Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems

Joy Arulraj
jarulraj@cs.cmu.edu
Carnegie Mellon University

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University

Subramanya R. Dulloor
subramanya.r.dulloor@intel.com
Intel Labs

ABSTRACT

The advent of non-volatile memory (NVM) will fundamentally change the dichotomy between memory and durable storage in database management systems (DBMSs). These new NVM devices are almost as fast as DRAM, but all writes to it are potentially persistent even after power loss. Existing DBMSs are unable to take full advantage of this technology because their internal architectures are predicated on the assumption that memory is volatile. With NVM, many of the components of legacy DBMSs are unnecessary and will degrade the performance of data intensive applications.

To better understand these issues, we implemented three engines in a modular DBMS testbed that are based on different storage management architectures: (1) in-place updates, (2) copy-on-write updates, and (3) log-structured updates. We then present NVM-aware variants of these architectures that leverage the persistence and byte-addressability properties of NVM in their storage and recovery methods. Our experimental evaluation on an NVM hardware emulator shows that these engines achieve up to 5.5× higher throughput than their traditional counterparts while reducing the amount of wear due to write operations by up to 2×. We also demonstrate that our NVM-aware recovery protocols allow these engines to recover almost instantaneously after the DBMS restarts.

1. INTRODUCTION

Changes in computer trends have given rise to new on-line transaction processing (OLTP) applications that support a large number of concurrent users and systems. What makes these modern applications unlike their predecessors is the scale in which they ingest information [41]. Database management systems (DBMSs) are the critical component of these applications because they are responsible for ensuring transactions' operations execute in the correct order and that their changes are not lost after a crash. Optimizing the DBMS's performance is important because it determines how quickly an application can take in new information and how quickly it can use it to make new decisions. This performance is affected by how fast the system can read and write data from storage.

DBMSs have always dealt with the trade-off between volatile and non-volatile storage devices. In order to retain data after a loss of power, the DBMS must write that data to a non-volatile device, such as a SSD or HDD. Such devices only support slow, bulk data transfers as blocks. Contrast this with volatile DRAM, where a DBMS can quickly read and write a single byte from these devices, but all data is lost once power is lost.

In addition, there are inherent physical limitations that prevent DRAM from scaling to capacities beyond today's levels [46]. Using a large amount of DRAM also consumes a lot of energy since it requires periodic refreshing to preserve data even if it is not actively used. Studies have shown that DRAM consumes about 40% of the overall power consumed by a server [42].

Although flash-based SSDs have better storage capacities and use less energy than DRAM, they have other issues that make them less than ideal. For example, they are much slower than DRAM and only support unwieldy block-based access methods. This means that if a transaction updates a single byte of data stored on an SSD, then the DBMS must write the change out as a block (typically 4 KB). This is problematic for OLTP applications that make many small changes to the database because these devices only support a limited number of writes per address [66]. Shrinking SSDs to smaller sizes also degrades their reliability and increases interference effects. Stop-gap solutions, such as battery-backed DRAM caches, help mitigate the performance difference but do not resolve these other problems [11].

Non-volatile memory (NVM)[1] offers an intriguing blend of the two storage mediums. NVM is a broad class of technologies, including phase-change memory [55], memristors [60], and STT-MRAM [26] that provide low latency reads and writes on the same order of magnitude as DRAM, but with persistent writes and large storage capacity like a SSD [13]. Table 1 compares the characteristics of NVM with other storage technologies.

It is unclear at this point, however, how to best leverage these new technologies in a DBMS. There are several aspects of NVM that make existing DBMS architectures inappropriate for them [14, 23]. For example, disk-oriented DBMSs (e.g., Oracle RDBMS, IBM DB2, MySQL) are predicated on using block-oriented devices for durable storage that are slow at random access. As such, they maintain an in-memory cache for blocks of tuples and try to maximize the amount of sequential reads and writes to storage. In the case of memory-oriented DBMSs (e.g., VoltDB, MemSQL), they contain certain components to overcome the volatility of DRAM. Such components may be unnecessary in a system with byte-addressable NVM with fast random access.

In this paper, we evaluate different storage and recovery methods for OLTP DBMSs from the ground-up, starting with an NVM-only storage hierarchy. We implemented three storage engine architectures in a single DBMS: (1) in-place updates with logging, (2) copy-on-write updates without logging, and (3) log-structured updates.

[1] NVM is also referred to as *storage-class memory* or *persistent memory*.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749441 .

	DRAM	PCM	RRAM	MRAM	SSD	HDD
Read latency	60 ns	50 ns	100 ns	20 ns	25 μs	10 ms
Write latency	60 ns	150 ns	100 ns	20 ns	300 μs	10 ms
Addressability	Byte	Byte	Byte	Byte	Block	Block
Volatile	Yes	No	No	No	No	No
Energy/ bit access	2 pJ	2 pJ	100 pJ	0.02 pJ	10 nJ	0.1 J
Endurance	$>10^{16}$	10^{10}	10^{8}	10^{15}	10^{5}	$>10^{16}$

Table 1: Comparison of emerging NVM technologies with other storage technologies [15, 27, 54, 49]: phase-change memory (PCM) [55], memristors (RRAM) [60], and STT-MRAM (MRAM) [26].

We then developed optimized variants for these approaches that reduce the computational overhead, storage footprint, and wear-out of NVM devices. For our evaluation, we use a hardware-based emulator where the system only has NVM and volatile CPU-level caches (i.e., no DRAM). Our analysis shows that the NVM-optimized storage engines improve the DBMS's throughput by a factor of 5.5× while reducing the number of writes to NVM in half. These results also suggest that NVM-optimized in-place updates is the ideal method as it has lowest overhead, causes minimal wear on the device, and allows the DBMS to restart almost instantaneously.

Our work differs from previous studies because we evaluate a DBMS using a single-tier storage hierarchy. Others have only employed NVM for logging [28, 30, 65] or used a two-level hierarchy with DRAM and NVM [53]. We note that it is possible today to replace DRAM with NV-DIMM [6], and run an NVM-only DBMS unmodified on this storage hierarchy. This enables us to achieve performance similar to that obtained on a DRAM and NVM storage hierarchy, while avoiding the overhead of dealing with the volatility of DRAM. Further, some NVM technologies, such as STT-RAM [26], are expected to deliver lower read and write latencies than DRAM. NVM-only DBMSs would be a good fit for these technologies.

The remainder of this paper is organized as follows. We begin in Section 2 with a discussion of NVM and why it necessitates a re-evaluation of DBMS storage architectures. Next, in Section 3 we describe our DBMS testbed and its storage engines that we developed for this study. We then present in Section 4 our optimizations for these engines that leverage NVM's unique properties. We then present our experimental evaluation in Section 5. We conclude with a discussion of related work in Section 6.

2. BACKGROUND

We now provide an overview of emerging NVM technologies and discuss the hardware emulator platform that we use in this paper.

2.1 Motivation

There are essentially two types of DBMS architectures: disk-oriented and memory-oriented systems [23]. The former is exemplified by the first DBMSs, such as IBM's System R [7], where the system is predicated on the management of blocks of tuples on disk using an in-memory cache; the latter by IBM's IMS/VS Fast Path [32], where the system performs updates on in-memory data and relies on the disk to ensure durability. The need to ensure that all changes are durable has dominated the design of systems with both types of architectures. This has involved optimizing the layout of data for each storage layer depending on how fast it can perform random accesses [31]. Further, updates performed on tuples stored in memory need to be propagated to an on-disk representation for durability. Previous studies have shown that the overhead of managing this data movement for OLTP workloads is considerable [35].

NVM technologies, like phase-change memory [55] and memristors [60], remove these tuple transformation and propagation costs through byte-addressable loads and stores with low latency. This means that they can be used for efficient architectures that are used in memory-oriented DBMSs [25]. But unlike DRAM, all writes to the NVM are potentially durable and therefore a DBMS can access the tuples directly in the NVM after a restart or crash without needing to reload the database first.

Although the advantages of NVM are obvious, making full use of them in an OLTP DBMS is non-trivial. Previous work that compared disk-oriented and memory-oriented DBMSs for NVM showed that the two architectures achieve almost the same performance when using NVM because of the overhead of logging [23]. This is because current DBMSs assume that memory is volatile, and thus their architectures are predicated on making redundant copies of changes on durable storage. Thus, we seek to understand the characteristics of different storage and recovery methods from the ground-up.

2.2 NVM Hardware Emulator

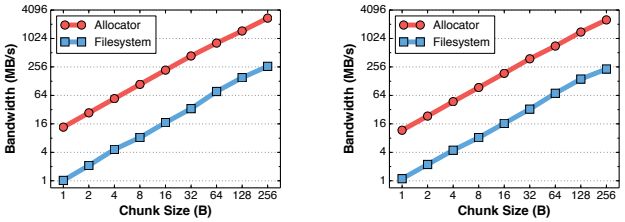
NVM storage devices are currently prohibitively expensive and only support small capacities. For this reason, we use a NVM hardware emulator developed by Intel Labs [27] in this paper. The emulator supports tunable read latencies and read/write bandwidths. This enables us to evaluate multiple hardware profiles that are not specific to a particular NVM technology. Unlike NVM simulators, like PCM-SIM [44], this emulator enables us to better understand the impact of cache evictions, prefetching, and speculative execution on long-running workloads. The hardware emulator is based on a dual-socket Intel Xeon platform. Each CPU supports four DDR3 channels with two DIMMs per channel. Half of the memory channels on each processor are reserved for the emulated NVM while the rest are used for regular memory. The emulator's custom BIOS firmware partitions the physical memory address space into separate address spaces for DRAM and emulated NVM.

NVM technologies have higher read and write latency than DRAM. We are able to emulate the latency for the NVM partition by using custom CPU microcode. The microcode estimates the additional cycles that the CPU would have to wait if DRAM is replaced by slower NVM and then stalls the CPU for those cycles. The accuracy of the latency emulation model was validated by comparing the performance of several applications on emulated NVM and slower NUMA memory [27]. Since the CPU uses a write-back cache for NVM, the high latency of writes to NVM is not observed on every write but the sustainable write bandwidth of NVM is lower compared to DRAM [38]. The emulator allows us to control the write bandwidth by limiting the number of DDR operations performed per microsecond. It is currently not able to independently vary the read and write bandwidths as this throttling feature affects all DDR operations. This limitation is not an issue for this evaluation as OLTP workloads are not bandwidth intensive [65].

The emulator exposes two interfaces to access the NVM storage:

Allocator Interface: The emulator uses a NVM-aware memory allocator that provides the POSIX malloc interface. Applications obtain and use NVM directly using this interface. Internally, *libnuma* library [4] is used to ensure that the application allocates memory only from emulated NVM and not regular DRAM. When the DBMS writes to a memory location, the written (dirty) data may reside indefinitely in the volatile CPU caches and not propagate immediately to NVM. We use a hardware write barrier primitive (`SFENCE`) to guarantee the durability of writes to NVM when they are flushed from CPU caches.

