increases latency, and leads to I/O read and write amplification. Finally, adding NVM to the LSM hierarchy increases the number of levels in the storage hierarchy which can increase read-access latency.

To address these limitations, we design **NoveLSM**, a persistent LSM-based key-value store that exploits the byte-addressability of NVMs to reduce read and write latency and consequently achieve higher throughput. NoveLSM achieves these performance gains through three key innovations. First, NoveLSM introduces a *persistent NVM-based memtable*, significantly reducing the serialization and deserialization costs which plague standard LSM designs. Second, NoveLSM makes the *persistent NVM memtables mutable*, thus allowing direct updates; this significantly reduces application stalls due to compaction. Further, direct updates to NVM memtable are committed in-place, avoiding the need to log updates; as a result, recovery after a failure only involves mapping back the persistent NVM memtable, making it three orders of magnitude faster than LevelDB. Third, NoveLSM introduces *optimistic parallel read*s to simultaneously access multiple levels of the LSM that can exist in NVM or SSD, thus reducing the latency of read requests and improving the throughput of the system.

We build NoveLSM by redesigning LevelDB, a widely-used LSM-based key-value store [4]. NoveLSM's design principles can be easily extended to other LSM implementations [1–3]. Our analysis reveals that NoveLSM significantly outperforms traditional LSMs when running on an emulated NVM device. Evaluation of NoveLSM with the popular DBbench [3,4] shows up to 3.8x improvement in write and up to 2x improvement in read latency compared to a vanilla LevelDB running on an NVM. Against state-of-the-art RocksDB, NoveLSM reduces write latency by up to 36%. When storing all the data in a persistent skip list and avoiding block I/O to SSTable, NoveLSM provides more than 5x and 1.9x gains over LevelDB and RocksDB. For the real-world YCSB workload, NoveLSM shows a maximum of 54% throughput gain for scan workload and an average of 15.6% across all workloads over RocksDB. Finally, the recovery time after a failure reduces significantly.

2 Background

We next provide background on LSM trees and on the design of popular LSM stores, LevelDB [4] and RocksDB [3], used extensively in this work. We also present a background on persistent memory and our method of emulating it.

2.1 Log Structured Merge Trees

An LSM-tree proposed by O'Neil et al. [32] is a persistent structure that provides efficient indexing for a key-value store. LSMs achieve higher write throughput by first staging data in memory and then across multiple levels on disk to avoid random disk writes. In LSMs, the levels have an increasing size; for example, in LevelDB, each level is at least ten times larger than the previous level. During an insert operation, the keys are first inserted into an in-memory level, and as this level fills up, the data slowly trickles down to disk-friendly block structures of the lower levels, where data is always sorted. Before every insert operation into the memory level, the data (key-value pairs) is logged in the persistent storage for recovery after a failure; the logs are garbage collected after data is safely flushed and persisted to on-disk structures. Next, the search and read operations proceed from the top memory level to the disk levels and their latency increases with increasing number of levels. In general, LSMs are update-friendly data structures and read operations are comparatively slower to other NoSQL designs.

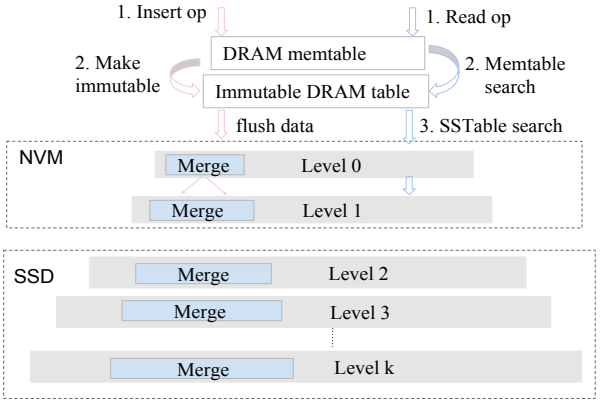

Figure 1: **Naive LevelDB design with NVM.** *Figure shows a simple method to add NVM to the LSM hierarchy. NVM is used only as a replacement to disk for storing SSTables. Shaded blocks show immutable storage, grey and red arrows show steps for read operation search and the background compaction.*

2.2 Popular LSM Stores

LevelDB is a popular LSM-based key-value store derived from Google's BigTable implementation and is widely-used from browsers to datacenter applications. Figure 1 shows LevelDB's design with NVM added to the storage hierarchy. During an insert operation, LevelDB buffers updates in a memory-based skip list table (referred to as memtable hereafter) and stores data on multiple levels of on-disk block structures know as sorted string tables (SSTable). After the memtable is full, it is made immutable and a background compaction thread moves the immutable memtable data to on-disk SSTable by serializing the data to disk-based blocks. Only two levels of memory tables (mutable and immutable) exist. With an exception to memtables, all lower levels are mutually exclusive and do not maintain redundant data.

The SSTables are traditional files with a sequence of I/O blocks that provide a persistent and ordered immutable mapping from keys to values, as well as interfaces for sequential, random, and range lookup operations. The SSTable file also has a block index for locating a block in $O(1)$ time. A key lookup can be performed with a single disk seek to read the block and binary search inside the block. In LevelDB, for read operations, the files with SSTables are memory-mapped to reduce the POSIX file system block-access overheads.

RocksDB is an LSM implementation that extends LevelDB to exploit SSD's high bandwidth and multicore parallelism. RocksDB achieves this by supporting multi-threaded background compaction which can simultaneously compact data across multiple levels of LSM hierarchy and extracts parallelism with multi-channel SSDs. RocksDB is highly configurable, with additional features such as compaction filters, transaction logs, and incremental replication. The most important feature of RocksDB that can be beneficial for NVM is the use of a Cuckoo hashing-based SST table format optimized for random lookup instead of the traditional I/O block-based format with high random-access overheads.

In this work, we develop NoveLSM by extending LevelDB. We choose LevelDB due to its simplicity as well as its broad usage in commercial deployments. The optimizations in RocksDB over LevelDB are complementary to the proposed NoveLSM design principles.

2.3 Byte-addressable NVMs

NVM technologies such as PCM are byte-addressable persistent devices expected to provide 100x lower read and write latency and up to 5x-10x higher bandwidth compared to SSDs [7, 12, 17, 22]. Further, NVMs can scale to 2x-4x higher density than DRAM [5]. These attributes make NVM a suitable candidate for replacing SSDs. Additionally, NVMs are expected to be placed in parallel with DRAM connected via the memory bus, thereby providing memory-like load/store access interface that can avoid POSIX-based block access supported in current storage devices. Further, the read (load) latency of NVMs is comparable to DRAM, but the write latency is expected be to 5x slower.

