A Prolegomenon on OLTP Database Systems for Non-Volatile Memory

Justin DeBrabant
Brown University
debrabant@cs.brown.edu

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

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

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

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

ABSTRACT

The design of a database management system's (DBMS) architecture is predicated on the target storage hierarchy. Traditional disk-oriented systems use a two-level hierarchy, with fast volatile memory used for caching, and slower, durable device used for primary storage. As such, these systems use a buffer pool and complex concurrency control schemes to mask disk latencies. Compare this to main memory DBMSs that assume all data can reside in DRAM, and thus do not need these components.

But emerging non-volatile memory (NVM) technologies require us to rethink this dichotomy. Such memory devices are slightly slower than DRAM, but all writes are persistent, even after power loss. We explore two possible use cases of NVM for on-line transaction processing (OLTP) DBMSs. The first is where NVM completely replaces DRAM and the other is where NVM and DRAM coexist in the system. For each case, we compare the performance of a disk-oriented DBMS with a memory-oriented DBMS using two OLTP benchmarks. We also evaluate the performance of different recovery algorithms on these NVM devices. Our evaluation shows that in both storage hierarchies, memory-oriented systems are able to outperform their disk-oriented counterparts. However, as skew decreases the performance of the two architectures converge, showing that neither architecture is ideally suited for an NVM-based storage hierarchy.

1. INTRODUCTION

Disk-oriented DBMSs are based on the same hardware assumptions that were made in 1970s with the original relational DBMSs. The architectures of these systems use a two-level storage hierarchy: (1) a fast but volatile byte-addressable memory for caching (i.e., DRAM) and (2) a slow, non-volatile block-addressable device for permanent storage (i.e., HDD or SSD). These systems take a pessimistic assumption that a transaction could access data that is not in memory and thus will incur a long delay while a OS retrieves the data needed from disk. They employ a heavyweight concurrency control scheme that allows multiple transactions to safely run at the same time; when one transaction stalls because of the disk, another transaction can continue execution. This requires the use of buffer pools and complex transaction serialization schemes.

Recent advances in manufacturing technologies have greatly increased the capacity of DRAM available on a single computer. But disk-oriented systems were not designed for the case where most, if not all, of the data resides entirely in memory. The result is that many of their legacy components have been shown to impede their scalability for OLTP workloads [15]. In contrast, the architecture of main-memory DBMSs assume that all data fits in main memory and thus are able to remove the slower, disk-oriented components from the system. As such, they have been shown to outperform disk-oriented DBMSs [27]. TimesTen [2] was an early example of this approach, and newer systems include H-Store [16], Hekaton [10], and HyPer [17].

In some OLTP applications, however, the database can grow to be larger than the amount of memory available to the DBMS. Although one could partition a database across multiple machines so that the aggregate memory is sufficient, this can increase the number of multi-node transactions in some applications and thereby degrade their performance [23]. Recent work has explored adding back slower disk-based storage in main memory DBMSs as a place to store "cold" data, thereby freeing up in-memory storage [9, 26]. These techniques exploit the skewed access patterns of OLTP workloads to support databases that exceed the memory capacity of the DBMS while still providing the performance advantages of a memory-oriented system.

The advent of *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 [25] and memristors [28], that provide low latency reads and writes on the same order of magnitude as DRAM but with persistent writes like an SSD. Researchers also speculate that NVM will have much higher storage densities than what is possible with current DRAM devices. A comparison of performance characteristics of non-volatile memory technologies relative to DRAM and Flash is shown in Table 1. Such low-latency, high-capacity NVM storage has the potential to significantly change the design of DBMS architectures [6]. It is unclear, however, which DBMS architecture is ideal for NVM or whether a new architecture design is necessary.

Given this outlook, this paper presents our initial foray into the use of NVM in OLTP DBMSs. We test several DBMS architectures on an experimental, hardware-based NVM emulator and explore their trade-offs using two OLTP benchmarks. The read and

Proceedings of the VLDB Endowment, Vol. 7, No. 14
Copyright 2014 VLDB Endowment 2150-8097/14/10.

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

	DRAM	NAND Flash	Memristor	PCM
Byte-Addressable	Yes	No	Yes	Yes
Capacity	1×	4×	2-4×	2-4×
Latency	1×	400×	3-5×	3-5×
Endurance	10^{16}	10^4	10^6	10^6-10^8

Table 1: Comparison of performance characteristics of non-volatile memory technologies relative to DRAM and NAND Flash.

write latencies of the emulator are configurable, and thus we are able to evaluate multiple potential NVM profiles that are not specific to a particular technology. To the best of our knowledge, our investigation is the first to use emulated NVM for OLTP DBMSs.