Filesystem Interface: The emulator exposes a NVM-backed volume to the OS through a special filesystem that is optimized for non-volatile memory [27]. This allows applications to use the POSIX filesystem interface to read/write data to files stored on



(a) Sequential Writes **(b)** Random Writes

Figure 1: Comparison of the durable write bandwidth of Intel Lab's NVM emulator using the allocator and filesystem interfaces.

NVM. Normally, in a block-oriented filesystem, file I/O requires two copies; one involving the block device and another involving the user buffer. The emulator's optimized filesystem, however, requires only one copy between the file and the user buffers. This improves the file I/O performance by 7–10× compared to block-oriented filesystems like EXT4. The filesystem interface allows existing DBMSs to make use of NVM for durable storage.

Both of the above interfaces use memory from the emulated NVM. The key difference, however, is that the filesystem interface supports a naming mechanism that ensures that file offsets are valid after the system restarts. The downside of the filesystem interface is that it requires the application's writes to go through the kernel's virtual filesystem (VFS) layer. In contrast, when the application uses the allocator interface, it can write to and read from NVM directly within userspace. However, the allocator interface does not automatically provide a naming mechanism that is valid after a system restart. We use a memory allocator that is designed for NVM to overcome this limitation.

2.3 NVM-aware Memory Allocator

An NVM-aware memory allocator for a DBMS needs to satisfy two key requirements. The first is that it should provide a *durability* mechanism to ensure that modifications to data stored on NVM are persisted. This is necessary because the changes made by a transaction to a location on NVM may still reside in volatile CPU caches when the transaction commits. If a power failure happens before the changes reach NVM, then these changes are lost. The allocator exposes a special API call to provide this durability mechanism. Internally, the allocator first writes back the cache lines containing the data from any level of the cache hierarchy to NVM using `CLFLUSH` [2] instruction. Then, it issues a `SFENCE` instruction to ensure that the data flushed from the CPU caches becomes durable. Otherwise, this data might still be buffered in the memory controller and lost in case of a power failure. From here on, we refer to the above mentioned durability mechanism as the *sync* primitive.

The second requirement is that it should provide a *naming* mechanism for allocations so that pointers to locations in memory are valid even after the system restarts. The allocator ensures that the virtual memory addresses assigned to a memory-mapped region never change. With this mechanism, a pointer to a NVM location is mapped to the same virtual location after the OS or DBMS restarts. We refer to these pointers as *non-volatile pointers* [51].

The NVM allocator that we use in our evaluation is based on the open-source NVM-related *libpmem* library [58]. We extended this allocator to follow a rotating best-fit allocation policy and to support multi-threaded usage. The allocator directly maps the NVM to its address space. Unlike the filesystem interface, accessing a region of memory obtained from this allocator does not require copying data to user buffers. After an OS restart, the allocator reclaims memory that has not been persisted and restores its internal metadata to a consistent state. This recovery mechanism is required only after the OS restarts and not after the DBMS restarts, because the allocator handles memory management for all applications.

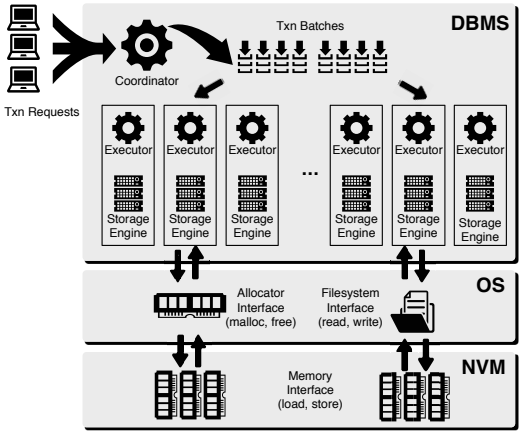


Figure 2: An overview of the architecture of the DBMS testbed.

To show that accessing NVM through the allocator interface is faster than using the filesystem interface, we compare them using a micro-benchmark. In this experiment, the application performs durable writes to NVM using the two interfaces with sequential and random access patterns. The application performs durable writes using the filesystem's *fsync* system call and the allocator's sync primitive. We vary the size of the data chunk that the application writes from 1 to 256 bytes. The results in Fig. 1 show that NVM-aware allocator delivers 10–12× higher write bandwidth than the filesystem. The performance gap is more evident when the application writes out small data chunks sequentially. We also note that the gap between sequential and random write bandwidth is lower than that observed in other durable storage technologies.

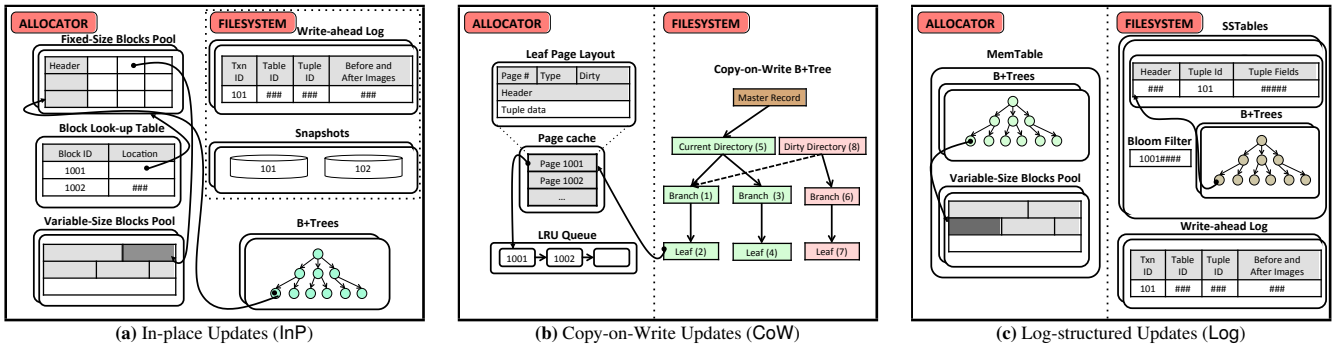
We now describe the DBMS testbed that we built to evaluate storage and recovery methods for DBMSs running on NVM. As we discuss in subsequent sections, we use non-volatile pointers to build non-volatile data structures that are guaranteed to be consistent after the OS or DBMS restarts. These data structures, along with the allocator interface, are used to build NVM-aware storage engines.

3. DBMS TESTBED

We developed a lightweight DBMS to evaluate different storage architecture designs for OLTP workloads. We did not use an existing DBMS as that would require significant changes to incorporate the storage engines into a single system. Although some DBMSs support a pluggable storage engine back-end (e.g., MySQL, MongoDB), modifying them to support NVM would still require significant changes. We also did not want to taint our measurements with features that are not relevant to our evaluation.

The architecture of the testbed running on the hardware emulator is depicted in Fig. 2. The DBMS's internal coordinator receives incoming transaction requests from the application and then invokes the target stored procedure. As a transaction executes in the system, it invokes queries to read and write tuples from the database. These requests are passed through a query executor that invokes the necessary operations on the DBMS's active storage engine.

The DBMS uses *pthreads* to allow multiple transactions to run concurrently in separate worker threads. It executes as a single process with the number of worker threads equal to the number of cores, where each thread is mapped to a different core. Since we do not want the DBMS's concurrency control scheme to interfere with our evaluation, we partition the database and use a lightweight



(a) In-place Updates (InP) **(b)** Copy-on-Write Updates (CoW) **(c)** Log-structured Updates (Log)

Figure 3: Architectural layout of the three traditional storage engines supported in the DBMS testbed. The engine components accessed using the allocator interface and those accessed using the filesystem interface are bifurcated by the dashed line.

locking scheme where transactions are executed serially at each partition based on timestamp ordering [59]. Using a DBMS that supports a pluggable back-end allows us to compare the performance characteristics of different storage and recovery methods on a single platform. We implemented three storage engines that use different approaches for supporting durable updates to a database: (1) *in-place* updates engine, (2) *copy-on-write* updates engine, and (3) *log-structured* updates engine. Each engine also supports both primary and secondary indexes.

We now describe these engines in detail. For each engine, we first discuss how they apply changes made by transactions to the database and then how they ensure durability after a crash. All of these engines are based on the architectures found in state-of-the-art DBMSs. That is, they use memory obtained using the allocator interface as volatile memory and do not exploit NVM's persistence. Later in Section 4, we present our improved variants of these engines that are optimized for NVM.

3.1 In-Place Updates Engine (InP)

The first engine uses the most common storage engine strategy in DBMSs. With *in-place* updates, there is only a single version of each tuple at all times. When a transaction updates a field for an existing tuple, the system writes the new value directly on top of the original one. This is the most efficient method of applying changes, since the engine does not make a copy of the tuple first before updating it and only the updated fields are modified. The design of this engine is based on VoltDB [5], which is a memory-oriented DBMS that does not contain legacy disk-oriented DBMS components like a buffer pool. The InP engine uses the STX B+tree library for all of its indexes [10].

Storage: Fig. 3a illustrates the architecture of the InP engine. The storage area for tables is split into separate pools for fixed-sized *blocks* and variable-length *blocks*. Each block consists of a set of *slots*. The InP engine stores the table's tuples in fixed-size slots. This ensures that the tuples are byte-aligned and the engine can easily compute their offsets. Any field in a table that is larger than 8 bytes is stored separately in a variable-length slot. The 8-byte location of that slot is stored in that field's location in the tuple.

The tables' tuples are unsorted within these blocks. For each table, the DBMS maintains a list of unoccupied tuple slots. When a transaction deletes a tuple, the deleted tuple's slot is added to this pool. When a transaction inserts a tuple into a table, the engine first checks the table's pool for an available slot. If the pool is empty, then the engine allocates a new fixed-size block using the allocator interface. The engine also uses the allocator interface to maintain the indexes and stores them in memory.

Recovery: Since the changes made by transactions committed after the last checkpoint are not written to "durable" storage, the InP engine maintains a durable *write-ahead log* (WAL) in the filesystem to assist in recovery from crashes and power failures. WAL records the transactions' changes before they are applied to DBMS [29]. As transactions execute queries that modify the database, the engine appends a new entry to the WAL for those changes. Each entry contains the transaction identifier, the table modified, the tuple identifier, and the before/after tuple images depending on the operation.

The most well-known recovery protocol for in-place updates is ARIES [47]. With ARIES, the engine periodically takes checkpoints that are stored on the filesystem to bound recovery latency and reduce the storage space consumed by the log. In our implementation, we compress (*gzip*) the checkpoints on the filesystem to reduce their storage footprint on NVM. During recovery, the engine first loads the last checkpoint. It then replays the log to ensure that the changes made by transactions committed since the checkpoint are present in the database. Changes made by uncommitted transactions at the time of failure are not propagated to the database. The InP engine uses a variant of ARIES that is adapted for main memory DBMSs with a byte-addressable storage engine [45]. As we do not store physical changes to indexes in this log, all of the tables' indexes are rebuilt during recovery because they may have been corrupted [45].