2.4 NVM Emulation

Since byte-addressable NVMs are not available commercially, we emulate NVMs similarly to prior research [12, 17, 22, 26] and our emulation methodology uses a 2.8 GHz, 32-core Intel Nehalem platform with 25 MB LLC, dual NUMA socket with each socket containing 16 GB DDR3 memory, and an Intel-510 Series SSD. We use Linux 4.13 kernel running DAX-enabled Ext4 [6] file system designed for persistent memory. We use one of the NUMA sockets as NVM node, and to emulate lower NVM bandwidth compared to DRAM, we thermal throttle the NUMA socket [25]. To emulate higher write latency, we use a modified version of NVM emulator [37] and inject delay by estimating the number of processor store cache misses [12, 17, 22]. For our experiments, we emulate 5x higher NVM write latency compared to DRAM access latency and keep the NVM read latency same as the DRAM latency. We vary NVM bandwidth from 2 GB/s to 8 GB/s; the 8 GB/s bandwidth is same as DRAM's per-channel bandwidth and is considered an ideal case.

3 Motivation

NVMs are expected to provide an order of magnitude lower latency and up to 8x higher bandwidth compared to SSDs; but can the current LSM software stack fully exploit the hardware performance benefits of NVM? To understand the impact of using NVM in current LSM designs, we analyze LevelDB's performance by using NVM for its persistent storage. We use the widely-used DBbench [4, 30, 35] benchmark with the total key-value database size set to 16 GB, and the value size set to 4 KB. Figure 2 compares the latency of sequential and random LSM write and read operations. We configure the maximum size of each SSTable file to 64 MB, a feature recently added to LevelDB to improve read performance [4].

As shown in Figure 2, although NVM hardware provides 100x faster read and write compared to SSD, LevelDB's sequential and random insert latency (for 5 GB/sec bandwidth) reduce by just 7x and 4x, respectively; the sequential and random read (fetch) latency reduces by less than 50%. The results show that *current LSMs do not fully exploit the hardware benefits of NVM and suffer from significant software overheads*. We next decipher the sources of these overheads.

Insert latency. A key-value pair insert (or update) operation to LSM is first buffered in the memory – mutable memtable (skip list in LevelDB) – before writing the key-value pair to the storage layer (SSTables). However, a power failure or a system crash can lead to data loss (buffered in memory). To avoid data loss, the key-value pairs and their checksum are first added to a sequential log in the persistent storage before inserting them to the memtable. When the memtable is full, it is made immutable, and a new mutable memtable is created to which new inserts continue. A background thread compacts the immutable memtable to storage; however, if the new mutable memtable also fills up before the completion of background compaction, all new inserts to LSM are stalled. Current LSM designs suffer from high compaction cost because compaction involves iterating the immutable memtable skip list, serializing data to disk-compatible (SSTable) format, and

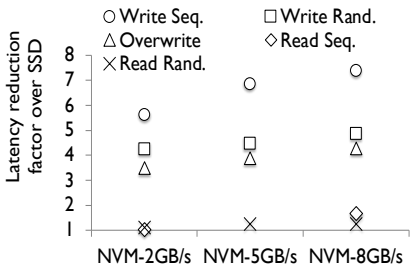

Figure 2: **Latency reduction factor.** *Analysis shows LevelDB using NVM for storage compared to SSD for 4 KB values; x-axis varies NVM bandwidth.*

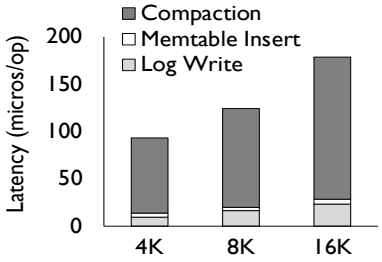

Figure 3: **Write latency cost split-up.** *Compaction happens in the background but stalls LSM when memtables are full and compaction is not complete.*

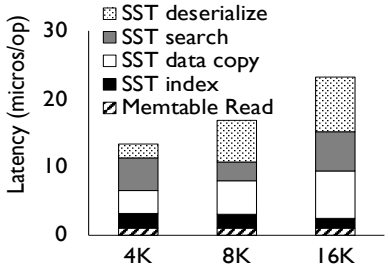

Figure 4: **Read latency cost split-up.** *SST denotes SSTable, and the graph shows time spent on memtable and different SSTable methods.*

finally committing them to the storage. Besides, the storage layer (SSTable) comprises of multiple levels, and the memtable compaction can trigger a chain of compaction across these levels, stalling all foreground updates.

Figure 3 shows the cost breakup for insert operations with 4 KB, 8 KB, and 16 KB values. As shown in the figure, data compaction dominates the cost, increasing latency by up to 83%, whereas log writes and checksum calculations add up to 17% of the total insert latency. Increasing the in-memory buffer (memtable) can reduce compaction frequency; however, this introduces several drawbacks. First, DRAM usage increases by two times: memory must be increased for both mutable and immutable memtables. Second, only after the immutable memtable is compacted, log updates can be cleaned, leading to a larger log size. Third, LSMs such as LevelDB and RocksDB do not enforce commits (sync) when writing to a log; as a result, an application crash or power-failure could lead to data loss. Fourth, a larger log also increases recovery time after a failure. Finally, the cost of checksumming and logging also increases.

Read operation latency. A read operation involves hierarchically searching the smaller in-memory mutable and immutable memtables, followed by searching multiple SSTable levels. Searching a memtable involves traversing the skip list without the need to deserialize data. However, searching a SSTable is complicated for the following reason: the SSTable contains multiple levels that store key-value pairs sorted by their key hash, and each level is searched using a binary search. After locating the blocks containing a key-value pair, the blocks are copied into a memory buffer and then deserialized from disk to an in-memory format. The search cost increases moving top-down across SSTable levels because each SSTable level is at least 10x larger than the previous level. To reduce SSTable search cost, LSMs such as LevelDB and RocksDB maintain an index table at each level which uses a Bloom filter to cache recently searched keys, which is useful only for workloads with high re-access rates (e.g., Zipfian distribution). Figure 4 breaks down the cost of a read operation for 4 KB, 8 KB, and 16 KB values. For small values, searching the SSTable dominates the cost, followed by copying disk blocks to memory and deserializing block contents to in-memory key-value pairs; the deserialization cost increases with increasing value size (e.g., 16 KB). Reducing data copy, search, and deserialization cost can significantly reduce read latencies.

