# Rethinking DRAM Caching for LSMs in an NVRAM Environment

Lucas Lersch[1,2], Ismail Oukid[1,2], Ivan Schreter[2], and Wolfgang Lehner[1]

[1] TU Dresden, Germany wolfgang.lehner@tu-dresden.de
[2] SAP SE, Germany {lucas.lersch,i.oukid,ivan.schreter}@sap.com

**Abstract.** The rise of NVRAM technologies promises to change the way we think about system architectures. In order to fully exploit its advantages, it is required to develop systems specially tailored for NVRAM devices. Not only this imposes great challenges, but also developing full system architectures from scratch is undesirable in many scenarios due to prohibitive development costs. Instead, we analyze in this paper the behavior of an existing log-structured persistent key-value store, namely LevelDB, when run on top of an emulated NVRAM device. We investigate initial opportunities for improvement when adapting a system tailored for HDD/SSDs to run on top of an NVRAM environment. Furthermore, we analyze the behavior of the DRAM caching components of LevelDB and whether more suitable caching policies are required.

**Keywords:** log-structured merge-tree, persistent memory, caching, storage

## 1 Introduction

For some years already, NVRAM technologies have been announced as the next evolution step for persistent storage. These devices are expected to offer latencies much closer to those of DRAM, while providing higher storage capacity. Furthermore, even if accessible as a block device through the file system layer, these non-volatile memories are byte-addressable and can be directly accessed by the processor through its caches. While more convenient, the persistence aspect makes it non-trivial to develop systems that directly access NVRAM the same way as DRAM while leveraging its non-volatility; data consistency, persistent memory leaks, and partial writes are some of the challenges to be considered. The focus of this work is to analyze caching trade-offs involved in a hybrid DRAM-NVRAM environment. We take a log-structured merge-tree system (LSM) [14] as a case of study and investigate its behavior running on NVRAM. Not only LSMs are in the core of many modern systems, but their architecture handle reads and writes separately. This separation is particularly interesting, as it enables an analysis of the impact of caching in read and write operations in isolation.

In this work, we put LevelDB as an example of an LSM system on top of an emulated NVRAM device. LevelDB is designed to make efficient use of HDDs/SSDs. While a block device driver can be easily wrapped around NVRAM to make it look like an HDD/SSD, this does not exploit its full potential. Therefore, the first contribution of this work is the design of a persistent memory

environment for LevelDB, named *pmemenv*, to enable a more efficient management of persistent storage at a cache-line granularity. We consider that this approach offers a good compromise, as we are adapting an existing system instead of proposing a completely new architectural design or relaying all storage management to general purpose file systems originally designed for block devices.

While many file systems support Direct Access (DAX) [3,19] to bypass the page cache when accessing NVRAM, most database systems implement their own cache at the application level. Since DRAM will still have a lower latency than NVRAM, the second contribution of this work is to investigate if better performance can be achieved by using DRAM for caching in LevelDB.

The remaining of the paper is organized as follows: Section 2 presents the required background. Section 3 covers related works. Section 4 describes architectural details of LSMs and LevelDB. Section 5 introduces the implementation of *pmemenv*. Section 6 discusses caching policies better suited for NVRAM. Section 7 presents an experimental analysis. Finally, Section 8 concludes the paper.

## 2 Background

The Storage Networking Industry Association (SNIA) [18] defines that NVRAM devices should be managed by an NVRAM-aware file system which supports direct access. In order to acquire direct access via load/store semantics, the user application has to create a file and memory-map it to its virtual memory space, creating a *persistent memory pool*. The direct access provided by the file system guarantees that operations are done in the persistent memory pool without any sort of DRAM caching.

However, when dealing with load/store operations, one must consider any instruction re-ordering that might be introduced by the compiler or the processor. The non-volatility of NVRAM makes instruction re-ordering critical, as, in case of a system failure, it might result in problems such as loss of data consistency, partial writes, and persistent memory leaks. To avoid that, applications have to be carefully designed to enforce a proper durability order of store instructions. This can be currently achieved with the help of hardware instructions such as SFENCE, CLFLUSH, and *non-temporal stores*. SFENCE guarantees that all preceding store-to-memory instructions have been executed. CLFLUSH evicts a cache line and writes its content to memory. Non-temporal stores ensure that data is written directly to memory, bypassing the processor cache. Intel has also announced the CLFLUSHOPT and CLWB instructions to enhance performance over CLFLUSH: CLFLUSHOPT is ordered with a smaller set of memory traffic, allowing multiple cache lines to be flushed in parallel within a single logical processor's instruction stream, and CLWB writes back a cache line to memory, without invalidating it, making future reads and writes much faster.