3.2 Copy-on-Write Updates Engine (CoW)

The second storage engine performs *copy-on-write* updates where instead of modifying the original tuple, it creates a copy of the tuple and then modifies that copy. As the CoW engine never overwrites committed data, it does not need to record changes in a WAL for recovery. The CoW engine instead uses different look-up *directories* for accessing the versions of tuples in the database. With this approach, known as *shadow paging* in IBM's System R [33], the DBMS maintains two look-up directories at all times: (1) the *current* directory, and (2) the *dirty* directory. The current directory points to the most recent versions of the tuples and only contains the effects of committed transactions. The dirty directory points to the versions of tuples being modified by active transactions. To ensure that the transactions are isolated from the effects of uncommitted transactions, the engine maintains a *master record* that always points to the current directory. Fig. 3b presents the architecture of the CoW engine. After applying the changes on the copy of the tuple, the engine updates the dirty directory to point to the new version of the tuple. When the transaction commits, the engine updates the master record atomically to point to the dirty directory. The engine maintains an internal page cache to keep the hot pages in memory.

System R implements shadow paging by copying the current directory to create the new dirty directory after every commit operation. But, creating the dirty directory in this manner incurs

high I/O overhead. The CoW engine uses LMDB's copy-on-write B+trees [56, 16, 36] to implement shadow paging efficiently. Fig. 3b illustrates an update operation on a CoW B+tree. When the engine modifies the leaf node 4 in the current directory, it only needs to make a copy of the internal nodes lying along the path from that leaf node up to the root of the current version. The current and dirty directories of the copy-on-write B+tree share the rest of the tree. This significantly reduces the I/O overhead of creating the dirty directory as only a fraction of the B+tree is copied. To further reduce the overhead of shadow paging, the CoW engine uses a group commit mechanism that batches the changes made by a group of transactions before committing the dirty directory.

Storage: The CoW engine stores the directories on the filesystem. The tuples in each table are stored in a HDD/SSD-optimized format where all the tuple's fields are inlined. This avoids expensive random accesses that are required when some fields are not inlined. Each database is stored in a separate file and the master record for the database is located at a fixed offset within the file. It supports secondary indexes as a mapping of secondary keys to primary keys.

The downside of the CoW engine is that it creates a new copy of tuple even if a transaction only modifies a subset of the tuple's fields. The engine also needs to keep track of references to tuples from different versions of the copy-on-write B+tree so that it can reclaim the storage space consumed by old unreferenced tuple versions [9]. As we show in Section 5.3, this engine has high write amplification (i.e., the amount of data written to storage is much higher compared to the amount of data written by the application). This increases wear on the NVM device thereby reducing its lifetime.

Recovery: If the DBMS crashes before the master record is updated, then the changes present in the dirty directory are not visible after restart. Hence, there is no recovery process for the CoW engine. When the DBMS comes back on-line, the master record points to the current directory that is guaranteed to be consistent. The dirty directory is garbage collected asynchronously, since it only contains the changes of uncommitted transactions.

3.3 Log-structured Updates Engine (Log)

Lastly, the third storage engine uses a *log-structured* update policy. This approach originated from log-structured filesystems [57], and then it was adapted to DBMSs as *log-structured merge* (LSM) trees [50] for write-intensive workloads. The LSM tree consists of a collection of *runs* of data. Each run contains an ordered set of entries that record the changes performed on tuples. Runs reside either in volatile memory (i.e., *MemTable*) or on durable storage (i.e., *SSTables*) with their storage layout optimized for the underlying storage device. The LSM tree reduces write amplification by batching the updates in MemTable and periodically cascading the changes to durable storage [50]. The design for our Log engine is based on Google's LevelDB [22], which implements the log-structured update policy using LSM trees.

Storage: Fig. 3c depicts the architecture of the Log engine. The Log engine uses a *leveled* LSM tree [40], where each level in the tree contains the changes for a single run. The data starts from the MemTable stored in the topmost level and propagates down to SSTables stored in lower parts of the tree over time. The size of the run stored in a given level is k times larger than that of the run stored in its parent, where k is the growth factor of the tree. The Log engine allows us to control the size of the MemTable and the growth factor of the tree. It first stores the tuple modifications in a memory-optimized format using the allocator interface in the MemTable. The MemTable contains indexes to handle point and range queries efficiently. When the size of the MemTable exceeds a threshold, the engine flushes it to the filesystem as an immutable SSTable stored in a separate file. The Log engine also constructs a Bloom filter [12] for each SSTable to quickly determine at runtime whether it contains entries associated with a tuple to avoid unnecessary index look-ups.

The contents of the MemTable are lost after system restart. Hence, to ensure durability, the Log engine maintains a WAL in the filesystem. The engine first records the changes in the log and then applies the changes to the MemTable. The log entry contains the transaction identifier, the table modified, the tuple identifier, and the before/after images of the tuple depending on the type of operation. To reduce the I/O overhead, the engine batches log entries for a group of transactions and flushes them together.

The log-structured update approach performs well for write-intensive workloads as it reduces random writes to durable storage. The downside of the Log engine is that it incurs high read amplification (i.e., the number of reads required to fetch the data is much higher than that actually needed by the application). To retrieve a tuple, the Log engine first needs to look-up the indexes of all the runs of the LSM tree that contain entries associated with the desired tuple in order to reconstruct the tuple [1]. To reduce this read amplification, the Log engine performs a periodic *compaction* process that merges a subset of SSTables. First, the entries associated with a tuple in different SSTables are merged into one entry in a new SSTable. Tombstone entries are used to identify purged tuples. Then, the engine builds indexes for the new SSTable.

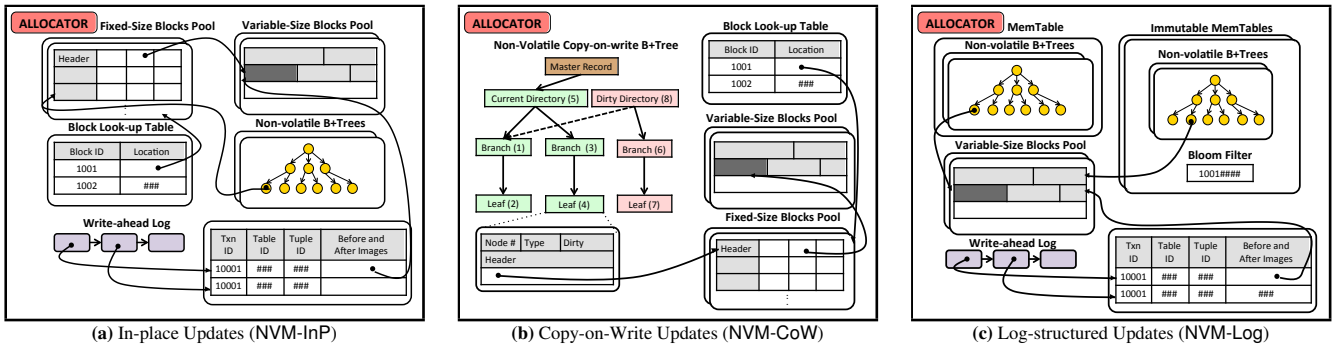
Recovery: During recovery, the Log engine rebuilds the MemTable using the WAL, as the changes contained in it were not written onto durable storage. It first replays the log to ensure that the changes made by committed transactions are present. It then removes any changes performed by uncommitted transactions, thereby bringing the MemTable to a consistent state.

4. NVM-AWARE ENGINES

All of the engines described above are derived from existing DBMS architectures that are predicated on a two-tier storage hierarchy comprised of volatile DRAM and a non-volatile HDD/SSD. These storage devices have distinct hardware constraints and performance properties [54]. First, the read and write latency of non-volatile storage is several orders of magnitude higher than DRAM. Second, the DBMS accesses data on non-volatile storage at block-granularity, while with DRAM it accesses data at byte-granularity. Third, the performance gap between sequential and random accesses is greater for non-volatile storage compared to DRAM.

The traditional engines were designed to account for and reduce the impact of these differences. For example, they maintain two layouts of tuples depending on the storage device. Tuples stored in memory can contain non-inlined fields because DRAM is byte-addressable and handles random accesses efficiently. In contrast, fields in tuples stored on durable storage are inlined to avoid random accesses because they are more expensive. To amortize the overhead for accessing durable storage, these engines batch writes and flush them in a deferred manner.

Many of these techniques, however, are unnecessary in a system with a NVM-only storage hierarchy [23, 27, 49]. As shown in Table 1, the access latencies of NVM are orders of magnitude shorter than that of HDDs and SSDs. Further, the performance gap between sequential and random accesses on NVM is comparable to that of DRAM. We therefore adapt the storage and recovery mechanisms of these traditional engines to exploit NVM's characteristics. We refer to these optimized storage engines as the *NVM-aware* engines.



(a) In-place Updates (NVM-InP) **(b)** Copy-on-Write Updates (NVM-CoW) **(c)** Log-structured Updates (NVM-Log)

Figure 4: NVM-Aware Engines – Architectural layout of the NVM-optimized storage engines.

As we show in our evaluation in Section 5, these engines deliver higher throughput than their traditional counterparts while still ensuring durability. They reduce write amplification using NVM's persistence thereby expanding the lifetime of the NVM device. These engines use only the allocator interface described in Section 2.3 with NVM-optimized data structures [49, 62].

Table 2 presents an overview of the steps performed by the NVM-aware storage engines, while executing the primitive database operations. We note that the engine performs these operations within the context of a transaction. For instance, if the transaction aborts while executing an operation, it must undo the effects of any earlier operation performed by the transaction.

4.1 In-Place Updates Engine (NVM-InP)

One of the main problems with the InP engine described in Section 3.1 is that it has high rate of data duplication. When a transaction inserts a tuple, the engine records the tuple's contents in the WAL and then again in the table storage area. The InP engine's logging infrastructure also assumes that the system's durable storage device has orders of magnitude higher write latency compared to DRAM. It therefore batches multiple log records and flushes them periodically to the WAL using sequential writes. This approach, however, increases the mean response latency as transactions need to wait for the group commit operation.

Given this, we designed the NVM-InP engine to avoid these issues. Now when a transaction inserts a tuple, rather than copying the tuple to the WAL, the NVM-InP engine only records a non-volatile pointer to the tuple in the WAL. This is sufficient because both the pointer and the tuple referred to by the pointer are stored on NVM. Thus, the engine can use the pointer to access the tuple after the system restarts without needing to re-apply changes in the WAL. It also stores indexes as non-volatile B+trees that can be accessed immediately when the system restarts without rebuilding.

Storage: The architecture of the NVM-InP engine is shown in Fig. 4a and Table 2 presents an overview of the steps to perform different operations. The engine stores tuples and non-inlined fields using fixed-size and variable-length slots, respectively. To reclaim the storage space of tuples and non-inlined fields inserted by uncommitted transactions after the system restarts, the NVM-InP engine maintains *durability state* in each slot's header. A slot can be in one of three states - unallocated, allocated but not persisted, or persisted. After the system restarts, slots that are allocated but not persisted transition back to unallocated state.

The NVM-InP engine stores the WAL as a non-volatile linked list. It appends new entries to the list using an atomic write. Each entry contains the transaction identifier, the table modified, the tuple identifier, and pointers to the operation's changes. The changes include tuple pointers for insert operation and field pointers for update operations on non-inlined fields. The engine persists this entry before updating the slot's state as persisted. If it does not ensure this ordering, then the engine cannot reclaim the storage space consumed by uncommitted transactions after the system restarts, thereby causing non-volatile memory leaks. After all of the transaction's changes are safely persisted, the engine truncates the log.