Summary. To summarize, existing LSMs suffer from high software overheads for both insert and read operations and fail to exploit NVM's byte-addressability, low latency, and high storage bandwidth. The insert operations suffer mainly from high compaction and log update overheads, and the read operations suffer from sequential search and deserialization overheads. Reducing these software overheads is critical for fully exploiting the hardware benefits of NVMs.

4 Design

Based on the analyses presented earlier, we first formulate NoveLSM's design principles and then discuss the details on how these principles are incorporated to NoveLSM's design.

4.1 NoveLSM Design Principles

NoveLSM exploits NVMs byte addressability, persistence, and large capacity to reduce serialization and deserialization overheads, high compaction cost, and logging overheads. Further, NoveLSM utilizes NVM's low latency and high bandwidth to parallelize search operations and reduce response time.

Principle 1: Exploit byte-addressability to reduce serialization and deserialization cost. NVMs provide byte-addressable persistence; therefore, in-memory structures can be stored in NVM as-is without the need to serialize them to disk-compatible format or deserialize them to memory format during retrieval. To exploit this, NoveLSM provides a persistent NVM memtable by designing a persistent skip list. During compaction, the DRAM memtable data can be directly moved to the NVM memtable without requiring serialization or deserialization.

Principle 2: Enable mutability of persistent state and leverage large capacity of NVM to reduce compaction cost. Traditionally, software designs treat data in the

storage as immutable due to high storage access latency; as a result, to update data in the storage, data must be read into a memory buffer before making changes and writing them back (mostly in batches). However, NVM byte-addressability provides an opportunity to directly update data on the storage without the need to read them to a memory buffer or write them in batches. To exploit mutability of persistent state, NoveLSM designs a large mutable persistent memtable to which applications can directly add or update new key-value pairs. The persistent memtable allows NoveLSM to alternate between small DRAM and large NVM memtable without stalling for background compaction to complete. As a result compaction cost significantly reduces.

Principle 3: Reduce logging overheads and recovery cost with in-place durability. Current LSM designs must first write updates to a log, compute the checksum and append them, before inserting them into the memtable. Most LSMs compromise crash consistency for performance by not committing the log updates. Further, recovery after an application failure or system crash is expensive; each log entry must be deserialized before adding it to the memtable. In contrast, NoveLSM avoids logging by immediately committing updates to the persistent memtable in-place. Recovery is fast and only requires memory mapping the entire NVM memtable without deserialization.

Principle 4: Exploit the low latency and high bandwidth of NVM to parallelize data read operations. LSM stores data in a hierarchy with top in-memory levels containing new updates, and older updates in the lower SSTables levels. With an increase in the number of key-value pairs in a database, the number of storage levels (i.e., SSTables) increases. Adding NVM memtables further increases the number of LSM levels. LSMs must be sequentially searched from top to bottom, which can add significant search costs. NoveLSM exploits NVMs' low latency and high bandwidth by parallelizing search across the memory and storage levels, without affecting the correctness of read operations.

4.2 Addressing (De)serialization Cost

To reduce serialization and deserialization cost in LSMs, we first introduce an immutable persistent memtable. During compaction, each key-value pair from the DRAM memtable is moved (via *memcpy()*) to the NVM memtable without serialization. The NVM memtable skip list nodes (that store key-value pairs) are linked by their relative offsets in a memory-mapped region instead of virtual address pointers and are committed in-place; as a result, the persistent NVM skip list can be safely recovered and rebuilt after a system failure. Figure 5.a shows the high-level design of an LSM with NVM memtable placed behind DRAM memtable.

Immutable NVM skip list-based memtable. We design a persistent memtable by extending LevelDB's skip list and adding persistence support. A skip list is a multi-dimensional linked-list that provides fast probabilistic insert and search operation avoiding the need to visit all elements of a linked list [33]. Popular LSM implementations, such as LevelDB and RocksDB, use a skip list because they perform consistently well across sequential, random, and scan workloads. We extend the skip list for persistence because it enables us to reuse LevelDB's skip list-specific optimizations such as aligned memory allocation and faster range queries.

In a persistent skip list, the nodes are allocated from a large contiguous memory-mapped region in the NVM. As shown in Figure 5.d, each skip list node points to a next node using physical offset relative to the starting address of the root node, instead of a virtual address. Iterating the persistent skip list requires root node's offset from starting address of the memory-mapped region. After a restart or during failure recovery, the persistent region is remapped, and the root offset is recovered from a log file; using the root node, all skip list nodes are recovered.

To implement a persistent skip list, we modify LevelDB's memtable with a custom persistent memory NVM allocator that internally uses the Hoard allocator [10]. Our allocator internally maps a large region of NVM pages on a DAX filesystem [6] and manages the pages using persistent metadata similar to Intel's NVML library [24]. Each skip list node maintains a physical offset pointer and a virtual address pointer to the next node, which are updated inside a transaction during an insert or update operation, as shown in Figure 5.d. A power or application failure in the middle of a key-value pair insertion or the offset update can compromise durability. To address this, we provide ordered persistent updates by using hardware memory barriers and cacheline flush instructions [15, 16, 22, 38]. Note that NoveLSM extends existing LSMs for NVMs rather than completely redesigning their data structures; this is complementary to prior work that focuses on optimizing LSMs' in-memory data structures [9, 34].

4.3 Reducing Compaction Cost

Although the immutable NVM design can reduce serialization cost and read latency, it suffers from several limitations. First, the NVM memtable is just a replica of the DRAM memtable. Hence, the compaction frequency is dependent on how fast the DRAM memtables fill. For applications with high insert rates, compaction cost dominates the performance.