Since it is unknown what future memory hierarchies will look like with NVM, we consider two potential use cases. The first is where the DBMS only has NVM storage with no DRAM. The second case is where NVM is added as another level of the storage hierarchy between DRAM and SSD. In both these configurations, the system still uses volatile CPU caches.

For both configurations, our results show that memory-oriented system outperforms traditional disk-oriented system on NVM. This difference is most pronounced when the skew in the workload is high, as this introduces significant lock contention in a disk-oriented system. However, the performance of memory-oriented system decreases as skew decreases while the performance of the disk-oriented system increases as skew decreases. This convergence is interesting, as it signifies that neither architecture is ideally suited for NVM. In Section 7, we discuss possible system architectures for NVM that leverage both memory-oriented and disk-oriented design features, thereby allowing uniform performance across all workload skews. In addition, we propose new possibilities for recovery schemes that take advantage of the persistence of NVM to provide nearly-instant recovery.

2. NVM HARDWARE EMULATOR

Evaluation of software systems using NVM is challenging due to lack of actual hardware. In this study, we use a NVM hardware emulator developed by Intel Lab. The emulator is implemented on a dual-socket Intel Xeon processor-based platform. Each processor has eight cores that run at 2.6 GHz and supports four DDR3 channels with two DIMMs per channel. The emulator's custom BIOS partitions the available DRAM memory into emulated NVM and regular (volatile) memory. Half of the memory channels on each processor are reserved for emulated NVM while the rest are used for regular memory. The emulated NVM is visible to the OS as a single NUMA node that interleaves memory (i.e., cache lines) across the two sockets. We are also able to configure the latency and bandwidth for the NVM partition by writing to CPU registers through the OS kernel. We use special CPU microcode and an emulation model for latency emulation where the amount of bandwidth throttling in the memory controller is programmable. We refer interested readers to [12] for more technical details on the emulator.

We divide the NVM partition into two sub-partitions. The first sub-partition is available to software as a NUMA node. The second sub-partition is managed by *PMFS*, a file system optimized for persistent memory [12]. Applications allocate and access memory in the first sub-partition using `libnuma` library or tools such as `numactl`. We refer to this interface provided by the emulator as the *NUMA interface*. Applications can also use regular POSIX file system interface to allocate and access memory in the second sub-partition through the *PMFS interface*. Since the memory bandwidth of our test systems executing the OLTP benchmarks is a small fraction of the available bandwidth, we do not modify the bandwidth throttling setting in our evaluation. Further, based on this observation, we postulate that OLTP systems are unlikely to be constrained by memory (particularly write) bandwidth. We now discuss the two emulator interfaces and how we utilize them.

2.1 NUMA Interface

We use the emulator's NUMA interface for evaluating the NVM-only DBMS architecture. The main benefit of this system configuration is that it allows us to evaluate DBMSs without making major modifications to the source code. All memory allocations for an application are assigned to the special NUMA node using `numactl`. Any read or write to memory are slowed down according to the emulator's latency setting. One potential drawback of this interface is that the DBMS's program code and OS data structures related to the DBMS's processes also reside in NVM. Furthermore, memory for other unrelated processes in the system could be allocated to the NUMA node. We did not observe this issue in our trials because of the default Linux memory policy that favors allocations from regular (volatile) memory nodes. In addition, the DBMS's program code is likely to be cached in the on-chip CPU caches, minimizing the overhead of fetching from the NVM.

2.2 PMFS Interface

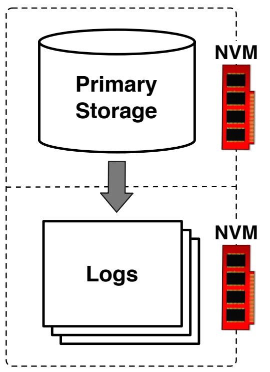
The emulator also supports a file system interface that allows us to deploy DBMSs using NVM with DRAM. Traditional file systems that operate at block granularity and in a layer above the block device abstraction are not best suited for fast, byte-addressable NVM. This is because the overhead of translating between two different address spaces (i.e., virtual addresses in memory and blocks in the block device) and maintaining a page cache in a traditional file system is significant. PMFS is a lightweight file system developed at Intel Labs that addresses this issue by completely avoiding page cache and block layer abstractions [12]. PMFS includes several optimizations for byte-addressable NVM that provide a significant performance improvement over traditional file systems (e.g., ext4). PMFS also allows applications to access NVM using memory-mapped I/O. We use the PMFS interface in the evaluation of both the NVM-only and NVM+DRAM architectures.