The engine supports primary and secondary indexes using non-volatile B+trees that it maintains using the allocator interface. We modified the STX B+tree library so that all operations that alter the index's internal structure are atomic [49, 62]. For instance, when adding an entry to a B+tree node, instead of inserting the key in a sorted order, it appends the entry to a list of entries in the node. This modification is necessary because if the entry crosses cache line boundaries, the cache line write-backs required to persist the entry need not happen atomically. Our changes to the library ensure that the engine can safely access the index immediately after the system restarts as it is guaranteed to be in a consistent state.

Recovery: The effects of committed transactions are durable after the system restarts because the NVM-InP engine immediately persists the changes made by a transaction when it commits. The engine therefore does not need to replay the log during recovery. But the changes of uncommitted transactions may be present in the database because the memory controller can evict cache lines containing those changes to NVM at any time [48]. The NVM-InP engine therefore needs to undo those transactions using the WAL.

To undo an insert operation, the engine releases the tuple's storage space using the pointer recorded in the WAL entry and then removes entries associated with the tuple in the indexes. In case of an update operation, the engine restores the tuple's state using the before image. If the after image contains non-inlined tuple fields, then the engine frees up the memory occupied by those fields. For a delete operation, it only needs to update the indexes to point to the original tuple. To handle transaction rollbacks and DBMS recovery correctly, the NVM-InP engine releases storage space occupied by tuples or non-inlined fields only after it is certain that they are no longer required. As this recovery protocol does not include a redo process, the NVM-InP engine has a short recovery latency that only depends on the number of uncommitted transactions.

4.2 Copy-on-Write Updates Engine (NVM-CoW)

The original CoW engine stores tuples in self-containing blocks without pointers in the copy-on-write B+tree on the filesystem. The problem with this engine is that the overhead of propagating modifications to the dirty directory is high; even if a transaction only modifies one tuple, the engine needs to copy the entire block to the filesystem. When a transaction commits, the CoW engine uses the filesystem interface to flush the dirty blocks and updates the master record (stored at a fixed location in the file) to point to the root of

	NVM-InP Engine	NVM-CoW Engine	NVM-Log Engine
INSERT	• Sync tuple with NVM. • Record tuple pointer in WAL. • Sync log entry with NVM. • Mark tuple state as persisted. • Add tuple entry in indexes.	• Sync tuple with NVM. • Store tuple pointer in dirty dir. • Update tuple state as persisted. • Add tuple entry in secondary indexes.	• Sync tuple with NVM. • Record tuple pointer in WAL. • Sync log entry with NVM. • Mark tuple state as persisted. • Add tuple entry in MemTable.
UPDATE	• Record tuple changes in WAL. • Sync log entry with NVM. • Perform modifications on the tuple. • Sync tuple changes with NVM.	• Make a copy of the tuple. • Apply changes on the copy. • Sync tuple with NVM. • Store tuple pointer in dirty dir. • Update tuple state as persisted. • Add tuple entry in secondary indexes.	• Record tuple changes in WAL. • Sync log entry with NVM. • Perform modifications on the tuple. • Sync tuple changes with NVM.
DELETE	• Record tuple pointer in WAL. • Sync log entry with NVM. • Discard entry from table and indexes. • Reclaim space at the end of transaction.	• Remove tuple pointer from dirty dir. • Discard entry from secondary indexes. • Recover tuple space immediately.	• Record tuple pointer in WAL. • Sync log entry with NVM. • Mark tuple tombstone in MemTable. • Reclaim space during compaction.
SELECT	• Find tuple pointer using index/table. • Retrieve tuple contents.	• Locate tuple pointer in appropriate dir. • Fetch tuple contents from dir.	• Find tuple entries in relevant LSM runs. • Rebuild tuple by coalescing entries.

Table 2: An overview of the steps performed by the NVM-aware storage engines, while executing primitive database operations. The syncing mechanism is implemented using `CLFLUSH` and `SFENCE` instructions on the hardware emulator. We describe the *sync* primitive in Section 2.3.

the dirty directory [16]. These writes are expensive as they need to switch the privilege level and go through the kernel's VFS path.

The NVM-CoW engine employs three optimizations to reduce these overheads. First, it uses a non-volatile copy-on-write B+tree that it maintains using the allocator interface. Second, the NVM-CoW engine directly persists the tuple copies and only records non-volatile tuple pointers in the dirty directory. Lastly, it uses the lightweight durability mechanism of the allocator interface to persist changes in the copy-on-write B+tree.

Storage: Fig. 4b depicts the architecture of the NVM-CoW engine. The storage area for tuples is spread across separate pools for fixed-sized and variable-length data. The engine maintains the durability state of each slot in both pools similar to the NVM-InP engine. The NVM-CoW engine stores the current and dirty directory of the non-volatile copy-on-write B+tree using the allocator interface. We modified the B+tree from LMDB [16] to handle modifications at finer granularity to exploit NVM's byte addressability. The engine maintains the master record using the allocator interface to support efficient updates. When the system restarts, the engine can safely access the current directory using the master record because that directory is guaranteed to be in a consistent state. This is because the data structure is append-only and the data stored in the current directory is never overwritten.

The execution steps for this engine are shown in Table 2. The salient feature of this engine's design is that it avoids the transformation and copying costs incurred by the CoW engine. When a transaction updates a tuple, the engine first makes a copy and then applies the changes to that copy. It then records only the non-volatile tuple pointer in the dirty directory. The engine also batches transactions to amortize the cost of persisting the current directory. To commit a batch of transactions, it first persists the changes performed by uncommitted transactions. It then persists the contents of the dirty directory. Finally, it updates the master record using an atomic durable write to point to that directory. The engine orders all of these writes using memory barriers to ensure that only committed transactions are visible after the system restarts.

Recovery: As the NVM-CoW engine never overwrites committed data, it does not have a recovery process. When the system restarts, it first accesses the master record to locate the current directory. After that, it can start handling transactions. The storage space consumed by the dirty directory at the time of failure is asynchronously reclaimed by the NVM-aware allocator.

4.3 Log-structured Updates Engine (NVM-Log)

The Log engine batches all writes in the MemTable to reduce random accesses to durable storage [50, 43]. The benefits of this approach, however, are not as evident for a NVM-only storage hierarchy because the performance gap between sequential and random accesses is smaller. The original log-structured engine that we described in Section 3.3 incurs significant overhead from periodically flushing MemTable to the filesystem and compacting SSTables to bound read amplification. Similar to the NVM-InP engine, the NVM-Log engine records all the changes performed by transactions on a WAL stored on NVM.

Our NVM-Log engine avoids data duplication in the MemTable and the WAL as it only records non-volatile pointers to tuple modifications in the WAL. Instead of flushing MemTable out to the filesystem as a SSTable, it only marks the MemTable as immutable and starts a new MemTable. This immutable MemTable is physically stored in the same way on NVM as the mutable MemTable. The only difference is that the engine does not propagate writes to the immutable MemTables. We also modified the compaction process to merge a set of these MemTables to generate a new larger MemTable. The NVM-Log engine uses a NVM-aware recovery protocol that has lower recovery latency than its traditional counterpart.

Storage: As shown in Fig. 4c, the NVM-Log engine uses an LSM tree to store the database. Each level of the tree contains a sorted run of data. Similar to the Log engine, this engine first stores all the changes performed by transactions in the MemTable which is the topmost level of the LSM tree. The changes include tuple contents for insert operation, updated fields for update operation and tombstone markers for delete operation. When the size of the MemTable exceeds a threshold, the NVM-Log engine marks it as immutable and starts a new MemTable. We modify the periodic compaction process the engine performs for bounding read amplification to merge a set of immutable MemTables and create a new MemTable. The engine constructs a Bloom filter [12] for each immutable MemTable to minimize unnecessary index look-ups.

Similar to the Log engine, the NVM-Log engine maintains a WAL. The purpose of the WAL is not to rebuild the MemTable, but rather to undo the effects of uncommitted transactions from the MemTable. An overview of the operations performed by the NVM-Log engine is shown in Table 2. Like the NVM-InP engine, this new engine also stores the WAL as a non-volatile linked list of entries. When a transaction inserts a tuple, the engine first flushes the tuple to NVM and records the non-volatile tuple pointer in a WAL entry. It then persists the log entry and marks the tuple as persisted. Finally,

it adds an entry in the MemTable indexes. After the transaction commits, the engine truncates the relevant log entry because the changes recorded in MemTable are durable. Its logging overhead is lower than the Log engine as it records less data and maintains the WAL using the allocator interface. The engine uses non-volatile B+trees [49, 62], described in Section 4.1, as MemTable indexes. Hence, it does not need to rebuild its indexes upon restarting.

Recovery: When the transaction commits, all the changes performed by the transaction are persisted in the in-memory component. During recovery, the NVM-Log engine only needs to undo the effects of uncommitted transactions on the MemTable. Its recovery latency is therefore lower than the Log engine as it no longer needs to rebuild the MemTable.

5. EXPERIMENTAL ANALYSIS

In this section, we present our analysis of the six different storage engine implementations. Our DBMS testbed allows us to evaluate the throughput, the number of reads/writes to the NVM device, the storage footprint, and the time that it takes to recover the database after restarting. We also use the *perf* toolkit to measure additional, lower-level hardware metrics of the system for each experiment [3].

The experiments were all performed on Intel Lab's NVM hardware emulator [27]. It contains a dual-socket Intel Xeon E5-4620 processor. Each socket has eight cores running at 2.6 GHz. It dedicates 128 GB of DRAM for the emulated NVM and its L3 cache size is 20 MB. We use the Intel memory latency checker [63] to validate the emulator's latency and bandwidth settings. The engines access NVM storage using the allocator and filesystem interfaces of the emulator as described in Section 2.2. We set up the DBMS to use eight partitions in all of the experiments. We configure the node size of the STX B+tree and the CoW B+tree implementations to be 512 B and 4 KB respectively. All transactions execute with the same serializable isolation level and durability guarantees.

5.1 Benchmarks

We first describe the benchmarks we use in our evaluation. The tables in each database are partitioned in such way that there are only single-partition transactions [52].

YCSB: This is a widely-used key-value store workload from Yahoo! [20]. It is representative of the transactions handled by web-based companies. It contains a single table comprised of tuples with a primary key and 10 columns of random string data, each 100 bytes in size. Each tuple's size is approximately 1 KB. We use a database with 2 million tuples ($\sim$2 GB).

The workload consists of two transaction types: (1) a *read* transaction that retrieves a single tuple based on its primary key, and (2) an *update* transaction that modifies a single tuple based on its primary key. We use four types of workload mixtures that allow us to vary the I/O operations that the DBMS executes. These mixtures represent different ratios of *read* and *update* transactions:

- **Read-Only:** 100% *reads*
- **Read-Heavy:** 90% *reads*, 10% *updates*
- **Balanced:** 50% *reads*, 50% *updates*
- **Write-Heavy:** 10% *reads*, 90% *updates*

We modified the YCSB workload generator to support two different levels of skew in the tuple access patterns that allows us to create a localized hotspot within each partition:

- **Low Skew:** 50% of the transactions access 20% of the tuples.
- **High Skew:** 90% of the transactions access 10% of the tuples.