Mutability for persistent memtable. To address the issue of compaction stalls, NoveLSM makes the NVM memtable mutable, thereby allowing direct updates to the NVM memtable (Figure 5.(b)); when the in-memory

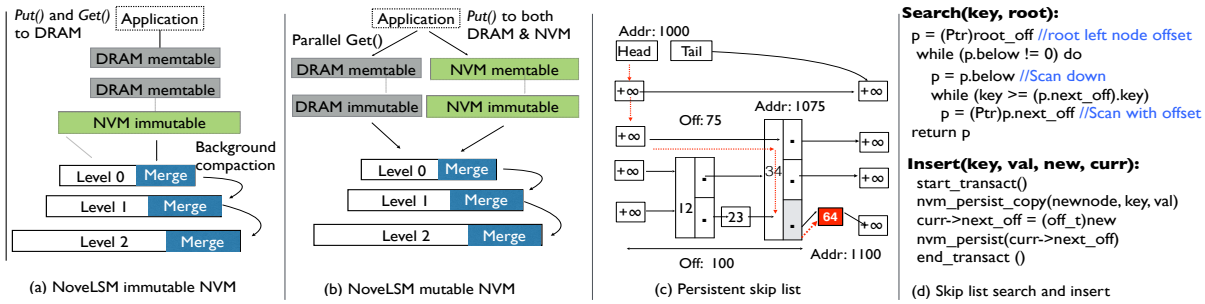

Figure 5: **NoveLSM's immutable and mutable NVM memtable design.** *(a) shows the immutable memtable design, where NVM memtable is placed below DRAM memtable to only store compaction data. (b) shows mutable memtable design where inserts can be directly updated to NVMs persistent skip list. (c) shows an example of inserting a key 64 to persistent NVM memtable at offset 100 from the root node with address 1000. Next pointer of node with key 34 contains offset 100. (d) shows NVM persistent skiplist search and insert pseudocode. Read operation searches for a key using the offset of node instead of pointer. Offsets represent the location of a node in a memory-mapped file. Inserts to NVM memtable are committed in-place and avoid separate logging.*

memtable is full, application threads can alternate to using the NVM memtable without stalling for the in-memory memtable compaction to complete.

The working of the mutable NVM design can be summarized as follows. During initialization, NoveLSM creates a volatile DRAM memtable and a mutable persistent NVM memtable. Current LSM implementations use a smaller memtable size to reduce DRAM consumption and avoid data loss after a failure. In contrast, NoveLSM uses a large NVM memtable; this is because NVMs can scale up to 4x larger than DRAM and also maintain persistence. To insert a key-value pair, first, the DRAM memtable is made active; the key-value pairs and their checksum are written to a log and then inserted into the DRAM memtable. When the DRAM memtable is full, it is made immutable, and the background compaction thread is notified to move data to the SSTable. Instead of stalling for compaction to complete, NoveLSM makes NVM memtable active (mutable) and keys are directly added to mutable NVM memtable. The large capacity of NVM memtable provides sufficient time for background compaction of DRAM and NVM immutable memtables without stalling foreground operations; as a result, NoveLSM's mutable memtable design significantly reduces compaction cost leading to lower insert/update latency. For read operations, the most recent value for a key is fetched by first searching the current active memtable, followed by immutable memtables and SSTables.

4.4 Reducing Logging Cost

NoveLSM eliminates logging for inserts added to mutable NVM memtable by persisting updates in-place. As a result, NoveLSM reduces the number of writes for each key-value pair and also reduces recovery time. We next discuss the details.

Logging. In current LSM implementations, each key-value pair and its 32-bit CRC checksum is first appended to a persistent log, then inserted into the DRAM memtable, and finally compacted to an SSTable, leading to high write amplification. Further, popular implementations such as LevelDB and RocksDB only append but do not commit (*fsync()*) data to the log; as a result, they compromise durability for better performance.

In contrast, for NoveLSM, when inserting into the mutable persistent memtable in NVM, all updates are written and committed in-place without requiring a separate log; as a result, NoveLSM can reduce write amplification. Log writes are avoided only for updates to NVM memtable, whereas, all inserts to the DRAM memtable are logged. Our evaluations show that using a large NVM memtable with direct mutability reduces logging to a small fraction of the overall writes, thereby significantly reducing logging cost and recovery time. Additionally, because all NVM updates are committed in-place, NoveLSM can provide stronger durability guarantees compared to existing implementations. Figure 5.d shows the pseudocode for NVM memtable insert. First, a new transaction is initiated and persistent memory for a key-value pair is allocated. The key-value pair is copied persistently by ordering the writes using memory store barrier and cache line flush of the destination addresses, followed by a memory store barrier [24, 38, 41]. As an additional optimization, small updates to NVM (8-byte) are committed with atomic store instruction not requiring a barrier. Finally, for overwrites, the old nodes are marked for lazy garbage collection.

Recovery. Recovering from a persistent NVM memtable requires first mapping the NVM memtable (a file) and identifying the root pointer of the skip list. Therefore, the NVM memtable root pointer offset and 20-bytes of skip list-related metadata are stored in a separate file. With a volatile DRAM and persistent NVM memtable, a failure can occur while a key with version V_i in the persistent NVM memtable is getting overwritten in the DRAM memtable to version V_{i+1}. A failure can also occur while a key with version V_i in the DRAM memtable, not yet compacted to storage, is getting overwritten to version V_{i+1} in the NVM. To maintain correctness, NoveLSM must always recover from the greatest committed version

of the key. To achieve version correctness, NoveLSM performs the following steps: (1) a new log file is created every time a new DRAM memtable is allocated and all updates to DRAM memtable are logged and made persistent; (2) when the NVM memtable (which is a persistent skip list on a memory-mapped file) is made active, inserts are not logged, and the NVM memtable is treated as a log file; (3) the NVM log files are also named with an incremental version number similar to any other log file. During LSM restart or failure recovery, NoveLSM starts recovering from the active versions of log files in ascending order. A key present in a log file log_{i+1}, which could be either a DRAM log or NVM memtable, is considered as the latest version of the key. Note that recovering data from NVM memtable only involves memory-mapping the NVM region (a file) and locating the skip list root pointer. Therefore, recovering from even a large NVM memtable is fast with almost negligible cost of mapping pages to the page tables.

4.5 Supporting Read Parallelism

NoveLSM leverages NVM low latency and high bandwidth to reduce the latency of each read operation by parallelizing search across memtables and SSTables. In this pursuit, NoveLSM does not compromise the correctness of read operation. In current LSMs, read operations progress top-down from memtable to SSTables. A read miss at each level increases read latency. Other factors such as deserializing data from the SSTable also add to read overheads.

Reducing read latency. To reduce read latency, NoveLSM takes inspiration from the processor design, which parallelizes cache and TLB lookup to reduce memory access latency for cache misses; NoveLSM parallelizes search across multiple levels of LSM: DRAM and NVM mutable memtables, DRAM and NVM immutable tables, and SSTables. Our design manages a pool of worker threads that search memtables or the SSTable. Importantly, NoveLSM uses only one worker thread for searching across the mutable DRAM and NVM memtable because of the relatively smaller DRAM memtable size compared to the NVM memtable.