Furthermore, a persistent memory pool might be memory mapped to a different virtual memory space at restart time. Therefore, an application accessing NVRAM through this interface must support persistent pointers [15] to keep track of its persistent allocated memory regions. Finally, in order to avoid partial writes, the application should consider that, while the unit of transfer between

the processor and NVRAM is a cache line, durable atomicity of writes is only guaranteed at a smaller granularity (8 Bytes on Intel x86 architectures).

In order to guarantee a certain level of consistency, database systems require careful control of when data is written to persistent storage. In an NVRAM scenario, even if this level of control can be achieved with the aid of the aforementioned hardware instructions, these introduce additional overhead and complexity. In this context, even if NVRAM provides fast random access, log-structured systems are still interesting for the following reasons: First, in contrast to update-in-place systems, maintaining consistency and atomicity of a log-structured write is simpler, as only the tail of the log is updated to reflect the operation, thus avoiding partial writes. Second, it is possible to employ non-temporal stores to bypass the CPU cache and write directly to NVRAM, avoiding operations such as CLFLUSH. Third, persistent log-structured key-value stores, such as LevelDB and RocksDB, aim at reducing the amount of data written to persistent storage (write amplification). This is even more appealing considering that NVRAM supports a limited number of writes.

We take the LSM as an example of a log-structured system to analyze its behavior on NVRAM. The LSM was initially proposed to improve write performance of update-in-place data structures (e.g., B+Tree) in HDDs. The write-optimized nature of LSMs makes them appealing to cloud systems that experience high write and data injection rates. Systems that must ingest an event log and query the ingested data with acceptable response time are common examples. The popularity of LSMs increased following the trend of Google's Bigtable [9]. Other systems that implement a similar LSM-based architecture at the storage layer are: HBase [2], Cassandra [1], Riak [6], LevelDB [4], and RocksDB [7]. We use LevelDB since it is a smaller, simpler, and easier-to-modify system.

## 3 Related Work

A significant amount of research has been conducted in the past years on NVRAM technologies. It is still unclear which role such devices will to play in the storage hierarchy [8]. Nevertheless, an assumption considered by the vast majority of research is that DRAM will keep co-existing with NVRAM in hybrid environments. In the following, we highlight approaches closely related to this paper.

Pelley et al. [16] analyze the behavior of Shore-MT, an update-in-place storage manager developed for HDDs, when running on an NVRAM device. The authors investigate the buffering components and the overhead of re-ordering writes in different recovery algorithms. Dulloor et al. [10] presented PMFS, a lightweight file system to manage files at the user-space level, avoiding going through the block layer and the page cache of the operating system. Also, the ext4 file system was extended with Direct Access (DAX) capabilities to bypass the operating system cache to better support NVRAM devices. Xu et al. [20] present a log-structured file system adapted to make efficient use of hybrid environments and exploit the fast random access of NVRAM.

More recently, Li et al. [13] investigated adapting RocksDB to NVRAM. They aim at improving the recovery time of RocksDB by replacing the votatile

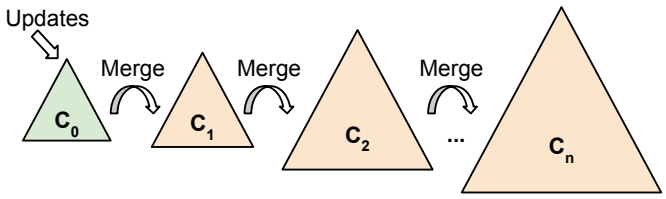


Fig. 1: General architecture of a log-structured merge-tree (LSM).

MemTable by a persistent one, thus avoiding any logging for durability. Furthermore, they also analyze the caching behavior when having NVRAM in between DRAM and SSDs in the storage hierarchy. RocksDB originated as a fork of LevelDB and provides currently many improvements over LevelDB. Nevertheless, while RocksDB shows better numbers performance-wise, both systems implement a log-structured merge-tree (LSM) at their core.

## 4 LSM & LevelDB Architecture

Figure 1 illustrates the general architecture of an LSM. Updates in an LSM are made in a separate in-memory data structure ($C_0$) which is made durable by logging. When $C_0$ reaches a certain threshold size, it is migrated to a lower level persistent component $C_1$, which can be, for instance, a tree structure. This can be generalized to a hierarchy of multiple persistent components $C_1..C_n$, in such a way that the size of components grows exponentially in relation to the preceding component in the hierarchy. All components store records in a sorted order.

A lookup operation has to consider the multiple components. A rolling-merge process between components usually runs in the background. This is required to reclaim space and keep a predictable performance by alleviating the read penalty introduced by multiple components. Indeed, components are sorted by recency and a lookup has to inspect from the most recent to the oldest component until the key is found. The improved write performance is achieved by two different characteristics: converting random writes into sequential ones and reducing the amount of data written to persistent storage (i.e., decreasing write-amplification).