For each workload mixture and skew setting pair, we pre-generate a fixed workload of 8 million transactions that is divided evenly among the DBMS's partitions. Using a fixed workload that is the same across all the engines allows us to compare their storage footprints and read/write amplification.

TPC-C: This benchmark is the current industry standard for evaluating the performance of OLTP systems [61]. It simulates an order-entry environment of a wholesale supplier. The workload consists of five transaction types, which keep track of customer orders, payments, and other aspects of a warehouse. Transactions involving database modifications comprise around 88% of the workload. We configure the workload to contain eight warehouses and 100,000 items. We map each warehouse to a single partition. The initial storage footprint of the database is approximately 1 GB.

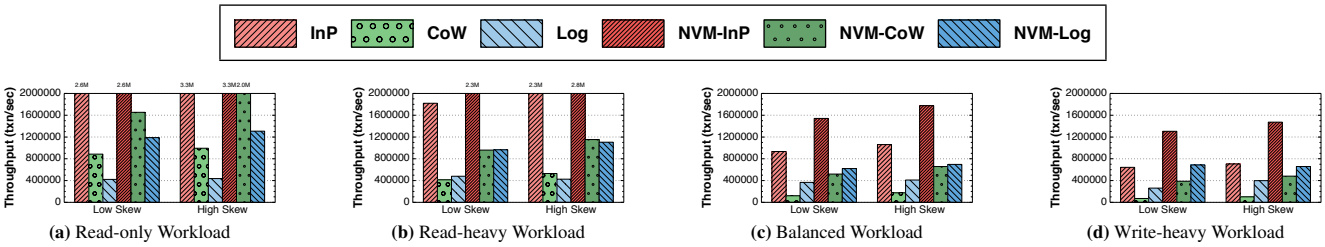
5.2 Runtime Performance

We begin with an analysis of the impact of NVM's latency on the performance of the storage engines. To obtain insights that are applicable for various NVM technologies, we run the YCSB and TPC-C benchmarks under three latency configurations on the emulator: (1) default DRAM latency configuration (160 ns), (2) a *low* NVM latency configuration that is 2$\times$ higher than DRAM latency (320 ns), and (3) a *high* NVM latency configuration that is 8$\times$ higher than DRAM latency (1280 ns). Prior work [27] has shown that the sustained bandwidth of NVM is likely to be lower than that of DRAM. We therefore leverage the bandwidth throttling mechanism in the hardware emulator [27] to throttle the NVM bandwidth to 9.5 GB/s, which is 8$\times$ lower than the available DRAM bandwidth on the platform. We execute all workloads three times on each engine and report the average throughput.

YCSB: Figs. 5 to 7 present the throughput observed with the YCSB benchmark while varying the workload mixture and skew settings under different latency configurations. We first consider the read-only workload results shown in Figs. 5a, 6a and 7a. These results provide an upper bound on performance since transactions do not modify the database and the engines therefore do not need to flush changes from CPU caches to NVM during execution.

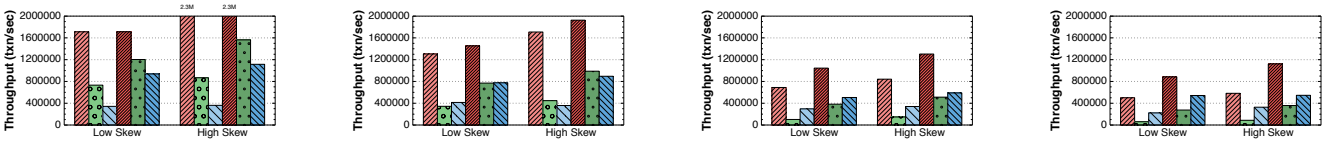
The most notable observation is that the NVM-InP engine is not faster than the InP engine for both skew settings. This is because both engines perform reads using the allocator interface. The CoW engine's throughput is lower than the in-place updates engine because for every transaction, it fetches the master record and then looks-up the tuple in the current directory. As the NVM-CoW engine accesses the master record and the non-volatile copy-on-write B+tree efficiently using the allocator interface, it is 1.9–2.1$\times$ faster than the CoW engine. The Log engine is the slowest among all the engines because it coalesces entries spread across different LSM tree components to reconstruct tuples. The NVM-Log engine accesses the immutable MemTables using the allocator interface and delivers 2.8$\times$ higher throughput compared to its traditional counterpart. We see that increasing the workload skew improves the performance of all the engines due to caching benefits. The benefits are most evident for the InP and NVM-InP engines; they achieve 1.3$\times$ higher throughput compared to the low skew setting. The performance gains due to skew are minimal in case of the Log and NVM-Log engines due to tuple coalescing costs.

We also observe that the performance gap between the two types of engines decreases in the read-only workload when we increase the NVM latency. In the high latency configuration, the NVM-CoW and the NVM-Log engines are 1.4$\times$ and 2.5$\times$ faster than their traditional counterparts. This is because the benefits of accessing data structures using the allocator interface are masked by slower



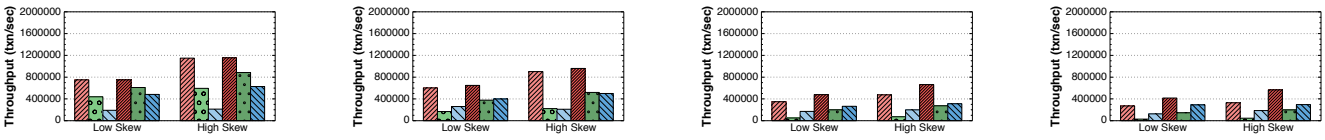
(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 5: YCSB Performance (DRAM Latency) – The throughput of the engines for the YCSB benchmark without any latency slowdown.



(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 6: YCSB Performance (Low Latency) – The throughput of the engines for the YCSB benchmark under the low NVM latency configuration (2×).



(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 7: YCSB Performance (High Latency) – The throughput of the engines for the YCSB benchmark under the high NVM latency configuration (8×).

NVM loads. The engines' throughput decreases sub-linearly with respect to the increased NVM latency. For example, with 8× higher latency, the throughput of the engines only drop by 2–3.4×. The NVM-aware engines are more sensitive to the increase in latency as they do not incur tuple transformation and copying costs that dampen the effect of slower NVM accesses in the traditional engines.

For the read-heavy workload, the results shown in Figs. 5b, 6b and 7b indicate that the throughput decreases for all the engines compared to the read-only workload because they must flush transactions' changes to NVM. Unlike before where the two engines had the same performance, in this workload we observe that the NVM-InP engine is 1.3× faster than the InP engine due to lower logging overhead. The performance of the CoW engine drops compared to its performance on the read-only workload because of the overhead of persisting the current directory. The drop is less prominent in the high skew workload because the updates are now concentrated over a few hot tuples and therefore the number of copy-on-write B+tree nodes that are copied when creating the dirty directory is smaller.

The benefits of our optimizations are more prominent for the balanced and write-heavy workloads. For the NVM-InP and the NVM-Log engines, we attribute this to lower logging overhead. In case of the NVM-CoW engine, this is because it does not have to copy and transform tuples from the filesystem whenever it modifies them. This allows this engine to achieve 4.3–5.5× higher throughput than the CoW engine. The performance gap between the Log and the CoW engines decreases because the former incurs lower tuple coalescing costs in these workloads. The Log engine is therefore 1.6–4.1× faster than the CoW engine. It still lags behind the InP engine, however, because batching updates in the MemTable are not as beneficial in the NVM-only storage hierarchy. With increased latency, the throughput of all the engines decreases less on these write-intensive workloads compared to the workloads that contain more reads. The throughput does not drop linearly with increasing NVM latency. With an 8× increase in latency, the throughput of the engines only drops by 1.8–2.9×. We attribute this to the effects of caching and memory-level parallelism in the emulator.

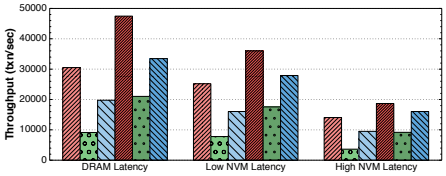


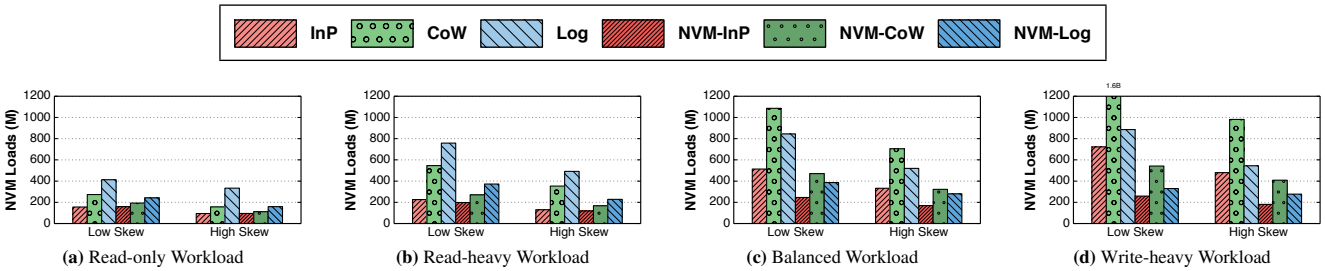
Figure 8: TPC-C Throughput – The performance of the engines for TPC-C benchmark for all three NVM latency settings.

TPC-C: Fig. 8 shows the engines' throughput while executing TPC-C under different latency configurations. Among all the engines, the NVM-InP engine performs the best. The NVM-aware engines are 1.8–2.1× faster than the traditional engines. The NVM-CoW engine exhibits the highest speedup of 2.3× over the CoW engine. We attribute this to the write-intensive nature of the TPC-C benchmark. Under the high NVM latency configuration, the NVM-aware engines deliver 1.7–1.9× higher throughput than their traditional counterparts. These trends closely follow the results for the write-intensive workload mixture in the YCSB benchmark. The benefits of our optimizations, however, are not as significant as previously observed with the YCSB benchmark. This is because the TPC-C transactions' contain more complex program logic and execute more queries per transaction.

5.3 Reads & Writes

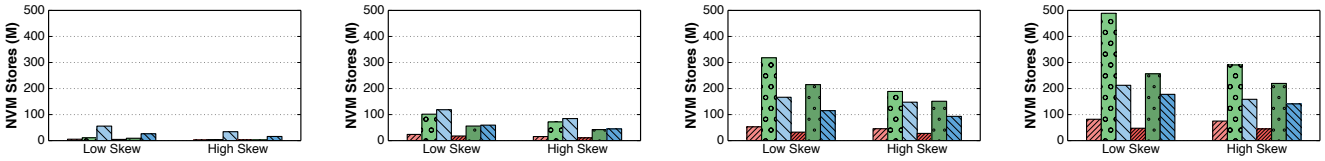
We next measure the number of times that the storage engines access the NVM device while executing the benchmarks. This is important because the number of write cycles per bit is limited in different NVM technologies as shown in Table 1. We compute these results using hardware performance counters on the emulator with the *perf* framework [3]. These counters track the number of loads (i.e. reads) from and stores (i.e. writes) to the NVM device during execution. In each trial, the engines' access measurements are collected after loading the initial database.