3. NVM-ONLY ARCHITECTURE

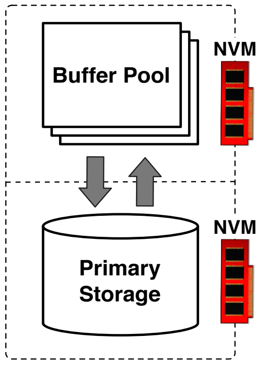
In the NVM-only architecture, the DBMS uses NVM exclusively for its storage. We compare a memory-oriented DBMS with a disk-oriented DBMS when both are running entirely on NVM storage using the emulator's NUMA interface. For the former, we use the H-Store DBMS [1], while for the latter we use MySQL (v5.5) with the InnoDB storage engine. Both systems are tuned according to their "best practice" guidelines for OLTP workloads.

The NVM-only architecture has implications for the DBMS's recovery scheme. In all DBMSs, some form of logging is used to guarantee recoverability in the event of a failure [14]. Disk-oriented DBMSs provide durability through the use of a write-ahead log, which is a type of *physical logging* wherein updated versions of data are logged to disk with each write operation. Such an approach has a significant performance overhead for main memory-oriented DBMSs [15, 19]. Thus, others have argued for the use of *logical logging* for main memory DBMSs where the log contains a record of the high-level operations that each transaction executed.

The overhead of writing out logical log records and the size of the log itself is much smaller for logical logging. The downside, however, is that the recovery process takes longer because the DBMS must re-execute each transaction to restore the database state. In contrast, during recovery in a physical logging system, the

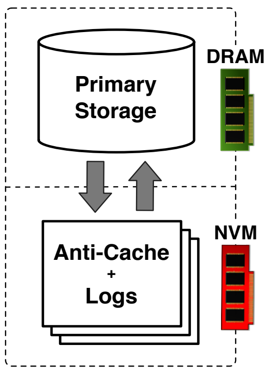


(a) Memory-oriented system

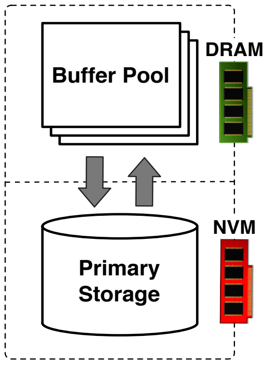


(b) Disk-oriented system

Figure 1: NVM-Only Architecture



(a) Anti-Caching system



(b) Disk-oriented system

Figure 2: NVM+DRAM Architecture

log is replayed forward to redo the effects of committed transactions and then replayed backwards to undo the effects of uncommitted transactions [20, 14]. But since all writes to memory are persistent under the NVM-only configuration, heavyweight logging protocols such as these are excessive and inefficient.

We now discuss the runtime operations of the memory-oriented and disk-oriented DBMSs that we evaluated on the NVM-only configuration in more detail (see Fig. 1). For each architecture, we analyze the potential complications and performance pitfalls from using NVM in the storage hierarchy.

3.1 Memory-oriented System

We use the emulator's NUMA interface to ensure that all of H-Store's in-memory data is stored on NVM. This data includes all tuples, indexes, and other database elements. We did not change any part of H-Store's storage manager or execution engine to use the byte-addressable NVM storage. But this means that the DBMS is not aware that writes to the memory are potentially durable.

Since H-Store was designed for DRAM, it employs a disk-oriented logical logging scheme [19]. To reduce recovery time, the DBMS also periodically takes a non-blocking checkpoint of all the partitions and writes them out to a disk-resident checkpoint. For our experiments in Section 5, we configured H-Store to write its checkpoints and log files to PMFS.

It is possible to use H-Store's logging scheme in the NVM-only architecture, but there is a trade-off between the DBMS's performance at runtime and the time it takes to recover the database after a crash. The issue is that there may be some transactions whose changes are durable in the NVM but will still be re-executed at recovery. This is because H-Store's storage manager writes all changes in place, and it does not have any way to determine that a change to the database has been completely flushed from its last-level cache to the NVM. Alternatively, the DBMS could periodically flush the volatile cache for a batch of transactions (i.e., group commit) to ensure that their changes are durable [24]. This feature, however, is currently not available in the emulator.