With this design, the read latency is reduced from $T_{read} \approx T_{mem_{DRAM}} + T_{mem_{NVM}} + T_{imm} + T_{SST}$ to $T_{read\_parallel} \approx max(T_{mem_{DRAM}} + T_{mem_{NVM}}, T_{imm}, T_{SST}) + C$. $T_{mem_{DRAM}}$, $T_{mem_{NVM}}$, T_{imm}, and T_{SST} represent the read time to search across the DRAM and NVM mutable memtable, the immutable memtable, and the SSTable, and C represents a constant corresponding to the time to stop other worker threads once a key has been found.

Guaranteeing version correctness for reads. Multiple versions of a key can exist across different LSM levels, with a newer version (V_{i+1}) of the key at the top LSM level (DRAM or NVM mutable memtable) and older versions ($V_i, V_{i-1}, ...$) in the lower immutable memtable and SSTables. In traditional designs, search operations sequentially move from the top memtable to lower SSTables, and therefore, always return the most recent version of a key. In NoveLSM, search operations are parallelized across different levels and a thread searching the lower level can return with an older version of the key; this impacts the correctness of read operation. To guarantee version correctness, NoveLSM always considers the value of a key returned by a thread accessing the highest level of LSM as the correct version. To satisfy this constraint, a worker thread T_i accessing L_i is made to wait for other worker threads accessing higher levels L_0 to L_{i-1} to finish searching, and only if higher levels do not contain the key, the value fetched by T_i is returned. Additionally, stalling higher-level threads to wait for the lower-level threads to complete can defeat the benefits of parallelizing read operation. To overcome this problem, in NoveLSM, once a thread succeeds in locating a key, all lower-level threads are immediately suspended.

Optimistic parallelism and management. Introducing parallelism for each read operation is only beneficial when the overheads related to thread management cost are significantly lower than the actual cost to search and read a key-value pair. NoveLSM uses an optimistic parallelism technique to reduce read latency.

Thread management cost. In NoveLSM, the main LSM thread adds a client's read request to a job pool, notifies all worker threads to service the request, and finally, returns the value for a key. NoveLSM always colocates the master and the worker threads to the same CPU socket to avoid the lock variable bouncing across processor caches on different sockets. Further, threads dedicated to parallelize read operation are bound to separate CPUs from threads performing backing compaction. These simple techniques are highly effective in reducing thread management cost.

Optimistic parallelism. While the thread pool optimizations reduce overheads, using multiple threads for keys that are present in DRAM or NVM memtable only adds more overheads. To avoid these overheads, we implement a Bloom filter for NVM and DRAM memtable. The Bloom filter predicts likeliness of a key in the memtable, and read parallelism is enabled only when a key is predicted to miss the DRAM or NVM memtable; false positives (keys that are predicted to be in memtable but are not present) only make the read operations sequential without compromising correctness.

5 Evaluation

Our evaluation of NoveLSM aims to demonstrate the design insights in reducing write and read latency and increasing throughput when using NVMs. We answer the following important questions.

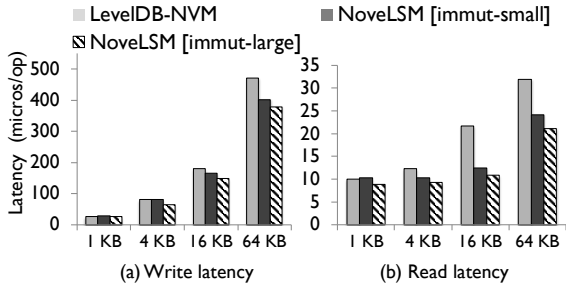

Figure 6: **Write and Read latency.** *Impact of NVM immutable for random writes and reads. Database size is constant.*

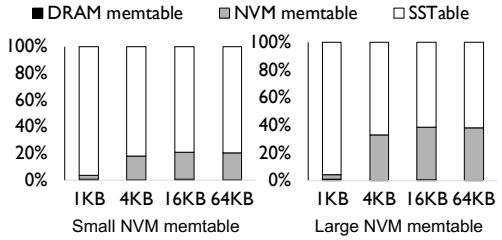

Figure 7: **NoveLSM immutable memtable hits.** *Figure shows percentage split of keys read from different LSM levels when using the immutable memtable design.*

1. What are the benefits of introducing a persistent immutable NVM memtable for different access patterns?
2. Does enabling mutability for NVM memtable reduce compaction cost and improve performance?
3. How effective is NoveLSM's optimistic parallelism in reducing read latency?
4. What is the impact of splitting NoveLSM across NVM and SSD compared to state-of-the-art approaches?
5. Is NoveLSM effective in exploiting NVMs byte-addressability to make failure recovery faster?

We first describe our evaluation methodology, and then evaluate NoveLSM with benchmarks and realistic workloads.

5.1 Methodology and Workloads

For our evaluation, we use the same platform described earlier § 2.4. NoveLSM reserves and uses 16 GB (a memory socket) to emulate NVM with 5 GB/sec NVM bandwidth and the read and write latency set to 100ns and 500ns respectively, similarly to [12, 17, 22], using methodology described earlier. We evaluate NoveLSM using DBbench [3, 4, 30] and the YCSB cloud benchmark [18]. The total LSM database size is restricted to 16 GB to fit in the NUMA socket that emulates NVM. The key size (for all key-values) is set to 16 bytes and only the value size is varied. We turn off database compression to avoid any undue impact on the results, as done previously [30].

5.2 Impact of NVM-immutable Memtable

We begin by evaluating the benefits and implications of adding a persistent NVM immutable to the LSM hierarchy. We study two versions of NoveLSM: NoveLSM with a small (2 GB) immutable NVM memtable (NoveLSM+immut-small), and NoveLSM with a large (4 GB) immutable NVM memtable (NoveLSM+immut-large). The remaining NVM space is used for storing SSTables. For comparison, we use a vanilla LevelDB that stores all its non-persistent data in a DRAM memtable and persistent SSTables in the NVM (LevelDB-NVM). Figures 6.a and 6.b show the average random write and read latency as a function of the value sizes in X-axis.