YCSB: The results for NVM reads and writes while executing the YCSB benchmark are shown in Figs. 9 and 10, respectively. In



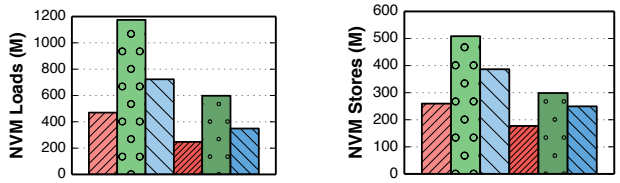
(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 9: YCSB Reads – The number of load operations executed by the engines while running the YCSB workload.



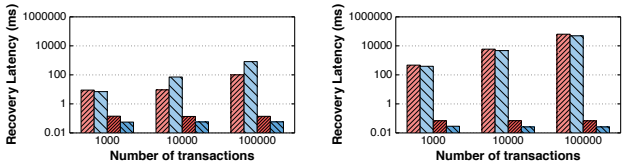
(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 10: YCSB Writes – The number of store operations executed by the engines while running the YCSB workload.



(a) Reads **(b)** Writes

Figure 11: TPC-C Reads & Writes – The number of load and store operations executed by the engines while running the TPC-C benchmark.



(a) YCSB **(b)** TPC-C

Figure 12: Recovery Latency – The amount of time that the engines take to restore the database to a consistent state after a restart.

the read-only workload, we observe that the Log engine performs the most load operations due to tuple coalescing. The NVM-aware engines perform up to 53% fewer loads due to better cache locality as they do not perform any tuple deserialization operations. When we increase the workload skew, there is a significant drop in the NVM loads performed by all the engines. We attribute this to caching of hot tuples in the CPU caches.

In the write-intensive workloads, we observe that the CoW engine now performs the most NVM stores. This is because it needs to copy several pages while creating the dirty directory. This engine also performs the largest number of load operations. The copying mechanism itself requires reading data off NVM. Further, the I/O overhead of maintaining this directory reduces the number of hot tuples that can reside in the CPU caches.

On the write-heavy workload, the NVM-aware engines perform 17–48% fewer stores compared to their traditional counterparts. We attribute this to their lightweight durability mechanisms and smaller storage footprints that enable them to make better use of hardware caches. Even with increased workload skew, the NVM-aware engines perform 9–41% fewer NVM writes. We note that the NVM accesses performed by the storage engines correlate inversely with the throughput delivered by these engines as shown in Section 5.2.

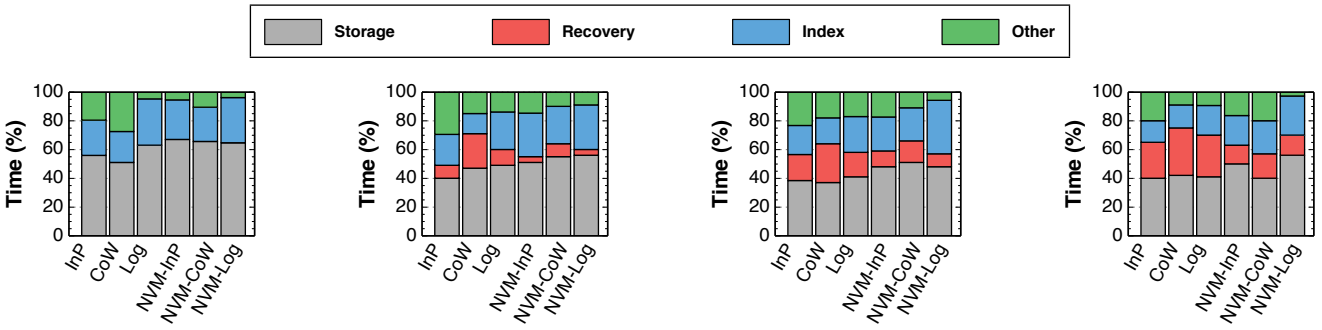
TPC-C: Fig. 11 presents the NVM accesses performed while executing the TPC-C benchmark. NVM-aware engines perform 31–42% fewer writes compared to the traditional engines. We see that the access patterns are similar to that observed with the write-intensive workload mixture in the YCSB benchmark. The Log engine performs more writes in this benchmark compared to the YCSB benchmark because it has more indexes. This means that updating a tuple requires updating several indexes as well.

5.4 Recovery

In this experiment, we evaluate the recovery latency of the storage engines. For each benchmark, we first execute a fixed number of transactions and then force a hard shutdown of the DBMS (`SIGKILL`). We then measure the amount of time for the system to restore the database to a consistent state. That is, a state where the effects of all committed transactions are durable, and the effects of uncommitted transactions are removed. The number of transactions that need to be recovered by the DBMS depends on the frequency of checkpointing for the InP engine and on the frequency of flushing the MemTable for the Log engine. The CoW and NVM-CoW engines do not perform any recovery mechanism after the OS or DBMS restarts because they never overwrite committed data. They have to perform garbage collection to clean up the previous dirty directory. This is done asynchronously and does not have a significant impact on the throughput of the DBMS.

YCSB: The results in Fig. 12a show the recovery measurements for the YCSB benchmark. We do not show the CoW and NVM-CoW engines as they never need to recover. We observe that the latency of the InP and Log engines grows linearly with the number of transactions that need to be recovered. This is because these engines first redo the effects of committed transactions before undoing the effects of uncommitted transactions. In contrast, the NVM-InP and NVM-Log engines' recovery time is independent of the number of transactions executed. These engines only need to undo the effects of transactions that are active at the time of failure and not the ones since the last checkpoint or flush. The NVM-aware engines therefore have a short recovery that is always less than a second.

TPC-C: The results for the TPC-C benchmark are shown in Fig. 12b. The recovery latency of the NVM-InP and NVM-Log engines is slightly higher than that in the YCSB benchmark because the



(a) Read-only Workload **(b)** Read-heavy Workload **(c)** Balanced Workload **(d)** Write-heavy Workload

Figure 13: Execution Time Breakdown – The time that the engines spend in their internal components when running the YCSB benchmark.

TPC-C transactions perform more operations. However, the latency is still independent of the number of transactions executed unlike the traditional engines because the NVM-aware engines ensure that the effects of committed transactions are persisted immediately.

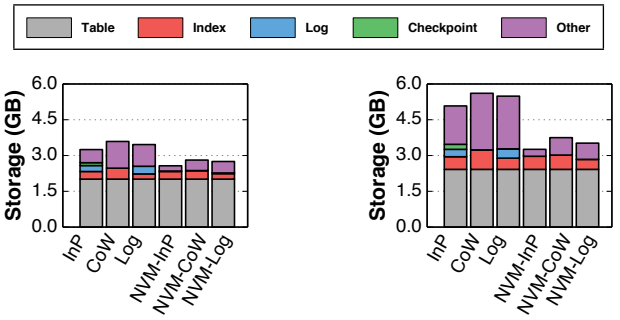
5.5 Execution Time Breakdown

In this experiment, we analyze the time that the engines spend in their internal components during execution. We only examine YCSB with low skew and low NVM latency configuration, which allows us to better understand the benefits and limitations of our implementations. We use event-based sampling with the *perf* framework [3] to track the cycles executed within the engine's components. We start this profiling after loading the initial database.

The engine's cycles are classified into four categories: (1) *storage* management operations with the allocator and filesystem interfaces, (2) *recovery* mechanisms like logging, (3) *index* accesses and maintenance, and (4) *other* miscellaneous components. This last category is different for each engine; it includes the time spent in synchronizing the engine's components and performing engine-specific tasks, such as compaction in case of the Log and NVM-Log engines. As our testbed uses a lightweight concurrency control mechanism, these results do not contain any overhead from locking or latching [59].

The most notable result for this experiment, as shown in Fig. 13, is that on the write-heavy workload, the NVM-aware engines only spend 13–18% of their time on recovery-related tasks compared to the traditional engines that spend as much as 33% of their time on them. We attribute this to the lower logging overhead in the case of the NVM-InP and NVM-Log engines, and the reduced cost of committing the dirty directory in the NVM-CoW engine. We observe that the proportion of the time that the engines spend on recovery mechanisms increases as the workload becomes write-intensive. This explains why the benefits of our optimizations are more prominent for the balanced and write-heavy workloads.

These results highlight the benefits of optimizing the memory allocator to leverage NVM's characteristics. This is because the NVM-aware engines spend most of their time performing storage management operations since their recovery mechanisms are so efficient. Interestingly, the engines performing copy-on-write updates spend a higher proportion of time on recovery-related tasks compared to other engines, particularly on the read-heavy workload. This highlights the cost of creating and maintaining the dirty directory for large databases, even using an efficient CoW B+tree. Another observation from Fig. 13 is that the Log and NVM-Log engines spend a higher fraction of their time accessing and maintaining indexes. This is because they perform multiple index look-ups on the LSM tree to reconstruct tuples. We observe that the NVM-Log engine spends less time performing the compaction process compared to the Log engine. This is due to the reduced overhead of maintaining the MemTables using the allocator interface.



(a) YCSB Storage **(b)** TPC-C Storage

Figure 14: Storage Footprint – The amount of storage occupied in NVM by the internal components of the engines.

5.6 Storage Footprint

Lastly, we compare the engines' usage of NVM storage at runtime. The storage footprint of an engine is the amount of space that it uses for storing tables, logs, indexes, and other internal data structures. This metric is important because we expect that the first NVM products will initially have a higher cost than current storage technologies [39]. For this experiment, we periodically collect statistics maintained by our allocator and the filesystem meta-data during the workload execution. This is done after loading the initial database for each benchmark. We then report the peak storage footprint of each engine. For all of the engines, we allow their background processes (e.g., group commit, checkpointing, garbage collection, compaction) to execute while we collect these measurements.

YCSB: We use the balanced workload mixture and low skew setting for this experiment. The initial size of the database is 2 GB. The results shown in Fig. 14a indicate that the CoW engine has the largest storage footprint. Since this workload contains transactions that modify the database and tuples are accessed more uniformly, this engine incurs high overhead from continually creating new dirty directories and copying tuples. The InP and Log engines rely on logging to improve their recovery latency at the expense of a larger storage footprint. The InP engine checkpoints have a high compression ratio and therefore consume less space.

The NVM-aware engines have smaller storage footprints compared to the traditional engines. This is because the NVM-InP and NVM-Log engines only record non-volatile pointers to tuples and non-inlined fields in the WAL. As such, they consume 17–21% less storage space than their traditional counterparts. For the CoW engine, its large storage footprint is due to duplicated data in its internal cache. In contrast, the NVM-CoW engine accesses the non-volatile copy-on-write B+tree directly using the allocator interface, and only records non-volatile tuple pointers in this tree and not entire tuples. This allows it to use 25% less storage space.