3.2 Disk-oriented System

In a disk-oriented DBMS, the system's internal data is divided into in-memory and disk-resident components. The DBMS maintains a buffer pool in memory to store copies of pages retrieved from the database's primary storage location on disk. We use the emulator's NUMA interface to store the DBMS's buffer pool in the byte-addressable NVM storage, while its data files and logs are stored in NVM through the PMFS interface.

Like with H-Store, MySQL is not aware that modifications to the buffer pool are persistent when using the NUMA interface. MySQL uses a *doublewrite* mechanism for flushing data to persistent storage. This involves first writing out the pages to a contiguous buffer on disk before writing them out to the data file. The doublewrite mechanism serves two purposes. First, it protects against torn writes that can occur when the DBMS has to atomically commit data that is larger the page size of the underlying storage device. Second, it also improves performance of (synchronous) logging as writes to the log buffer are sequential. This mechanism is not useful, however, in the NVM-only architecture where both the doublewrite buffer and the data file are on NVM. Since the doublewrite mechanism maintains multiple copies of each tuple, the DBMS unnecessarily wastes storage space in the NVM. The performance difference between random and sequential I/O on NVM is also much smaller than disk, thus batching the writes together in the doublewrite buffer does not provide the same gains as it does on a disk. Furthermore, the overhead of `fsync` in PMFS is also lower than in disk-oriented file systems.

4. NVM+DRAM ARCHITECTURE

In this configuration, the DBMS relies on both DRAM and NVM for satisfying its storage requirements. If we assume that the entire dataset cannot fit in DRAM, the question arises of how to split data between the two storage layers. Because of the relative latency advantage of DRAM over NVM, one strategy is to attempt to keep the hot data in DRAM and the cold data in NVM. One way is to use a buffer pool to cache hot data, as in traditional disk-oriented DBMSs. With this architecture, there are two copies of cached data, one persistent copy on disk and another copy cached in the DRAM-based buffer pool. The DBMS copies pages into the buffer pool as they are needed, and then writes out dirty pages to the NVM for durability. Another approach is to use the *anti-caching* system design proposed in [9] where all data initial resides in memory and then cold data is evicted out to disk over time. One key difference in this design is that exactly one copy of the data exists at any point in time. Thus, a tuple is either in memory or in the anti-cache. An overview of these two architectures is shown in Fig. 2.

4.1 Anti-Caching System

Anti-caching is a memory-oriented DBMS design that allows the system to manage databases that are larger than the amount of memory available without incurring the performance penalty of a disk-oriented system [9]. When the amount of in-memory data exceeds a user-defined threshold, the DBMS moves data to disk to free up space for new data. To do this, the system dynamically constructs blocks of the coldest tuples and writes them asynchronously to the anti-cache on disk. The DBMS maintains in-memory "tombstones" for each evicted tuple. When a running transaction attempts to access an evicted tuple through its tombstone, the DBMS aborts that transaction and fetches that it needs from the anti-cache without blocking other transactions. Once the data that the transaction needs is moved back into memory, the transaction is restarted.

For this study, we propose an extension of anti-caching where the cold data is stored in an NVM-optimized hash table rather than disk. We modify the cold data storage manager to adapt the anti-caching system to NVM. In the original implementation, we use BerkeleyDB [22] key-value store to manage anti-caching blocks. For NVM-backed data files, BerkeleyDB proved to be too heavyweight as we need finer-grained control over writes to NVM. To this end, we implemented a lightweight array-based block store using the emulator's PMFS interface. Elements of the array are anti-cache blocks and array indexes correspond to the anti-cache block id. If a block is transferred from the anti-cache to DRAM, the array index where the block was stored is added to a free list. When a new anti-cache block needs to be written, a vacant anti-cache block is acquired from the free list. We use a slab-based allocation method, where each time the anti-cache is full, a new slab is allocated and added to the free list. If the anti-cache shrinks, then the DBMS compacts and deallocates sparse slabs.

4.2 Disk-oriented System

We configure a disk-oriented DBMS to run on the NVM+DRAM architecture. We allow the buffer pool to remain in DRAM and store the data and log files using the PMFS interface. The main difference between this configuration and the the NVM-only MySQL configuration presented in Section 3.2 is that all main memory accesses in this configuration go to DRAM instead of NVM.

