Redesigning LSMs for Nonvolatile Memory with NoveLSM

Sudarsun Kannan*, Nitish Bhat+, Ada Gavrilovska+, Andrea Arpaci-Dusseau**, Remzi Arpaci-Dusseau**
Rutgers University*, Georgia Tech+, UW-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 RocksDB, BigTable, LevelDB, HBase, and Cassandra 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 in the critical path related to logging and compaction costs. While logging updates to storage are necessary for crash-recovery, compaction is required to restrict LSM's DRAM buffer size and to commit data to storage. Most recent proposals have mostly focused on redesigning LSMs for SSDs to improve throughput [2].

Adding byte-addressable, persistent, and fast nonvolatile 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. With storage bottlenecks shifting from the hardware to the software stack, it is becoming critical to reduce and eliminate software storage overheads. When contrasting NVMs to current storage technologies, such as flash memory and hard-disks, NVMs exhibit the following properties: (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. We believe it worthwhile to explore how to redesign LSMs to work well with NVM for the following reasons. In addition to providing backward compatibility, NVMs are expected to co-exist with large-capacity SSDs for the next few years similar to the co-existence of SSDs and hard disks, making the redesign of LSMs for heterogeneous storage important without losing their SSD and hard disk optimizations. Importantly, the benefits of batched, sequential writes is important even for NVMs, given the 5x-10x higher-than-DRAM write latency. Hence, we focus on redesign existing LSM implementations.

Our redesign of LSM technology for NVM focuses on the following three critical problems. First, existing LSMs maintain different data representation in-memory (called memtables) and on persistent storage (called String Sorted Tables). 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 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** [1], 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.

We build NoveLSM by redesigning LevelDB, a widely-used LSM-based key-value store. NoveLSM's design principles can be easily extended to other LSM implementations such as RocksDB. Our analysis reveals that NoveLSM significantly outperforms traditional LSMs when running on an emulated NVM device. Evaluation of NoveLSM with the popular DBbench 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 5xand 1.9xgains 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.

The full paper of this abstract appeared at USENIX ATC 2018 [1]. The source code is available at $https://github.com/SudarsunKannan/lsm\_nvm$

2. NoveLSM Design Principles

NoveLSM exploits NVMs byte addressability, persistence, and large capacity to reduce (1) serialization and deserialization overheads, (2) high compaction cost, (3) logging overheads, and exploits NVM's low latency and higher bandwidth (4) to parallelize search operations and reduce response time. As we briefly describe our design insights, we also demonstrate their performance benefits in Figure 2, by comparing NoveLSM against vanilla LevelDB (LevelDB-NVM) and RocksDB (RocksDB-NVM) that use NVM to store on-disk data structures (SSTables). We emulate NVM on a DAX file system, and emulate read and write latency of 100ns and 500ns respectively, with NVM bandwidth set to 2GB/sec.

Principle 1: NVM's byte-addressability to reduce (de)serialization 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, in NoveLSM, we first design a large persistent immutable NVM memtable by designing a persistent skip list (a multi-dimensional linked-list with probabilistic insert and search). To provide crash-consistency and recover after a system/power failure, updates to the immutable persistent skip list are committed in-place with the aid of hardware memory barriers and cacheline flush instructions. When the DRAM memtable (which is smaller) is full, data is compacted directly to the NVM memtable without serializing them. We design persistent skip list by memory-mapping a large file in NVM; the 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 updated and committed in-place. Figure 1 (a) shows the high-level design of an LSM with large NVM memtable placed behind DRAM memtable. As shown in Figure 2, the immutable NVM memtable (NoveLSM [immut-large]) improves LSM write and read performance compared to LevelDB by avoiding serialization but does not solve the problem of high compaction cost, as the compaction frequency is dominated by DRAM memtable size.