TPC-C: The graph in Fig. 14b shows the storage footprint of the engines while executing TPC-C. For this benchmark, the initial size of the database is 1 GB and it grows to 2.4 GB. Transactions inserting new tuples increase the size of the internal data structures in the CoW and Log engines (i.e., the copy-on-write B+trees and the SSTables stored in the filesystem). By avoiding unnecessary data duplication using NVM's persistence property, the NVM-aware engines have 31–38% smaller storage footprints. The space savings are more significant in this benchmark because the workload is write-intensive with longer running transactions. Thus, the logs in the InP and the Log engines grow more quickly compared to the small undo logs in their NVM-aware counterparts.

5.7 Discussion

Our analysis shows that the NVM access latency has the most impact on the runtime performance of the engines, more so than the amount of skew or the number of modifications to the database in the workload. This difference due to latency is more pronounced with the NVM-aware variants; their absolute throughput is better than the traditional engines, but longer latencies cause their performance to drop more significantly. This behavior is because they are no longer bottlenecked by heavyweight durability mechanisms [23].

The NVM-aware engines also perform fewer store operations, which will help extend NVM device lifetimes. We attribute this to the reduction in redundant data that the engines store when a transaction modifies the database. Using the allocator interface with non-volatile pointers for internal data structures also allows them to have a smaller storage footprint. This in turn avoids polluting the CPU's caches with unnecessary copying and transformation operations. It also improves the recovery times of the engines that use a WAL since they no longer record redo information.

Overall, we find that the NVM-InP engine performs the best across a wide set of workload mixtures and skew settings for all NVM latency configurations. The NVM-CoW engine did not perform as well for write-intensive workloads, but may be a better fit for DBMSs that support non-blocking read-only transactions. For the NVM-Log engine, many of its design assumptions are not copacetic for a single-tier storage hierarchy. The engine is essentially performing in-place updates like the NVM-InP engine but with additional overhead of maintaining its legacy components.

6. RELATED WORK

We now discuss previous research on using NVM. SafeRAM [21] is one of the first projects that explored the use of NVM in software applications. Using simulations, they evaluated the improvement in throughput and latency of OLTP workloads when disk is replaced by battery-backed DRAM. Later work demonstrated the utility of NVM in a client-side cache for distributed filesystems [8].

Recently, several application-level APIs for programming with persistent memory have been proposed. Mnemosyne [64] and NV-heaps [18] use software transactional memory to support transactional updates to data stored on NVM. NVMalloc [49] is a memory allocator that considers wear leveling of NVM. The primitives provided by these systems allow programmers to use NVM in their applications but do not provide the transactional semantics required by a DBMS.

In the context of DBMSs, recent work demonstrated that memory-oriented DBMSs perform only marginally better than disk-oriented DBMSs when using NVM because both systems still assume that memory is volatile [24]. Others have developed new recovery mechanisms for DBMSs that are using a NVM + DRAM storage hierarchy [53, 30, 17, 65]. Pelley et al. introduce a group commit mechanism to persist transactions' updates in batches to reduce the number of write barriers required for ensuring correct ordering [53]. This work is based on the Shore-MT engine [37], which means that the DBMS records page-level before-images in the undo logs before performing in-place updates. This results in high data duplication.

Wang et al. present a passive group commit method for a distributed logging protocol extension to Shore-MT [65]. Instead of issuing a barrier for every processor at commit time, the DBMS tracks when all of the records required to ensure the durability of a transaction are flushed to NVM. This approach is similar to other work on a log manager that directly writes log records to NVM, and addresses the problems of detecting partial writes and recoverability [28]. Both of these projects rely on software-based simulation of a NVM + DRAM system.

MARS [17] is an in-place updates engine optimized for NVM that relies on a hardware-assisted primitive that allows multiple writes to arbitrary locations to happen atomically. MARS does away with undo log records by keeping the changes isolated using this primitive. Similarly, it relies on this primitive to apply the redo log records at the time of commit. In comparison, our NVM-InP engine is based on non-volatile pointers, a software-based primitive. It removes the need to maintain redo information in the WAL, but still needs to maintain undo log records until the transaction commits.

SOFORT [51] is a hybrid storage engine designed for a storage hierarchy with both NVM and DRAM. The engine is designed to not perform any logging and uses MVCC. Similar to SOFORT, we make use of non-volatile pointers [58]. Our usage of persistent pointers is different. Their persistent pointer primitive is a combination of page ID and offset. We eschew the page abstraction in our engines, as NVM is byte-addressable. Hence, we use raw NVM pointers to build non-volatile data structures. For example, the engines performing in-place and log-structured updates need to perform undo logging, which is where we use non-volatile pointers to reduce data duplication.

Beyond DBMSs, others have looked into using NVM in filesystems. BPFS is a filesystem designed for persistent memory [19]. It uses a variant of shadow paging that supports atomic updates by relying on a special hardware instruction that ensures ordering between writes in different epochs. PMFS is another filesystem from Intel Labs that is designed for byte-addressable NVM [27]. It relies on write-ahead logging to preserve meta-data consistency and uses shadow paging only for data. It assumes a simpler hardware barrier primitive than epochs. Further, it optimizes memory-mapped I/O by directly mapping the persistent memory to the application's address space. The filesystem interface used by the traditional storage engines in our DBMS testbed is backed by PMFS.

7. CONCLUSION

This paper explored the fundamentals of storage and recovery methods in OLTP DBMSs running on an NVM-only storage hierarchy. We implemented three storage engines in a modular DBMS testbed with different architectures: (1) in-place updates, (2) copy-on-write updates, and (3) log-structured updates. We then developed optimized variants of each of these engines that better make use of NVM's characteristics. Our experimental analysis with two different OLTP workloads showed that our NVM-aware engines outperform the traditional engines by up to 5.5× while reducing the number of writes to the storage device by more than half on write-intensive workloads. We found that the NVM access latency has the most impact on the runtime performance of the engines, more so than the workload skew or the number of modifications to the database in the workload. Our evaluation showed that the NVM-aware in-place updates engine achieved the best throughput among all the engines with the least amount of wear on the NVM device.

8. REFERENCES

[1] Apache Cassandra. http://datastax.com/documentation/cassandra/2.0/.

[2] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[3] Linux perf framework. https://perf.wiki.kernel.org/index.php/Main_Page.

[4] NUMA policy library. http://linux.die.net/man/3/numa.

[5] VoltDB. http://voltdb.com.

[6] AGIGARAM. DDR3 NVDIMM. http://www.agigatech.com/ddr3.php.

[7] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[8] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *ASPLOS*, pages 10–22, 1992.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[10] T. Bingmann. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/.

[11] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan. The necessary death of the block device interface. In *CIDR*, 2013.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.

[13] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. Res. Dev.*, 52(4):449–464, July 2008.

[14] J. Chang, P. Ranganathan, T. Mudge, D. Roberts, M. A. Shah, and K. T. Lim. A limits study of benefits from nanostore-based future data-centric system architectures. CF '12, pages 33–42, 2012.

[15] F. Chen, M. Mesnier, and S. Hahn. A protected block device for persistent memory. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[16] H. Chu. MDB: A Memory-Mapped Database and Backend for OpenLDAP. Technical report, OpenLDAP, 2011.

[17] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, 2013.

[18] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, pages 105–118. ACM, 2011.

[19] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, pages 133–146, 2009.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[21] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. VLDB, pages 327–335. Morgan Kaufmann Publishers Inc., 1989.

[22] J. Dean and S. Ghemawat. LevelDB. http://leveldb.googlecode.com.

[23] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS@VLDB*, 2014.

[24] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, Sept. 2013.

[25] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[26] A. Driskill-Smith. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[27] S. R. Dulloor, S. K. Kumar, A. Keshavamurthy, P. Lantz, D. Subbareddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[28] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. ICDE, pages 1221–1231, 2011.

[29] M. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997.

[30] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: Reducing transaction logging overhead with pcm. CIKM, pages 2401–2404, 2011.

[31] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, pages 509–516, 1992.

[32] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS Fast Path. Technical report, Tandem, 1985.

[33] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system R database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[34] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. SIGMETRICS '03, pages 283–294, 2003.

[35] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[36] M. Hedenfalk. Copy-on-write B+ Tree. http://www.bzero.se/ldapd/.

[37] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[38] N. P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA, 1993.

[39] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[40] B. Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. Technical report, Tokutek, 2014.

[41] D. Laney. 3-D data management: Controlling data volume, velocity and variety. Feb. 2001.

[42] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36(12).

[43] LevelDB. Implementation details of LevelDB. `https://leveldb.googlecode.com/svn/trunk/doc/impl.html`.

[44] P. Macko. A simple PCM block device simulator for Linux. `https://code.google.com/p/pcmsim/people/list`.

[45] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[46] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM J. Res. Dev.*, 46(2-3).

[47] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[48] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. PACT '09, pages 261–270.

[49] I. Moraru, D. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, 2013.

[50] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[51] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. DaMoN, pages 8:1–8:7, 2014.

[52] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[53] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[54] T. Perez and C. Rose. Non-volatile memory: Emerging technologies and their impact on memory systems. *PURCS Technical Report*, 2010.

[55] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[56] O. Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, pages 2:1–2:27, 2008.

[57] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[58] A. Rudoff. Persistent memory library. `https://github.com/pmem/linux-examples/tree/master/libpmem`.

[59] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[60] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, (7191):80–83, 2008.

[61] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/`, June 2007.

[62] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[63] V. Viswanathan, K. Kumar, and T. Willhalm. Intel Memory Latency Checker. `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`.

[64] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 91–104. ACM, 2011.

[65] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[66] J. H. Yoon, H. C. Hunter, and G. A. Tressler. Flash & dram si scaling challenges, emerging non-volatile memory technology enablement - implications to enterprise storage and server compute systems. Flash Memory Summit, aug 2013.

APPENDIX

A. ANALYTICAL COST MODEL

In this section, we present a cost model to estimate the amount of data written to NVM per operation, by the traditional and NVM-optimized storage engines. This model highlights the strengths and weaknesses of these engines.

We begin the analysis by stating the assumptions we use to simplify the model. First, the database operations are presumed to be always successful. The amount of data written to NVM while performing an aborted operation will depend on the stage at which the operation fails. We therefore restrict our analysis to only successful operations. Second, the engines handle fixed-length and variable-length tuple fields differently. The fixed-length fields are stored in-line, while the variable-length fields are stored separately. To illustrate this difference, we assume that the update operation alters one fixed-length field and one variable-length field. Note that the tuple itself can contain any number of fixed-length and variable-length fields depending on the database schema.

Let us now describe some notation. We denote the size of the tuple by T. This depends on the specific table on which the engine performs the database operation. Let the size of the fixed-length field and the variable-length field altered by the update operation be F and V, respectively. These parameters depend on the table columns that are modified by the engine. The size of a pointer is represented by p. The NVM-optimized engines use non-volatile pointers to tuples and variable-length tuple fields to reduce data duplication. We use θ to denote the write-amplification factor of the engines performing log-structured updates. θ could be attributed to the periodic compaction mechanism that these engines perform to bound read-amplification and depends on the type of LSM tree. Let B represent the size of a node in the CoW B+tree used by the CoW and NVM-CoW engines. We indicate small fixed-length writes to NVM, such as those used to maintain the status of tuple slots, by ε.