5. EXPERIMENTAL EVALUATION

To evaluate these different memory configuration and DBMS designs, we performed a series of experiments on the NVM emulator. We deployed four different system configurations: two executing entirely on NVM and two executing on a hybrid NVM+DRAM hierarchy. For the NVM-only analysis, we configured MySQL to execute entirely out of NVM and have compared it with H-Store configured to execute entirely in NVM. For the NVM+DRAM hierarchy analysis, we configured MySQL to use a DRAM-based buffer pool and store all persistent data in PMFS. As a comparison, we implemented the NVM adaptations to the anti-caching system described above by modifying the original H-Store based anti-caching implementation. We used two benchmarks in our evaluation and a range of different configuration parameters.

5.1 System Configuration

All experiments were conducted on the NVM emulator described in Section 2. For each system, we evaluate the benchmarks on two different NVM latencies: $2\times$ DRAM and $8\times$ DRAM, where the base DRAM latency is approximately 90 ns. We consider these latencies to represent the best case and worst case NVM latencies respectively [12]. We chose this range of latencies to make our results as independent from the underlying NVM technology as possible.

5.2 Benchmarks

We now describe the two benchmarks that we use for our evaluation. We use H-Store's internal benchmarking framework for both the H-Store on NVM and the anti-caching analysis. For the MySQL benchmarking, we use the OLTP-Bench [11] framework.

YCSB: The Yahoo! Cloud Services Benchmark (YCSB) is a workload that is representative of large-scale services provided by web-scale companies. It is a key-value store workload. We configure each tuple to consist of a unique key and 10 columns of random string data, each 100 bytes in size. Thus, the total size of a tuple is approximately 1KB. The workload used for this analysis consists of two transaction types, a `read` and an `update` transaction. The read randomly selects a key and reads a single tuple. The update randomly selects a key and updates all 10 non-key values for the tuple selected. The mix of read and update transactions in a workload is an important parameter in our analysis, as writes are much more costly, especially if data in the buffer pool must be kept consistent. We use three different workload mixtures:

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

In addition to the read-write mix, we also control the amount of skew that determines how often a tuple is accessed by transactions. We use YCSB's Zipfian distribution to model temporal skew in the workloads, meaning that newer items are accessed much more frequently than older items. The amount of skew is controlled by the Zipfian constant $s > 0$, where higher values of s generate higher skewed workloads. We pick values of s in the range of 0.5 to 1.5, which is representative of a range of skewed workloads.

TPC-C: This benchmark is an industry standard for evaluating the performance of OLTP systems [29]. The benchmark simulates an order-processing application, and consists of nine tables and five different transaction types. Only two of the transaction types modify tuples, but they make up 88% of a TPC-C workload. We use 100 warehouses and 100,000 items, resulting in a total data size of 10GB. For simplicity, we have configured transactions to only access data from a single warehouse. Thus, all transactions are single-sited (i.e., there are no distributed transactions) because warehouses are mapped to partitions. For the anti-cache trials, we evict data from the `HISTORY`, `ORDERS`, and `ORDER_LINE` tables, as these are the only tables where transactions insert new data.

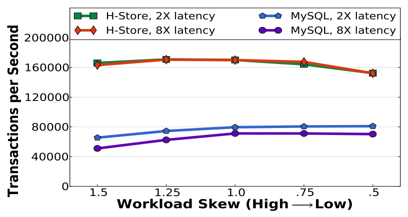
5.3 Results

We will now discuss the results of executing the two benchmarks, YCSB and TPC-C, on each of the NVM-only and NVM-DRAM architectures described in Sections 3 and 4.

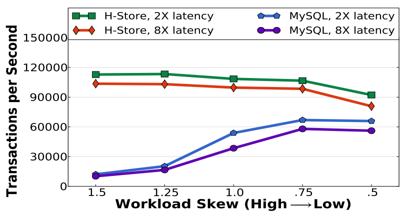
5.3.1 NVM-Only Architecture

YCSB: We evaluate YCSB on each system across the range of skew parameters and workload mixtures described above. We first consider the impact of NVM latency on the throughput of memory-oriented and disk-oriented systems. The results for the read-heavy workload shown in Fig. 3b indicate that increasing NVM latency decreases throughput of H-Store and MySQL by 12.3% and 14.8% respectively. There is no significant impact on H-Store's performance in the read-only workload shown in Fig. 3a, which indicates that latency mainly impacts the performance of logging.