### 4.1 LevelDB

LevelDB is an open-source, embeddable, persistent key-value store originally developed by Google. It treats keys and values as arbitrary byte arrays and stores data sorted by key. The user can interact with the system through a basic interface including *Put(key,value)*, *Get(key)*, and *Delete(key)*. In the following, we detail some relevant implementation details required to understand the remaining of this paper. The architecture of LevelDB is summarized in Figure 3. LevelDB uses a skip-list [17] as its in-memory data structure, called *MemTable*. The persistent components are organized as levels $L_0$ to $L_n$. Each level is composed of a certain number of *Sorted String Tables* (SST).

An SST has a fixed size (around 2 MB) and consists of four types of blocks. A *data block*, usually 4 KB, contains keys and values in sorted order (possibly compressed). Additionally, an SST has a single *index block* used to locate the

data block of a given key. Optionally, there can be *meta blocks* that store information such as bloom filters, in which case there will also be a *meta index block* to locate them. The layout of an SST is represented in Figure 2. Whenever the MemTable reaches a threshold size (4 MB by default), it is compressed and converted into an SST of $L_0$. With the exception of $L_0$, SSTs of the same level do not have overlapping key ranges, meaning that at most one SST per level has to be read. When the number of SSTs in $L_n$ reaches its threshold, they are merged with the SSTs of $L_{n+1}$ that have overlapping key ranges to generate new SSTs. LevelDB has 7 levels by default, with $L_0$ containing a maximum of 4 SSTs, $L_1$ 10 SSTs, and each level after contains a maximum of factor 10 the amount of SSTs in the level above. Finally, a system catalog keeps track of information on all levels, e.g., current SSTs and their key ranges.

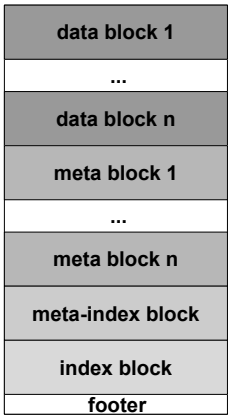



Fig. 2: Sorted String Table layout.

**Caching** By default, LevelDB uses memory-mapped files and the page cache of the operating system for improved performance. These features are orthogonal and we have disabled them in our experiments to isolate the behavior of LevelDB's own caching component.

In addition to the MemTable, LevelDB implements two DRAM read-only caches: *table cache* and *block cache*. The table cache is used to hold entries containing metadata about SSTs and index blocks (possibly meta blocks) of SSTs recently accessed. The block cache holds exclusively the data blocks from SSTs. In order to improve concurrency, both caches are composed of 32 shards (default), and each shard implements least recently used (LRU) as the default block replacement policy. A read operation must first check if the given key is present in the MemTable. If not, then it will locate the candidate SST in the next level based on the SST key ranges contained in the catalog. Once the relevant SST is found, the table cache and block cache are searched for the corresponding index block and data block, respectively.

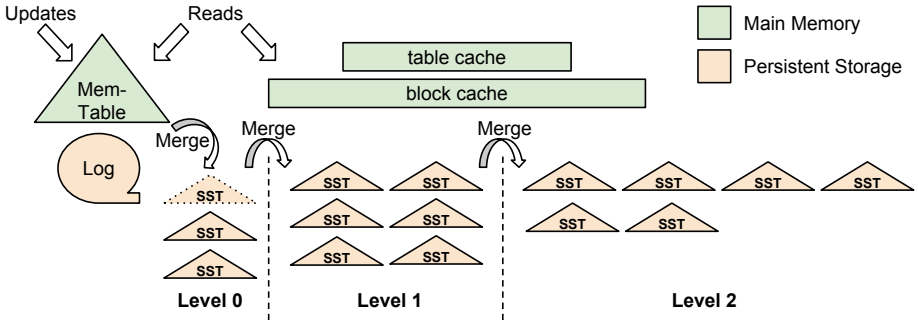



Fig. 3: LevelDB architecture represented with the first three levels for simplicity.

## 5 *Pmemenv*: Persistent Memory Environment

We have implemented a lightweight persistent memory environment, denoted *pmemenv*, using Intel's NVML library [5]. *Pmemenv* is tailored specifically for accessing and managing LevelDB's components in NVRAM directly in the user-space. This contrasts with the usual interface, where the application has to go through the file system layer to access the persistent storage. *Pmemenv* has two main advantages: First, it enables zero-copy reads, meaning that data can be read directly from NVRAM without loading it to DRAM. Second, it enables read and write operations to NVRAM at a cache-line granularity.

LevelDB manages SSTs through a virtual interface, enabling users to customize the required behavior. NVML enables the user to create a persistent memory pool and manage objects in it through a persistent allocator interface, hiding from the user the complexity required to properly enforce the order of write operations. The library uses a lightweight logging scheme to guarantee the fail-safe atomicity of these persistent allocations. A persistent pointer for each allocated block is stored in a collection located in a fixed memory region of the pool. The user is able to iterate over this collection and retrieve every allocated object in the persistent memory pool, thus preventing persistent memory leaks. As a side note, NVML also offers transactional support to enable more complex atomic memory operations than allocation/deallocation of blocks of memory. These memory transactions, however, are not explicitly used in this work.