Given this notation, we present the cost model in Table 3. The data written to NVM is classified into three categories: (1) memory, (2) log, and (3) table storage. We now describe some notable entries in the table. While performing an insert operation, the InP engine writes three physical copies of a tuple. In contrast, the NVM-InP engine only records the tuple pointer in the log and table data structures on NVM. In the case of the CoW and NVM-CoW engines, there are two possibilities depending on whether a copy of the relevant B+tree node is absent or present in the dirty directory. For the latter, the engines do not need to make a copy of the node before applying

	Insert		Update		Delete	
InP	Memory	: T	Memory	: F + V	Memory	: ε
	Log	: T	Log	: 2 × (F + V)	Log	: T
	Table	: T	Table	: F + V	Table	: ε
CoW	Memory	: B + T ‖ T	Memory	: B + F + V ‖ F + V	Memory	: B + ε ‖ ε
	Log	: 0	Log	: 0	Log	: 0
	Table	: B ‖ T	Table	: B ‖ F + V	Table	: B ‖ ε
Log	Memory	: T	Memory	: F + V	Memory	: ε
	Log	: T	Log	: 2 * (F + V)	Log	: T
	Table	: θ × T	Table	: θ × (F + V)	Table	: ε
NVM-InP	Memory	: T	Memory	: F + V + p	Memory	: ε
	Log	: p	Log	: F + p	Log	: p
	Table	: p	Table	: 0	Table	: ε
NVM-CoW	Memory	: T	Memory	: T + F + V	Memory	: ε
	Log	: 0	Log	: 0	Log	: 0
	Table	: B + p ‖ p	Table	: B + p ‖ p	Table	: B + ε ‖ ε
NVM-Log	Memory	: T	Memory	: F + V + p	Memory	: ε
	Log	: p	Log	: F + p	Log	: p
	Table	: θ × T	Table	: θ × (F + p)	Table	: ε

Table 3: Analytical cost model for estimating the amount of data written to NVM, while performing insert, update, and delete operations, by each engine.

the desired transformation. We distinguish these two cases in the relevant table entries using vertical bars. Note that these engines have no logging overhead as they always apply modifications in the dirty directory. The performance gap between the traditional and the NVM-optimized engines, particularly for write-intensive workloads, directly follows from the cost model presented in the table.

B. IMPACT OF B+TREE NODE SIZE

We examine the sensitivity of our experimental results to size of the B+tree nodes in this section. The engines performing in-place and log-structured updates use the STX B+tree [10] for maintaining indexes, while the engines performing copy-on-write updates use the append-only B+tree [56, 16, 36] for storing the directories. In all our experiments, we use the default node size for both the STX B+tree (512 B) and copy-on-write B+tree (4 KB) implementations. For this analysis, we vary the B+tree node size and examine the impact on the engine's throughput, while executing different YCSB workloads under low NVM latency (2×) and low workload skew settings. We restrict our analysis to the NVM-aware engines as they are representative of other engines.

The graphs, shown in Fig. 15, indicate that the impact of B+tree node size is more significant for the CoW B+tree than the STX B+tree. In case of the CoW B+tree, we observe that increasing the node size improves the engine's performance on read-heavy workloads. This can be attributed to smaller tree depth, which in turn reduces the amount of indirection in the data structure. It also reduces the amount of metadata that needs to be flushed to NVM to ensure recoverability. However, the engine's performance on write-heavy workloads drops as the B+tree nodes get larger. This is because of the copying overhead when performing updates in the dirty directory of the CoW B+tree. We found that the engines performing copy-on-write updates perform well on both types of workloads when the node size is 4 KB. With the STX B+tree, our experiments suggest that the optimal node size is 512 B. This setting provides a nice balance between cache misses, instructions executed, TLB misses, and space utilization [34]. Hence, in all of our experiments in Section 5, we configured the B+trees used by all the engines to their optimal performance settings.

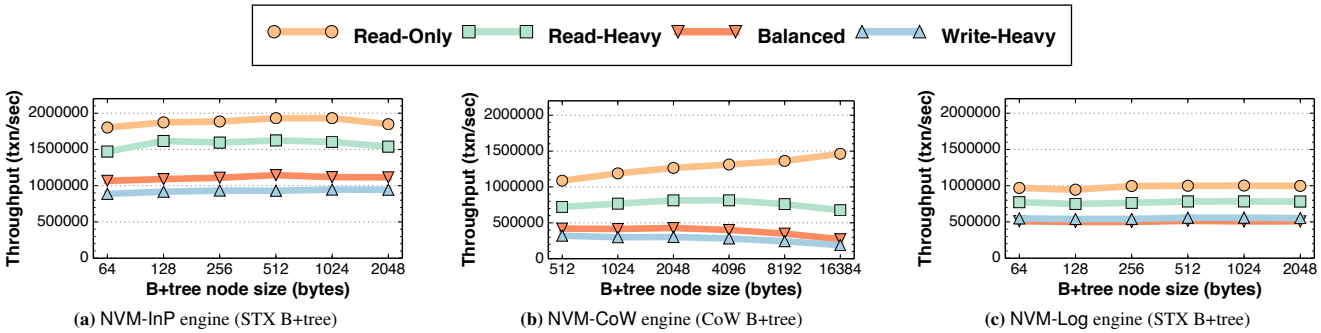
C. NVM INSTRUCTION SET EXTENSIONS

In this section, we explore the impact of newly proposed NVM-related instruction set extensions [2] on the performance of the engines. These extensions have been added by Intel in late 2014 and are not currently commercially available. As we describe in Section 2.3, we currently implement the *sync* primitive using the `SFENCE` and `CLFLUSH` instructions. We believe that these new extensions, such as the `PCOMMIT` and `CLWB` instructions [2], can be used to ensure the correctness and improve the performance of this primitive in future processors because they are more efficient and provide better control for how a DBMS interacts with NVM.

The `PCOMMIT` instruction guarantees the durability of stores to persistent memory. When the data is written back from the CPU caches, it can still reside in the volatile buffers on the memory controller. After the `PCOMMIT` instruction is executed, the store must become persistent. The `CLWB` instruction writes back a target cache line to NVM similar to the `CLFLUSH` instruction. It is, however, different in two ways: (1) it is a weakly-ordered instruction and can thus perform better than the strongly-ordered `CLFLUSH` instruction, and (2) it can retain a copy of the line in the cache hierarchy in exclusive state, thereby reducing the possibility of cache misses during subsequent accesses. In contrast, the `CLFLUSH` instruction always invalidates the cacheline, which means that data has to be retrieved again from NVM.

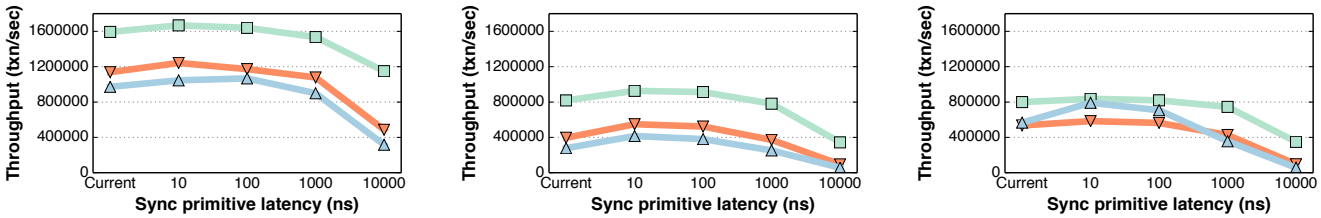
To understand the performance impact of the sync primitive comprising of `PCOMMIT` and `CLWB` instructions, we emulate its latency using `RDTSC` and `PAUSE` instructions. We note that our software-based latency emulation does not capture all the complex interactions in real processors. However, it still allows us to perform a useful *what-if* analysis before these instruction set extensions are available. We vary the latency of the sync primitive from 10–10000 ns and compare it with the currently used sync primitive. Since the traditional engines use PMFS [27], which is loaded in as a kernel module, they require more changes for this experiment. We therefore restrict our analysis to the NVM-aware engines. We execute different YCSB workloads under low NVM latency (2×) and low workload skew settings.

The results in Fig. 16 show that the engines are sensitive to the performance of the sync primitive. Performance measurements of the engines while using the current sync primitive are shown on the left side of each graph to serve as a baseline. We observe that the throughput of all the engines drops significantly with the increasing sync primitive latency. This is expected as these engines make extensive use of this primitive in their non-volatile data structures. The impact is therefore more pronounced on write-intensive workloads.



(a) NVM-InP engine (STX B+tree) **(b)** NVM-CoW engine (CoW B+tree) **(c)** NVM-Log engine (STX B+tree)

Figure 15: B+Tree Node Size – The impact of B+tree node size on the performance of the NVM-aware engines. The engines run the YCSB workloads under low NVM latency (2×) and low skew settings.



(a) NVM-InP engine **(b)** NVM-CoW engine **(c)** NVM-Log engine

Figure 16: NVM Instruction Set Extensions – The impact of *sync* primitive latency on the performance of the NVM-aware engines. The engines run the YCSB workloads under low NVM latency (2×) and low skew settings. Performance obtained using the current primitive, built using the `SFENCE` and `CLFLUSH` instructions, is shown on the left side of each graph to serve as a baseline.

We note that the NVM-CoW engine is slightly less sensitive to latency of the sync primitive than the NVM-InP and NVM-Log engines. We attribute this to the fact that this engine primarily uses data duplication to ensure recoverability and only uses the sync primitive to ensure the consistency of the CoW B+tree. In case of the NVM-Log engine, its performance while executing the write-heavy workload is interesting. Its throughput becomes less than the throughput on the balanced workload only when the latency of the sync primitive is above 1000 ns. This is because the engine needs to reconstruct tuples from entries spread across different LSM tree components.

We conclude that the trade-offs that we identified in these NVM-aware engines in the main body of the paper still hold at higher sync primitive latencies. Overall, we believe that these new instructions will be required to ensure recoverability and improve the performance of future NVM-aware DBMSs.

D. FUTURE WORK

A hybrid DRAM and NVM storage hierarchy is a viable alternative, particularly in case of high NVM latency technologies and analytical workloads. We plan to expand our NVM-aware engines to run on that storage hierarchy. There are several other aspects of DBMS architectures for NVM that we would like study further. We plan to investigate concurrency control algorithms that are more sophisticated than the scheme that we used in our testbed to see whether they can be made to scale on a NVM-based system. In particular, we are interested in exploring methods for supporting hybrid workloads (i.e., OLTP + OLAP) on NVM.

We will evaluate state-of-the-art index data structures to understand whether their trade-offs are appropriate for NVM (e.g., wear-leveling) and adapt them to exploit its persistence properties. We also anticipate the need for techniques to protect the contents of the database from errant code running and prevent the DBMS from corrupting the database by modifying a location in NVM that it should not.

E. ACKNOWLEDGEMENTS

This work was supported (in part) by the Intel Science and Technology Center for Big Data and the U.S. National Science Foundation (CCF-1438955). All of you need to recognize how Jignesh Patel be tearing through bits with QuickStep like he's slinging rocks at less than retail. Respect in the 608. Word is bond.