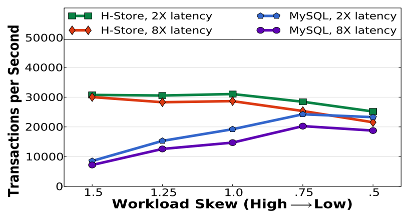
The throughput of these systems vary with the amount of skew in the workload. The impact of skew on H-Store's performance is more pronounced in the read-heavy workload shown in Fig. 3b. Throughput drops by 18.2% in the read-heavy workload as the skew level is reduced. The drop in throughput is due to the application's larger working set size, which increases the number of cache misses and subsequent accesses to NVM. In contrast, MySQL performs poorly on high-skew workloads but its throughput improves



(a) Read-Only

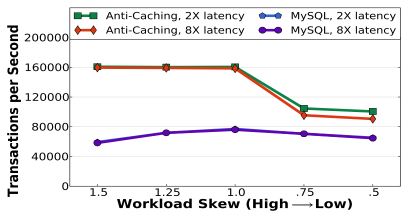


(b) Read-Heavy

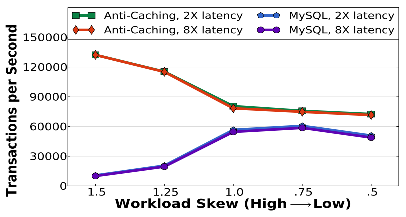


(c) Write-Heavy

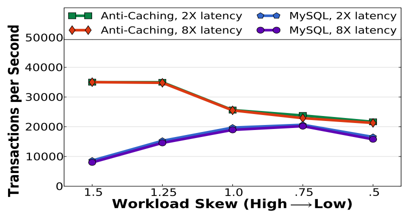
Figure 3: NVM-only Architecture – YCSB.



(a) Read-Only $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$



(b) Read-Heavy $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$



(c) Write-Heavy $\left(\frac{\text{data\_size}}{\text{mem\_size}} = 8\right)$

Figure 4: NVM+DRAM Architecture – YCSB.

by 5× as skew decreases. This is because a disk-oriented system uses locks to allow transactions to execute concurrently. Thus, if a large portion of the transactions access the same tuples, then lock contention becomes a bottleneck.

We can summarize the above observations as follows: (1) increasing NVM latency mainly impacts the performance of the logging mechanism, and (2) the throughput of memory-oriented and disk-oriented systems vary differently as skew decreases. We contend that the ideal system for a NVM-only architecture will possess features of both memory-oriented and disk-oriented systems.

TPC-C: For the TPC-C benchmark, most transactions insert or access new records (i.e., `NewOrder`), and older records are almost never accessed. As such, there is strong temporal skew built into the semantics of the benchmark. Only a subset of the tables are actually increasing in size, and the rest are static. In Fig. 5a, we see that throughput of both systems only varies slightly with an increase in NVM latency, and that for both latencies the throughput of H-Store is 10× higher than that of the disk-oriented system.

5.3.2 *NVM+DRAM Architecture*

YCSB: We use the same YCSB skew and workload mixes, but configure the amount of DRAM available to the DBMSs to be $\frac{1}{8}$ of the total database size. There are several conclusions to draw from the results shown in Fig. 4. The first is that the throughput of the two systems trend differently as skew changes. For the read-heavy workload in Fig. 4b, anti-caching achieves 13× higher throughput over MySQL when skew is high, but only a 1.3× improvement when skew is low. Other workload mixes have similar trends. This is because the anti-caching system performs best when there is high skew since it needs to fetch fewer blocks and restart fewer transactions. In contrast, the disk-oriented system performs worse on the high skew workloads due to high lock contention. We note that at the lowest skew level, MySQL's throughput decreases due to lower hit rates for data in the CPU's caches.

Another notable finding is that both systems do not exhibit a major change in performance with longer NVM latencies. This is significant, as it implies that neither architecture is bottlenecked by the I/O on the NVM. Instead, the decrease in performance is due to the overhead of fetching and evicting data from NVM. For the disk-oriented system, this overhead comes from managing the buffer pool, while in the anti-caching system it is from restarting transactions and asynchronously fetching previously evicted data.

We can summarize the above observations as follows: (1) the throughput of the anti-caching system decreases as skew decreases, (2) the throughput of the disk-oriented system increases as skew decreases, and (3) neither architecture is bottlenecked by I/O when the latency of NVM is between 2-8× the latency of DRAM. Given these results, we believe that the ideal system architecture for a NVM+DRAM memory hierarchy would need to posses features of both anti-caching and disk-oriented systems to enable it to achieve high throughput regardless of skew.