Random write latency. Figure 6.a compares the random write latency. For the naive LevelDB-NVM, when the in-memory (DRAM) immutable memtable is full, a compaction thread first serializes data to SSTable. In contrast, NoveLSM uses a persistent NVM immutable memtable (a level below the 64 MB DRAM immutable memtable). When the DRAM immutable memtable is full, first data is inserted and flushed to NVM memtable skip list without requiring any serialization. When NVM memtable is also full, its contents are serialized and flushed to SSTable by a background thread. Using a larger NVM memtable (NoveLSM+immut-large) as a buffer reduces the memory to disk format compaction cost but without compromising crash consistency. Therefore, the NVM immutable design achieves up to 24% reduction in latency for 64 KB value compared to LevelDB-NVM. However, due to lack of direct NVM memtable mutability, the compaction frequency is dependent on the DRAM memtable capacity, which impacts immutable NVM designs performance.

Random read latency. Figure 6.b shows the read latency results. In case of LevelDB-NVM, reading a key-value pair from SSTable requires first locating the SSTable level, searching for the key within a level, reading the corresponding I/O blocks, and finally deserializing disk blocks to in-memory data. NoveLSM's immutable memtable skip list also incurs search cost; however, it avoids indexing, disk block read, and deserialization cost. Figure 7 shows the NVM immutable table hit rate for different value sizes when using small and large NVM tables. For 4 KB value size, the memtable hit rate (DRAM or NVM) for small NVM memtable is less than 17% and the additional search in the NVM memtable increases latency. However, for NoveLSM+immut-large, the hit rate is around 29% and the read latency reduces by 33% compared to LevelDB-NVM. Because we keep the database size constant and vary the value size, for larger value sizes (e.g., 64 KB), the number of key-values in the database is less, increasing hit rate by up to 38% and reducing latency by up to 53%. For single-threaded DBbench, the throughput gains are same as latency reduction gains; hence, we do not show throughput results.

Summary. NoveLSM's immutable memtable reduces write latency by 24% and read latency by up

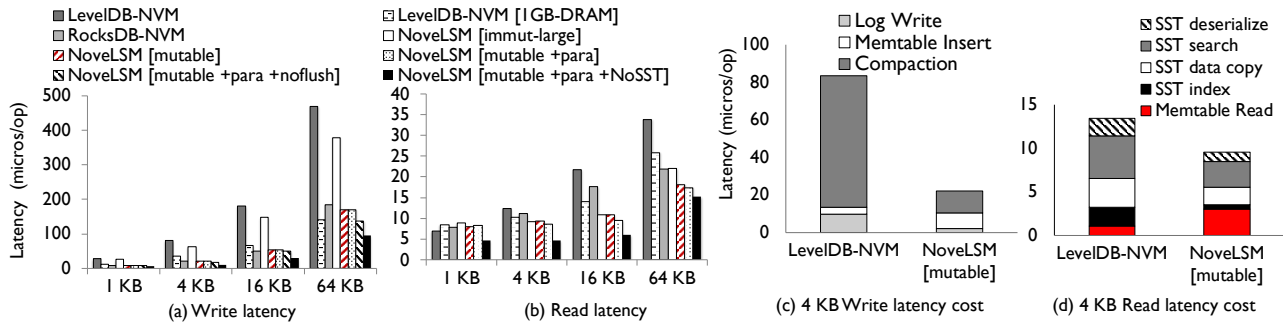

Figure 8: **NVM mutability impact.** *Figure shows (a) write latency, (b) read latency. LevelDB-NVM and RocksDB-NVM use NVM for SSTable. LevelDB-NVM [1GB-DRAM] uses a large DRAM memtable; [mutable + para] shows read parallelism. [mutable + para + noflush] shows NoveLSM without persistent flush, [mutable + para + NoSST] shows using only NVM memtable without SSTable. Figure (c) and (d) show NoveLSM Write and Read operation latency cost splitup for 4 KB values.*

to 53%. Lack of direct NVM memtable mutability and frequent compaction impacts write performance.

5.3 NVM Memtable Mutability

To understand the effectiveness of NoveLSM's mutable memtable in reducing compaction cost, we begin with NoveLSM+immut-large discussed in the previous result, and analyze four other NoveLSM techniques: NoveLSM+mutable uses a large (4 GB) NVM memtable which is placed in parallel with the DRAM memtable and allows direct transactional updates (without logging) supported by persistent processor cache flushes; NoveLSM+mutable+para enables read parallelism; NoveLSMmutable+para+noflush shows the latency without persistent processor cache flushes; and finally, NoveLSM+mutable+NoSST uses only persistent NVM memtable for the entire database without SSTables. For comparison, in addition to LevelDB-NVM, we also compare the impact of increasing the vanilla LevelDB-NVM DRAM memtable size to 1GB (LevelDB-NVM+1GB-DRAM) and RocksDB-NVM [3] by placing all its SSTable in NVM. RocksDB-NVM is configured with default configuration values used in other prior work [9, 30]. For our experimentation, we set the DRAM memtable to 64 MB for all configuration except LevelDB-NVM+1GB-DRAM. Figure 8.c and Figure 8.d show the cost split up for a 4 KB random write and read operation.

Write performance. Figure 8.a shows the average write latency as a function of value size. When the mutable NVM memtable is active, its large capacity provides background threads sufficient time to finish compaction, consequently reducing foreground stalls. For 4 KB values, NoveLSM+mutable reduces latency by more than 3.8x compared to LevelDB-NVM and NoveLSM+immut-large, due to reduction of both compaction and log write cost as shown in Figure 8.c. For 64 KB value size, write latency reduces by 2.7x compared to LevelDB-NVM. While increasing the vanilla LevelDB-NVM's DRAM memtable size (1GB-DRAM) improves performance, however, (1) DRAM consumption increases by twice (DRAM memtable and immutable table), (2) increases log size and recovery time (discussed shortly), and importantly, (3) compromises crash consistency because both LevelDB and RocksDB do not commit log updates to storage by default.

For smaller value sizes, RocksDB-NVM marginally reduces write latency compared to mutable NoveLSM (NoveLSM+mutable) design that provides in-place commits (with processor cache flushes). RocksDB benefits come from using a Cuckoo hash-based SST [11] that improves random lookups (but severely impacts scan operations), parallel compaction to exploit SSD parallelism, and not flushing log updates to storage. While incorporating complementary optimizations, such as Cuckoo hash-based SST and parallel compaction, can reduce latency, even avoiding persistent cache flush (NoveLSM+mutable+para+noflush) reduces latency compared to RocksDB. For larger values, NoveLSM reduces latency by 36% compared to RocksDB-NVM providing the same durability guarantees. Finally, NoveLSM+mutable+NoSST, by using large NVM memtable and adding the entire database to the skip list, eliminates compaction cost reducing the write latency by 5x compared to LevelDB-NVM and more than 1.9x compared to RocksDB-NVM.