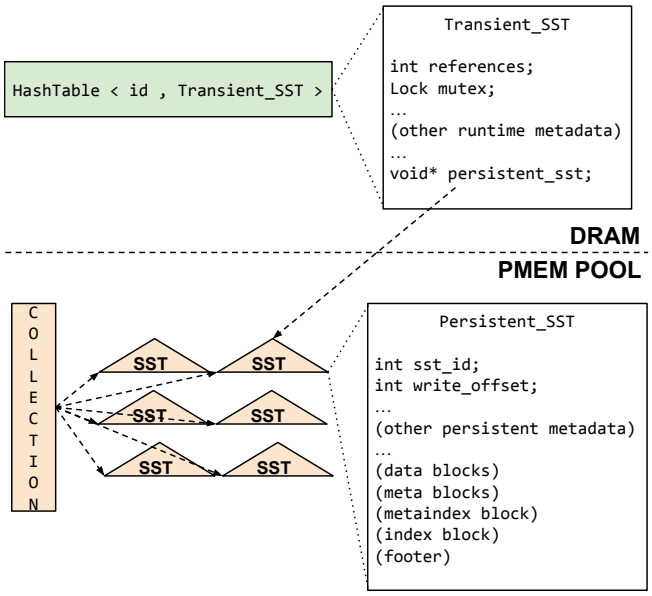



Fig. 4: *pmemenv* architecture.

Through NVML, *pmemenv* implements the following atomic operations on SSTs required by LevelDB: create/allocate, write bytes (append-only), read bytes, delete/deallocate, rename. Figure 4 illustrates the general architecture

of *pmemenv*. It comprises two main parts: an in-memory hash table and a persistent memory pool. Every SST is composed of a transient part and a persistent part. The hash table is used to map the unique identifier of an SST to its transient part. The transient part of an SST includes non-critical metadata that is only required during normal execution, such as reference counters, mutexes, and status flags. The transient part also contains a pointer indicating the location of the persistent part in the persistent memory pool. The persistent part contains critical data required by basic operations and to recover the hash table after a system failure. In our implementation, the persistent part of an SST is composed of the unique identifier, the current append offset, and the remaining data and index blocks shown in Figure 2. The NVML library already stores the size of allocated persistent memory blocks, therefore, even if this is critical data, we do not store it and rely on the library to provide this information. In case of a system failure, the hash table can be rebuilt by iterating over the collection of pointers pointing to the allocated SSTs and retrieving the SST identifiers. The metadata in the transient SST part is set to default values and the pointer is set to the corresponding address in the persistent memory pool.

The separation of SSTs into transient and persistent parts allows metadata to be moved between them, enabling the system to possibly slide a *persistence bar* to choose which parts to make persistent [15]. In one extreme scenario, all data, metadata, and data structures are allocated in the persistent memory pool. This would reduce the recovery time to a minimum at a possible performance cost and additional complexity. Nevertheless, this is out of the scope of this paper.

Finally, since all writes to SSTs are log-structured (i.e., append-only), they can use non-temporal stores to bypass the processor cache. Writes of an arbitrary number of bytes to an SST are protected from partial writes by updating the current write offset of the SST (8 Bytes), which is guaranteed to be an atomic operation. The MemTable log and other auxiliary files, such as the catalog of SSTs, are also managed by *pmemenv* the same way as SSTs.

## 6 2Q Cache Policy for NVRAM

LevelDB implements the LRU as its default replacement policy, meaning that whenever the cache is full and a miss occurs, the least recently accessed block is evicted to make space for the requested one. However, when the SSTs are in NVRAM, the processor is able to directly read these blocks without bringing them to DRAM. On one hand, DRAM has a lower latency and enables faster access to data blocks. On the other hand, not only a cache policy introduces additional complexity, but there is also an overhead for transferring data from NVRAM to DRAM when a miss occurs. This overhead might not be worthwhile when compared to the cost of simply accessing the data directly from NVRAM.

Ideally, we would like the cache replacement policy to keep track of accesses not only to cached blocks, but also to un-cached ones. This would enable the policy to make a better decision whether it is advantageous to transfer a given block to DRAM, avoiding a hotter block to be evicted following a miss to a colder one. As an example, this behavior would avoid that a table scan trashes the cache by evicting all of its contents.

The 2Q replacement policy [12] considers similar goals. While the original 2Q was proposed in the context of main memory and hard disks, we have adapted the concept to NVRAM by enabling zero-copy reads from the persistent storage. Similar to the original, our 2Q policy has two components: AM and A1. AM holds cached blocks and is managed by some replacement policy (LRU in our case). A1 does not hold any blocks, but only keeps track of accesses to un-cached blocks (blocks read directly from NVRAM). Since only references to blocks are kept, the space consumption of A1 is minimal. The size of A1 is tunable and references are kept in a FIFO queue.