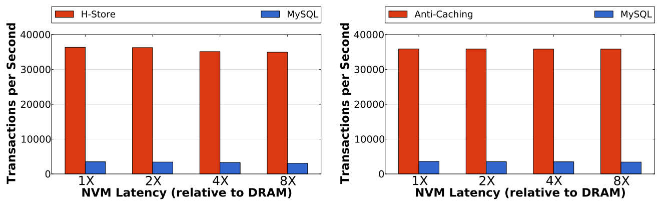
TPC-C: We next ran the TPC-C benchmark on the anti-caching and disk-oriented DBMSs using different NVM latencies. The results in Fig. 5b show that the throughput of both DBMSs do not change significantly as NVM latency increases. This is expected, since all of the transactions' write operations are initially stored on DRAM. These results corroborate previous studies that have shown the 10× performance advantage of an anti-caching system over the disk-oriented DBMS [9]. For the anti-caching system, this workload essentially measures how efficiently it is able to evict data to PMFS (since no transaction reads old data).

5.3.3 *Recovery*

Lastly, we evaluate recovery schemes in H-Store using the emulator's NUMA interface. We implemented logical logging (i.e., command logging) and physical logging (i.e., ARIES) recovery schemes within H-Store. For each scheme, we first measure the DBMS's runtime performance when executing a fixed number of TPC-C transactions (50,000). We then simulate a system failure and then measure how long it takes the DBMS to recover the database state from each scheme's corresponding log stored on PMFS.

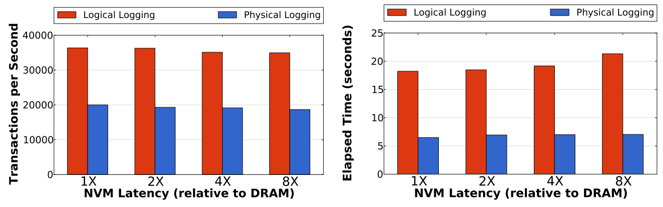
For the runtime measurements, the results in Fig. 6a show that H-Store achieves 2× higher throughput when using logical logging compared to physical logging. This is because logical logging only records the executed commands and thus is more lightweight. The amount of logging data for the workload using scheme is only 5MB. In contrast, physical logging keeps track of all modifications made at tuple-level granularity and its corresponding log 220MB. This reduced footprint makes logical logging more attractive for the first NVM devices that are expected to have limited capacities.

Next, in the results for the recovery times, Fig. 6b shows that logical logging 3× is slower than physical logging. One could reduce



(a) NVM-only Architecture **(b) NVM+DRAM Architecture**

Figure 5: NVM Latency Evaluation – Performance comparison for the TPC-C benchmark using different NVM latencies.



(a) Throughput **(b) Recovery Time**

Figure 6: Recovery Evaluation – Comparison of recovery schemes in H-Store using the TPC-C benchmark.

this time in logical logging by having the DBMS checkpoint more frequently, but this will impact steady-state performance [19].

We note that both schemes are essentially doing unnecessary work, since all writes to memory when using the NUMA interface are potentially durable. A better approach is to use a recovery scheme that is designed for NVM. This would allow a DBMS to combine the faster runtime performance of logical logging with the faster recovery of physical logging.

6. RELATED WORK

Pelley et al. [24] examine a novel recovery mechanism for OLTP systems running on NVM. The mechanism is based on a group commit protocol that persists the effects of transactions in batches to reduce the number of write barriers required for ensuring correct ordering. It relies on a volatile staging buffer to hold the effects of the ongoing batch. This is better suited for NVM than traditional ARIES-style recovery mechanisms that are designed to handle I/O delays of high-latency storage devices. Fang et. al. [13] designed a log manager that directly writes log records to NVM and addresses the problems of detecting partial writes and recoverability. This prior work relies on software-based emulation that can only emulate a NVM+DRAM system. In contrast, we use a hardware-based emulator that supports an NVM-only configuration.

Many systems have been proposed to simplify the usage of NVM in applications by providing an interface for programming with persistent memory. Mnemosyne [30] and NV-heaps [7] use software transactional memory to support transactional semantics along with word-level and object-level persistence. NVMalloc [21] is a memory allocator that considers wear leveling of NVM. Although the primitives provided by these systems allow programmers to use NVM with their applications, they are orthogonal to providing transactional semantics required by a DBMS.