Read parallelism. Figure 8.b shows the read latency for all configurations. First, compared to LevelDB-NVM, NoveLSM+mutable reduces read latency for 4 KB value size by 30%. RocksDB-NVM with a random-access friendly Cuckoo-hash SSTable significantly reduces memtable miss latency cost, providing better performance for smaller values. For smaller values (1 KB, 4 KB), NoveLSM's optimistic parallelism shows no gains compared to RocksDB because the cost of thread management suppresses benefits of parallel read. However, for larger value sizes, NoveLSM's parallelism combined with the reduction in deserialization cost reduces NoveLSM's read latency by 2x and 24% compared to LevelDB-NVM and RocksDB respectively. In-

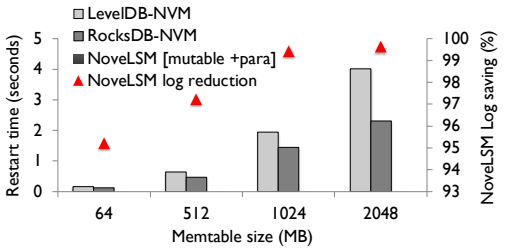

Figure 9: **Failure recovery performance.** *The figure shows recovery time as a function of memtable size in X-axis. For LevelDB and RocksDB, DRAM memtable size is increased, whereas for NoveLSM, NVM memtable increased, and DRAM memtable size is kept constant at 64 MB.*

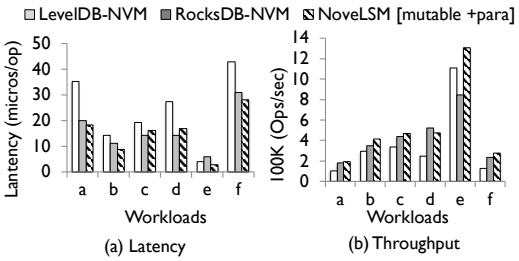

Figure 10: **YCSB (a) latency and (b) throughput.** *Results only shown for run-phase after warm-up. NoveLSM's mutable memtable size set to 4 GB. Workload A has 50-50% update-read ratio, B is read-intensive with 95% reads and 5% updates (overwrites); C is read-only, D is also read-only, with the most recently inserted records being most popular, E is scan-intensive (95% scan, and 5% insert), and F has 50% reads and 50% write-modify-reads.*

corporating RocksDB's optimized SSTable can further improve NoveLSM's read performance. As a proof, the NoveLSM-NoSST case reduces the read latency by 45% compared to RocksDB.

Splitting LSMs across NVM and SSDs. NoveLSM can support large LSMs that spill over to SSD when NVM capacity is full. To understand the performance impact, we set the LSM database size to 16 GB. We compare two approaches: (1) LevelDB-NVM-SSD that splits SSTable across NVM (8 GB) and SSD (8 GB), (2) NoveLSM-mutable-SSD that uses a half-and-half configuration with 4 GB NVM for mutable memtable, 4 GB NVM for higher levels of SSTable, and 8 GB SSTable on SSD. We do not consider RocksDB because of the complexity involved in supporting multiple storage devices for a complex compaction mechanism, which is beyond the scope of this work. When evaluating the two configurations, we determine that LevelDB-NVM-SSD suffers from high compaction cost. For larger value sizes, memtables fill-up quickly, triggering a chain of compaction across both NVM and SSD SSTables. In contrast, NoveLSM's mutable NVM memtable reduces compaction frequency allowing background threads with sufficient time to compact, thus reducing stalls; consequently, NoveLSM reduces latency by more than 45% for 64 KB values compared to LevelDB-NVM-SSD.

Summary. The results highlight the benefits of using a mutable memtable for write operations and supporting parallelism for read operations in both NVM-only and NVM+SSD configurations. Incorporating RocksDB's SSTable optimizations can further improve NoveLSM's performance.

5.4 Failure Recovery

NoveLSM's mutable persistence provides in-place commits to NVM memtable and avoids log updates. In Figure 9, we analyze the impact of memtable size on recovery time after a failure. To emulate failure, we crash DBbench's random write workload after inserting half the keys. On the X-axis, for LevelDB-NVM and RocksDB-NVM, we increase DRAM memtable size, whereas, for NoveLSM-mutable, the DRAM memtable size is kept constant at 64 MB and only the NVM mutable memtable size is varied.

For LevelDB-NVM and RocksDB-NVM, all updates to DRAM memtable are also logged; hence, increasing the DRAM memtable size also increases the log size that must be read during recovery, thereby increasing the recovery time. Recovery involves iterating the log, verifying checksums, serializing logged key-value pairs to an SSTable disk block, and inserting them to the top level of the SSTable which is merged with lower levels. As a result, for a 1 GB DRAM memtable size, LevelDB-NVM's recovery is as high as 4 seconds; RocksDB-NVM recovers faster than LevelDB due to its specialized SST format. For NoveLSM, recovery involves identifying a correct version of the persistent memtable before the crash, memory-mapping the NVM memtable's persistent skip list, and modifying the root pointer to the current virtual address of the application. As a result, restart performance for NoveLSM is more than three orders faster. Importantly, NoveLSM logs only the updates to DRAM memtable, thereby reducing logging writes by up to 99%.

5.5 Impact on YCSB

To understand the benefits and implication for cloud workloads, we run the widely-used YCSB [18] benchmark and compare LevelDB-NVM, RocksDB-NVM, and NoveLSM-mutable-para approaches. We use the six workloads from the YCSB cloud suite with different access patterns. YCSB has a warm-up (write-only) and a run phase, and we show the run phase results when using 4-client threads. Figure 10 shows the 95th percentile latency and throughput (in 100K operations per second). We use 4 KB value size and 16 GB database. The SSTables are placed in NVM for all cases, and NoveLSM's mutable memtable is set to 4 GB. First, for workload A, with the highest write ratio

(50%), NoveLSM's direct mutability improves throughput by 6% over RocksDB and 81% over LevelDB-NVM, even for small 4 KB value sizes. Both workload B and workload C are read-intensive, with high random reads. NoveLSM's read parallelism is effective in simultaneously accessing data across multiple LSM levels for four client threads. For workload D, most accesses are recently inserted values, resulting in high mutable and immutable memtable hits even for RocksDB. NoveLSM checks the bloom filter for each access for enabling read parallelism (parallelism is not required as keys are present in memtable), and this check adds around $1\mu s$ overhead per key resulting in a slightly lower throughput compared to RocksDB. Next, for the scan-intensive workload E, LevelDB and NoveLSM's SSTable are highly scan-friendly; in contrast, RocksDB's SSTable optimized for random-access performs poorly for scan operations. As a result, NoveLSM shows 54% higher throughput compared to RocksDB-NVM. Finally, workload F with 50% updates (overwrites) adds significant logging and compaction-related serialization overhead. NoveLSM's direct mutability reduces these cost improving throughput by 17% compared to RocksDB and more than 2x over LevelDB-NVM.