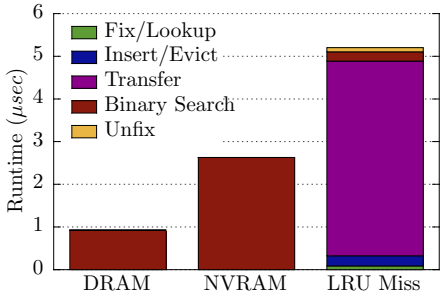



Fig. 5: Average runtime of binary search over 4 KB of integers.

Algorithm 1 shows the pseudo-code for the two main functions of the replacement policy. The function *FIX* is called to request access to a block. If the block is already cached, it is directly returned (line 3). Otherwise, if it is not going to cause another block to be evicted or if there was another reference to it in the recent past (line 5), the block is transferred to DRAM (line 6). If the block was accessed recently, it means that it is probably hot and is a good candidate to be transferred to DRAM. If both conditions are false, the block is read directly from NVRAM and a reference to it is added to A1 (line 9-10). A hash table is used for efficient containment test of both AM and A1. The *VICTIM* function is called when the cache is full and we need to pick a block for eviction. The block picked is the least recently used one (line 16), but a reference to it is additionally added to A1 (line 17). Since A1 is a FIFO with a limited size, it discards its oldest reference when a new one is inserted.

Figure 5 shows the runtime of a binary search over 4 KB of integers in DRAM, NVRAM (latency set to 4x that of DRAM), and simulating a miss of LRU in a DRAM cache. In the breakdown of the cost of a LRU miss it is possible to see the constant overhead introduced by the cache component (fix, unfix, eviction, etc), as well as the huge cost of transferring data from NVRAM to DRAM. The binary search represented in the breakdown is faster than the one in DRAM, because the data was already cached by the CPU caches during the transfer. The cache miss in the LRU policy has a constant cost comprised of lookup, eviction, and move of a block to DRAM. The additional cost required by the policy exceeds by far the cost of simply doing the binary search in NVRAM. The 2Q policy introduces two different scenarios for a cache miss: the first scenario simply adds a reference to the FIFO and reads directly from NVRAM, while the second scenario is similar to LRU. In an NVRAM context, a well-tuned 2Q policy should prefer to pay the lower miss cost for data not frequently accessed and the higher miss cost for data expected to be frequently accessed in the near future. While most proposed replacement strategies (LRU, LFU, CLOCK, etc) focus on

**Algorithm 1** 2Q policy pseudo-code

```
1: function FIX(blk_id)
2:     if AM.contains(blk_id) then
3:         return AM.get(blk_id)
4:     else
5:         if !AM.full() OR A1.contains(blk_id) then
6:             AM.load(blk_id)
7:             return AM.get(blk_id)
8:         else
9:             A1.add(blk_id)
10:            return NVRAM.get(blk_id)
11:        end if
12:    end if
13: end function
14:
15: function VICTIM( )
16:    blk_id ← AM.remove_lru()
17:    A1.add(blk_id)
18: end function
```

improving the hit ratio, the idea of being able to choose between two different miss costs adds a new dimension. Assuming that the hit ratio is determined by the replacement strategy and the cache size, a smaller 2Q cache is likely to have more misses than a larger LRU cache. However, if most of the 2Q misses pay the lower cost, similar or even better performance than LRU can be achieved with a lower memory consumption. This introduces the non-intuitive idea that a higher hit ratio does not necessarily translate to better performance, as the costs for misses might differ.

## 7 Evaluation

We use the Intel NVM Emulation platform that emulates an NVRAM device by accessing a dedicated area of DRAM with higher, tunable latency. The higher access latency to DRAM is achieved thanks to a special BIOS. A full description of this system can be found in [11]. The system is equipped with two Intel Xeon E5 processors. Each one has 8 cores, running at 2.6GHz, and featuring 32 KB L1 data and 32 KB L1 instruction cache as well as 256 KB L2 cache. The 8 cores of one processor share a 20 MB last-level cache. The system has 64 GB of DRAM and 192 GB of emulated NVRAM. The emulated NVRAM device is mounted with the ext4 file system with DAX support. In the experiments, we set the latency of NVRAM to 360 ns, approximately 4x the latency of DRAM (90 ns). Considering HDD/SSD is out of the scope of this work, since all experiments are based on the assumption that the storage device can be directly accessed through the CPU caches. The system runs Linux with kernel version 4.4.21.

We use NVML Release Version 1.2 and LevelDB Release Version 1.19. All the source code was compiled using GCC 4.8.5. We disabled the memory mappings of SSTs and operating system caching in LevelDB. Compression and filtering of

SSTs (bloom filters) are also not used. The MemTable is set to its default size of 4MB. All requests to LevelDB are made from a single thread.