Principle 2: Persistence mutability to reduce compaction. Traditionally, software designs treat data in storage as immutable due to high storage access latency; as a result, to update data in storage, data must be read into a memory buffer before making changes and compacting them back to storage. In the NVM immutable design, the frequency of compaction is dominated by how fast the DRAM memtables fill and fails to utilize the NVM's large byte addressable capacity. While increasing the DRAM memtable capacity can reduce compactions, updates to volatile DRAM must be logged to recover from a failure and the recovery cost increases with DRAM memtable size. To overcome these challenges, we design a mutable NVM memtable (NoveLSM [mutable]) that 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. NoveLSM designs a large mutable persistent memtable to which applications can directly add or update new key-value pairs as shown in Figure 1 (b). As a result, applications can alternate between a small DRAM and a large NVM memtable without stalling for background compaction to complete. To breifly summarize the working, during initialization, NoveLSM creates a volatile DRAM memtable and a large mutable persistent NVM (large because NVMs can scale 4x larger than DRAM). Applications first insert key-value pairs to the DRAM memtable (and also write to a log); when the DRAM memtable is full, a background compaction thread is notified to move data to the SSTable and instead of stalling for compaction, the NVM memtable is made active, to which inserts continue. The large capacity of the NVM provides sufficient time for background compaction without impacting correctness. As shown in Figure 2, NoveLSM [mutable] significantly improves read and write performance compared to LevelDB-NVM. Although, for smaller value sizes, RocksDB-NVM marginally reduces write latency with features such as Cuckoo hash-based storage for on-disk structure optimized for random access (but significantly slow on range queries) and parallel compaction, for large values, such features are not beneficial, resulting in NoveLSM outperforming RocksDB by 36%. Finally, NoveLSM+mutable+NoSST, an idealistic design, avoids disk-structures (SSTable) using a very large NVM memtable, and outperforms all the approaches by up to 1.9x.

Principle 3: Reduce logging and accelerate recovery with in-place durability. Current LSM designs must first write

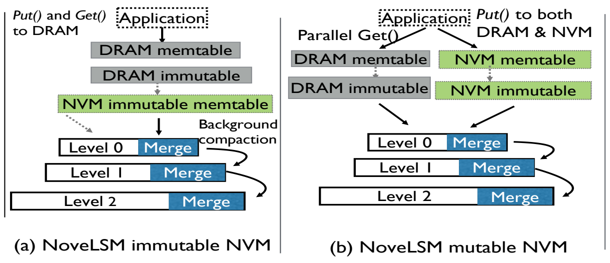

Figure 1: **NoveLSM's (a) immutable and (b) mutable NVM memtable design.** (a) shows the immutable memtable design, where NVM memtable is placed below DRAM memtable to only store compaction data. (b) shows mutable memtable design where inserts can be directly updated to NVMs persistent skip list.

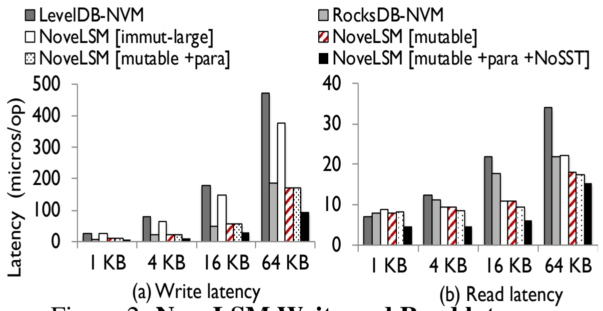

Figure 2: **NoveLSM Write and Read latency.**

updates to a log, compute the checksum and append them, before inserting them into the memtable. In fact, popular LSMs implementations such as LevelDB and RocksDB do not force log commits, and trade crash consistency for better performance. Increasing the log size also increases recovery cost. To reduce logging cost and accelerate recovery, NoveLSM avoids logging by committing updates to the NVM memtable in-place. This also provides stronger durability. Recovery is fast and only requires memory mapping the entire NVM memtable without deserialization cost. Only updates to the DRAM memtable are logged. To preserve version correctness of key-value pairs across DRAM logs and the NVM memtable, NoveLSM treats the memory-mapped NVM memtable also as a log file and starts recovery from the latest version of the log which could be either the DRAM log or NVM memtable (more details about recovery in the full paper [1]). NoveLSM substantially reduces log writes and improves recovery by more than 100x.

Principle 4: Read parallelism by exploiting NVM's low latency and high bandwidth. 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 to the LSM hierarchy further increases the cost of search because layers must be sequentially searched to preserve version correctness. NoveLSM addresses this problem by taking inspiration from the processor design to parallelize cache and TLB lookup. NoveLSM parallelizes each read operation by using a pool of dedicated worker threads to search across the memory hierarchy (DRAM and NVM memtables) and SSTables. To guarantee version correctness, NoveLSM always considers the value of a key returned by a thread accessing the highest level of LSM (e.g., key in memory table over SSTable) as the correct version. As shown in Figure 2 (b), this significantly reduces read latency.

References

[1] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," USENIX ATC 2018.

[2] L.Lu, T.S.Pillai, A.C.Arpaci-Dusseau,and R.H.Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-conscious Storage,", USENIX