Summary. NoveLSM's direct mutability and read parallelism provide high-performance for both random and sequential workloads.

6 Related Work

Key-value store and storage. Prior works such as SILT [29], FlashStore [20], SkimpyStash [21] design key-value stores specifically targeting SSDs. FlashStore and SkimpyStas treat flash as an intermediate cache and place append-only logs to benefit from the high sequential write performance of SSD. SILT reduces DRAM usage by splitting in-memory log across the DRAM and SSD and maintaining a sorted log index in the memory. In summary, prior works enforce sequentiality by batching and adding software layers for improving throughput. In contrast, we design NoveLSM to reduce I/O access latency with heap-based persistent structures.

Application redesign for persistent memory. Byte addressability, low latency, and high bandwidth make NVMs a popular target for redesigning data structures and applications originally designed for block storage. Venkatraman et al. [36] were one of the first to explore the benefits of persistence-friendly B-trees for NVMs. Since then, several others have redesigned databases [8], key-value stores [31], B-trees [14], and hashtables [19].

LSM and redesign for storage. Several prior works have redesigned LSMs for SSD. Wang et al [39] expose SSD's I/O channel information to LevelDB to exploit the parallel bandwidth usage. WiscKey [30] redesigns LSMs for reducing the read and write amplification and exploiting SSD bandwidth. VT-tree [35] design proposes a file system and a user-level key-value store for workload-independent storage. In NoveLSM, we reduce the write latency with a mutable persistent skip list and the read latency by parallelizing reads across the LSM levels.

LSM redesign for NVM. NoveLSM is focused on extending existing LSMs for NVMs rather than completely redesigning their data structures; this is complementary to projects such as FloDB and PebblesDB [9, 34]. We were recently made aware of a concurrently developed effort with similar goals as NoveLSM. NVM-Rocks [28] shares similar ideas on using a persistent mutable memtable to reduce access latencies and recovery costs. To improve read latencies, it introduces a hierarchy of read caches. NoveLSM retains the in-DRAM memtable of the original LSM design, benefiting latencies for both cached reads and writes, and introduces parallelism *within* read operations to reduce read latency. We look forward to gaining access to NVMrocks and analyzing the tradeoffs that each technique contributes to the overall LSM performance.

7 Conclusion

We present NoveLSM, an LSM-based persistent key-value store that exploits NVM byte-addressability, persistence, and large capacity by designing a heap-based persistent immutable NVM skip list. The immutable NVM skip list facilitates DRAM memtable compaction without incurring memory to I/O data serialization cost and also accelerates reads. To reduce the compaction cost further, we introduce direct mutability of NVM memtables, which allow applications can to directly commit data to NVM memtable with stronger durability and avoid logging. Reducing compaction and logging overheads reduces random write latency by up to 3.8x compared to LevelDB running on NVM. To reduce read latency, we design opportunistic parallelism, which reduces read latency by up to 2x. Finally, the persistent memtable makes the restarts three orders of magnitude faster. As storage moves closer to memory, and storage bottlenecks shifts towards software, increased effort to optimize such software will undoubtedly be required to realize further performance gains.

Acknowledgements

We thank the anonymous reviewers and Michio Honda (our shepherd) for their insightful comments. This material was supported by funding from DoE Office of Science Unity SSIO and NSF grant CNS-1421033. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Apache Cassandra. `http://cassandra.apache.org/`.

[2] Apache HBase. `http://hbase.apache.org/`.

[3] Facebook RocksDB. `http://rocksdb.org/`.

[4] Google LevelDB . `http://tinyurl.com/osqd7c8`.

[5] Intel-Micron Memory 3D XPoint. `http://intel.ly/1eICR0a`.

[6] Linux DAX file system. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[7] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[8] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, 2015), SIGMOD '15.

[9] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia, 2017), EuroSys '17.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA, 2000), ASPLOS IX.

[11] BORTHAKUR, D. RocksDB Cuckoo SST. `http://rocksdb.org/blog/2014/09/12/cuckoo.html`.

[12] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43.

[13] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008).

[14] CHEN, S., AND JIN, Q. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow. 8*, 7 (Feb. 2015).

[15] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* (Farmington, PA, November 2013).

[16] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS XVI.

[17] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA, 2009), SOSP '09.

[18] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA, 2010), SoCC '10.

[19] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting Hash Table Design for Phase Change Memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (Monterey, California, 2015), INFLOW '15.

[20] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1414–1425.

[21] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece, 2011), SIGMOD '11.

[22] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands, 2014), EuroSys '14.

[23] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France, 2015), EuroSys '15.

[24] INTEL. Logging library. `https://github.com/pmem/nvml`.

[25] KANNAN, S., GAVRILOVSKA, A., GUPTA, V., AND SCHWAN, K. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada, 2017), ISCA '17.

[26] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom, 2016), EuroSys '16.

[27] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Phase Change Memory in Enterprise Storage Systems: Silver Bullet or Snake Oil? *SIGOPS Oper. Syst. Rev. 48*, 1 (May 2014), 82–89.

[28] LI, J., PAVLO, A., AND DONG, S. NVMRocks: RocksDB on Non Volatile Memory Systems, 2017. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[29] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, 2011), SOSP '11.

[30] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA, 2016), FAST'16.

[31] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-value Store. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage'14.

[32] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Inf. 33*, 4 (June 1996).

[33] PUGH, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM 33*, 6 (June 1990).

[34] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[35] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building Workload-independent Storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA, 2013), FAST'13.

[36] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (San Jose, California, 2011), FAST'11.

[37] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. NVM Quartz emulator. `https://github.com/HewlettPackard/quartz`.

[38] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA, 2011), ASPLOS XVI.

[39] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands, 2014), EuroSys '14.

[40] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, 2015), USENIX ATC '15.

[41] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California, 2013), MICRO-46.