### 7.1 Write Performance

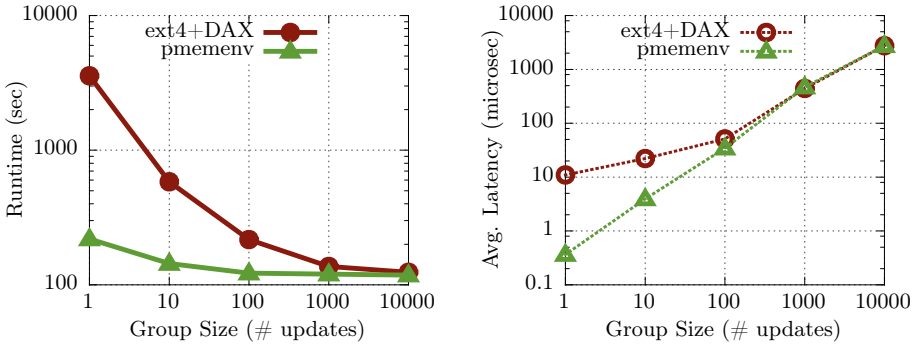
We first analyze runtime and latency of two approaches for writing to NVRAM: through the ext4 file system with Direct Access (DAX) support (as a drop-in replacement for HDDs/SSDs) and through *pmemenv*. As mentioned in Section 5, the file system manages these operations at a block granularity (usually 4 KB), while *pmemenv* allows finer control over written data. This implies that, in a scenario where durability of single operations must be guaranteed, *pmemenv* is able to write only the changed cache-lines. However, most systems implement some sort of group commit to hide write latencies. LevelDB enables this by batching many *Put* operations in a *WriteBatch* that is accumulated in DRAM and is later made durable as a single *Put*. We consider scenarios with different *WriteBach* sizes. Additionally, we investigate if batching writes in DRAM still offers benefits for *pmemenv*. We run a write-only workload of YCSB with 100M key-value records, where each key is 16 bytes and each value is 112 Bytes, giving a total of 128 Bytes per record. The results are depicted in Figure 6.

First, for a group size of one, the ext4+DAX configuration has to persist data at the granularity of pages via *fsync*, which incurs a high cost if single operations are to be made durable. *Pmemenv* is able to avoid the kernel path and to persist data at a much smaller granularity, which reduces the overhead related to write operations. Increasing the group size drastically improves the performance of ext4+DAX (16 times faster when increasing group size from 1 to 100), as the cost of *fsync* is amortized across many insertions. For *pmemenv*, an improvement of approximately 50% in runtime is observed when increasing the group size from 1 to 10. For larger group sizes the difference is not significant.

Grouping insert operations in batches introduces a trade-off between throughput (runtime) and latency, as the first requests to arrive in a group are delayed. Figure 6b shows the average latency of single insert requests for different group sizes. The standard deviation can be observed in Table 1. For smaller group sizes (1 to 100) in ext4+DAX, the increased latency is justified by the large gains in runtime, making the batching of operations an obvious choice for most applications. However, for *pmemenv*, this trade-off is not so clear and the decision of sacrificing latency for better runtime might become a matter of Service Level Agreements, like response time required by applications. Finally, not only *pmemenv* presents lower runtime and lower average latency than ext4+DAX, but also a lower standard deviation for group sizes 1 to 100, which translates to a more predictable performance over time.

| **File System** | **1** | **10** | **100** | **1000** | **10000** |
|---|---|---|---|---|---|
| ext4 + DAX | 23 | 56 | 161 | 586 | 4750 |
| pmemenv | 12 | 39 | 125 | 517 | 4749 |

Table 1: Latency standard deviation of Figure 6b in microseconds.

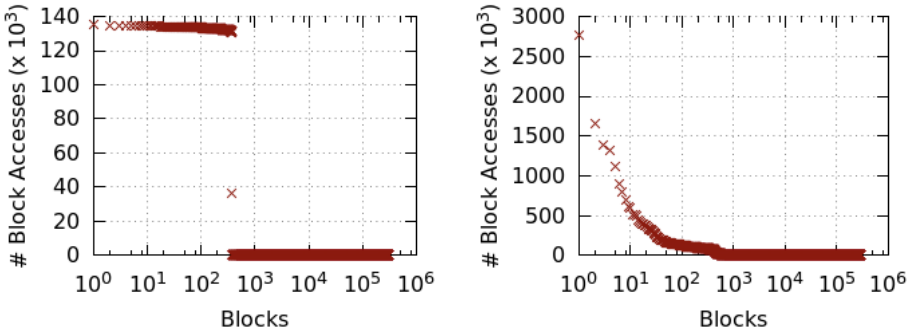



(a) Runtime (b) Average Latency

Fig. 6: Insertion of 100 million key-value records with varying *WriteBatch* size.