Copeland et al. [8] predict that both latency and throughput can be improved by leveraging battery-backed DRAM for supporting transaction processing applications. Baker et al. [5] evaluate the utility of battery-backed DRAM in distributed file systems as a client-side file cache to reduce write traffic to file servers. Although these studies provide insights on the implications of NVM properties on storage systems, they rely on modeling and trace-driven simulation rather than direct evaluation on hardware. Bailey et al. [4] explore the impact of NVM devices on different OS components like virtual memory and file systems. Badam [3] gives an overview of the potential impact of NVM on different storage technologies and software systems. Kim et al. [18] propose an in-memory filesystem for non-volatile memory akin to PMFS.

7. FUTURE WORK

We now discuss two avenues of future work that we are actively pursuing. The first is a new OLTP DBMS that is designed to achieve nearly instant recovery for the NVM-only architecture. The second is an optimized variant of the anti-caching system for the NVM+DRAM architecture.

7.1 N-Store

The results in Fig. 3 show that the performance of the memory-oriented architecture converges with that of the disk-oriented system in workloads that involve writes, especially as skew decreases. We attribute this to the overhead of logging and diminishing benefits from H-Store's concurrency control scheme that is optimized for main memory [27]. In addition, in the recovery experiments shown in Fig. 6b, we observe that recovery latency is high for logical logging. We can reduce this overhead by checkpointing more frequently, but this also degrades performance [19]. Based on these constraints, we recognize the need to design a new DBMS that is specifically designed for NVM. The recoverability and concurrency control mechanisms of this system will leverage the persistence properties of NVM. We envision that this new DBMS, dubbed *N-Store*, will be a hybrid architecture that borrows ideas from both memory-oriented and disk-oriented systems.

7.2 Anti-Caching with Synchronous Fetches

As our results in Fig. 4 show, the anti-caching system outperforms the disk-oriented architecture across all skew levels, but that its performance advantage decreases as skew decreases. This is due to the overhead of aborting and restarting transactions when the DBMS must fetch evicted data. The performing this retrieval synchronously is prohibitively high in disk-based storage. Thus, the cost of restarting the transaction once the block is fetched is justified. However, if we replace disk with NVM, then the cost of fetching a block is significantly less. This means that it may be better to just stall the transaction and retrieve the data that it needs immediately rather than aborting the transaction and restarting it after the data that it needs is moved into memory. We plan to explore this variation of the anti-caching mechanism using the same principles to control data allocation and movement in NVM.

8. ACKNOWLEDGEMENTS

This research was funded (in part) by the Intel Science and Technology Center for Big Data. We also want to give a quick shout out to the HyPer team. We think that your research is dope.

9. CONCLUSION

In this paper, we explored two possible architectures using non-volatile memory (i.e., NVM-only and NVM+DRAM architectures). For each architecture, we evaluated memory-oriented and disk-oriented OLTP DBMSs. Our analysis shows that memory-oriented systems are better-suited to take advantage of NVM and outperform their disk-oriented counterparts. However, in both the NVM-only and NVM+DRAM architectures, the throughput of the memory-oriented systems decreases as workload skew is decreased while the throughput of the disk-oriented architectures increases as workload skew is decreased. Because of this, we conclude that neither system is ideally suited for NVM. Instead, a new system is needed with principles of both disk-oriented and memory-oriented systems and a lightweight recovery scheme designed to utilize the non-volatile property of NVM.

10. REFERENCES

[1] H-Store. `http://hstore.cs.brown.edu`.

[2] Oracle TimesTen Products and Technologies. Technical report, February 2007.

[3] A. Badam. How persistent memory will change software systems. *Computer*, 46(8):45–51, 2013.

[4] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *HotOS*, pages 2–2, 2011.

[5] 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.

[6] 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.

[7] 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 R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 105–118. ACM, 2011.

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

[9] 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.

[10] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[11] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[12] 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.

[13] 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.

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

[15] 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.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[17] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE, pages 195–206, 2011.

[18] H. Kim, J. Ahn, S. Ryu, J. Choi, and H. Han. In-memory file system for non-volatile memory. In *RACS*, pages 479–484, 2013.

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

[20] 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.

[21] I. Moraru, G. A. David, K. Michael, T. Niraj, R. Parthasarathy, and B. Nathan. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, 2013.

[22] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

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

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

[25] 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.

[26] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *DaMon*, 2013.

[27] 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.

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

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

[30] 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.