### 7.2 Read Performance

We also analyze if better performance can be achieved by dedicating a portion of DRAM for caching hot data. Since writes in LevelDB are made in a separate data structure, the remaining caching components benefit mainly read operations. Therefore, we have considered read-only YCSB workloads to better outline the performance impact. Each workload issues 50 million lookups over 10 million key-value pairs. Before each workload is executed, the caches are warmed up by executing read requests until they become completely full. The warmup time is not considered. We analyze two different scenarios: uniform and skewed distribution (80% of requests to 20% of the records) of key requests. It is worth noting that each *Get* request for a key translates into two block requests, one for the index block and one for the data block. Hence, even a uniform distribution of keys presents a skewed distribution of block accesses. Figure 7 illustrates the number of accesses of each block sorted from the most to the last accessed.

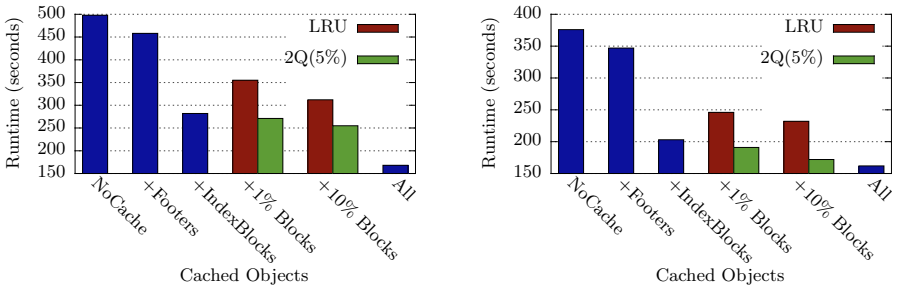



(a) Uniform (b) Skewed

Fig. 7: Distribution of accesses to blocks.

Figure 8 presents the runtime of both read-only workloads for different system configurations. The X axis represents which portions of SSTs are cached in DRAM. We start with a *NoCache* approach, where the caching components were completely removed and all SSTs are read directly from NVRAM through *pmemenv*. Later, we gradually increase the DRAM consumption by statically

placing portions of every SSTs in DRAM. The *Footers* scenario has the footers of all SSTs in DRAM. The footer of an SST contains pointers to the index blocks, as well as checksums and additional status flags. Footers are frequently accessed (it is where each read in an SST starts) and, since they are relatively small (around 64 Bytes), keeping all of them in DRAM improves performance at a minimal cost of memory consumption. Next, *IndexBlocks* considers holding the index blocks of all SSTs in DRAM. For our workloads, every index block is approximately 18 KB and there are around 500 SSTs, giving a total of about 10 MB additional DRAM consumption (less than 1% of the total size). The observed performance gains are significant and justify the additional memory consumption. At this point, we can conclude that a careful placement of frequently accessed data in DRAM is beneficial despite the low latency of NVRAM.




(a) Uniform (b) Skewed

Fig. 8: Runtime of read-only workload.

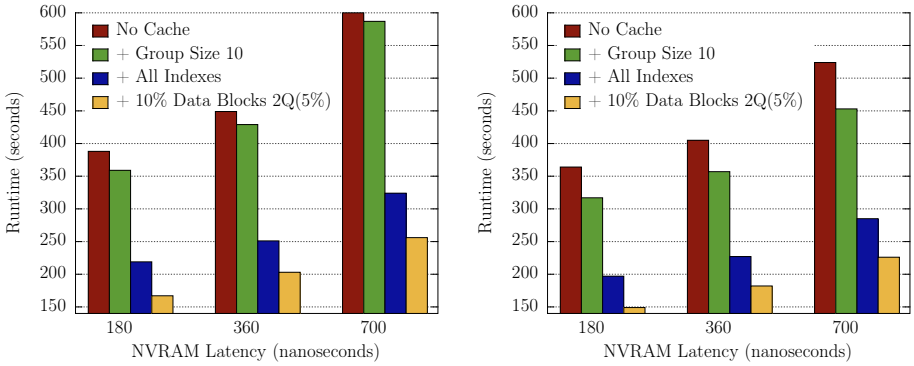
However, so far we have only statically placed data in DRAM or NVRAM, there is no caching component involved. In addition to keeping all index blocks in DRAM we introduce a caching component for the data blocks, which enables the system to dynamically adapt by keeping frequently accessed data blocks in DRAM. The scenarios *1% Blocks* and *10% Blocks* cache the indicated amount of data blocks. The interesting observation for LRU (default cache policy in LevelDB) is that dedicating additional DRAM harms the system's performance initially. While the performance improves with more DRAM (*10% Blocks*), larger amounts of DRAM would be required to achieve the same performance of caching only index blocks. This is explained by the cost of cache misses in LRU: lookup, eviction, and transfer of block from NVRAM to DRAM. If cached data is not accessed enough times, this cost is high compared to the alternative of directly accessing data in NVRAM and avoiding the overhead caching. As discussed in Section 6, the cost of transferring data to DRAM is only worthwhile if the policy can predict that this block will be accessed frequently in the near future.

To alleviate the high cost of misses, we have implemented 2Q to enable a more lightweight policy. We have set the A1 size to 5% of the AM size. Our initial goal with 2Q is to avoid the observed behavior where the system gets slower when more DRAM is dedicated for caching. In contrast to LRU, there is always some performance improvement with larger caches for data blocks. This comes from the fact that 2Q avoids evicting a cached block and moving a new block to DRAM when a miss occurs. Finally, the *All* scenario represents

the runtime with the whole dataset cached in DRAM. It is possible to see in Figure 8b that the 2Q cache with enough DRAM to hold 10% of the data blocks can achieve similar performance of holding all blocks in DRAM.

### 7.3 Mixed Workloads

Based on the observations from the previous experiments, we analyze the overall behavior of the system in workloads containing both updates and lookups. Two mixed workloads with skewed access are considered: 25% and 50% of updates. We also run the experiments with varying NVRAM latencies to show that the behavior is the same regardless of the slowdown/speedup incurred by higher/lower latencies.




(a) 25% Updates (b) 50% Updates

Fig. 9: Runtime of skewed mixed workload.

Similar to previous results, we start with a *NoCache* approach and gradually dedicate more DRAM for caching purposes. The *Group Size 10* has enough DRAM for holding 10 update operations and persist them as a single *WriteBatch*. Later, in addition to that, we reserve enough DRAM to hold *All Indexes*. Finally, we hold up to 10% of the data blocks in a 2Q cache. Figure 9 presents the gradual performance gains achieved in each of these steps. The biggest improvement happens when keeping all index blocks in DRAM. Not only index blocks are frequently accessed, but their additional DRAM consumption is minimal, making it realistic to hold all of them in DRAM and avoiding any replacement policy overhead. While a cache for data blocks with 2Q replacement policy offers some benefits in terms of performance, it is up to the user to decide if the cost of additional DRAM justify these gains. Nevertheless, we consider that enabling the system to manage hot and cold data is important and better caching policies can probably achieve this behavior with even better performance.

## 8 Conclusion

We have adapted LevelDB to run on top of an emulated NVRAM device. We considered a hybrid DRAM-NVRAM environment and discussed relevant implementation details. We implemented *pmemenv*, a data accessor to enable fine-grained management of SSTs in NVRAM. Furthermore, we analyzed the LevelDB caching components for improving write and read performance. Even with

its lower latency, simply dedicating a portion of DRAM to cache data from NVRAM is not necessarily beneficial. On one hand, we have observed that poor caching policies might even harm performance. On the other hand, a careful placement of data offers significant benefits. A dynamic management of hot and cold data can be achieved through lightweight caching policies In this context, we have shown that 2Q never harms the system performance and enables the system to make better decisions about caching. Future work can probably achieve even better results with other lightweight and well-tuned policies.

## References


1. Apache Cassandra. http://cassandra.apache.org/, accessed: 2017-02-17
2. Apache HBase. https://hbase.apache.org/, accessed: 2017-02-17
3. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt, accessed: 2017-02-17
4. LevelDB. http://leveldb.org/, accessed: 2017-02-17
5. NVML. http://pmem.io/nvml/libpmem/, accessed: 2017-02-17
6. Riak. http://basho.com/products/riak-kv/, accessed: 2017-02-17
7. RocksDB. http://rocksdb.org/, accessed: 2017-02-17
8. Bonnet, P.: What's Up with the Storage Hierarchy? CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research (Online Proceedings) (2017)
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (2006)
10. Dulloor, S., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System Software For Persistent Memory. Eurosys Conference (2014)
11. Dulloor, S.R.: Systems and Applications for Persistent Memory. https://smartech.gatech.edu/bitstream/handle/1853/54396/DULLOOR-DISSERTATION-2015.pdf (2015), phD Thesis
12. Johnson, T., Shasha, D.E.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. PVLDB (1994)
13. Li, J., Pavlo, A., Dong, S.: NVMRocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems, accessed: 2017-02-17
14. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. (1996)
15. Oukid, I., Booss, D., Lehner, W., Bumbulis, P., Willhalm, T.: SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. International Workshop on Data Management on New Hardware (2014)
16. Pelley, S., Wenisch, T.F., Gold, B.T., Bridge, B.: Storage Management in the NVRAM Era. PVLDB (2013)
17. Pugh, W.: Concurrent maintenance of skip lists. Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-90-80 (1990)
18. SNIA: NVM Programming Model V1.1. http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.1.pdf (2015)
19. Wilcox, M.: Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/, accessed: 2017-02-17
20. Xu, J., Swanson, S.: NOVA A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST) (2016)