Storage Management in the NVRAM Era

Steven Pelley
University of Michigan
spelley@umich.edu

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

Brian T. Gold
Oracle Corporation
brian.t.gold@gmail.com

Bill Bridge
Oracle Corporation
bill.bridge@oracle.com

ABSTRACT

Emerging nonvolatile memory technologies (NVRAM) offer an alternative to disk that is persistent, provides read latency similar to DRAM, and is byte-addressable. Such NVRAMs could revolutionize online transaction processing (OLTP), which today must employ sophisticated optimizations with substantial software overheads to overcome the long latency and poor random access performance of disk. Nevertheless, many candidate NVRAM technologies exhibit their own limitations, such as greater-than-DRAM latency, particularly for writes.

In this paper, we reconsider OLTP durability management to optimize recovery performance and forward-processing throughput for emerging NVRAMs. First, we demonstrate that using NVRAM as a drop-in replacement for disk allows near-instantaneous recovery, but software complexity necessary for disk (i.e., Write Ahead Logging/ARIES) limits transaction throughput. Next, we consider the possibility of removing software-managed DRAM buffering. Finally, we measure the cost of ordering writes to NVRAM, which is vital for correct recovery. We consider three recovery mechanisms: *NVRAM Disk-Replacement*, *In-Place Updates* (transactions persist data in-place), and *NVRAM Group Commit* (transactions commit/persist atomically in batches). Whereas *In-Place Updates* offers the simplest design, it introduces persist synchronizations at every page update. *NVRAM Group Commit* minimizes persist synchronization, offering up to a 50% throughput improvement for large synchronous persist latencies.

1. INTRODUCTION

For decades disk has been the primary technology for durable storage. Although inexpensive and dense, disk provides high performance only for coarse-grained sequential access and suffers enormous slowdowns for random reads and writes. Flash-based solid-state disks greatly reduce coarse-grain random access latency, but retain a block-grain interface and still suffer from poor write performance; past work suggests that effective data management strategies remain similar for rotating disk and Flash [20]. Recently, Nonvolatile Random Access Memories (NVRAM), such as phase change memory and spin-transfer torque RAM, have emerged as viable storage alternatives [5]. Whereas the details of these technologies vary, they share the ability to write persistently with low latency at fine granularity. However, many candidate technologies share limitations, such as limited write endurance and high write latency [15].

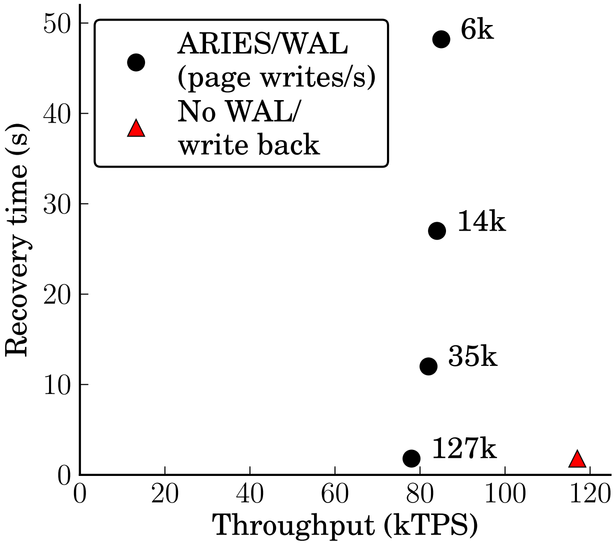


Figure 1: TPCB recovery latency vs throughput. Increasing page flush rate reduces recovery latency. Removing WAL entirely improves throughput by 50%.

These NVRAM technologies stand to revolutionize Online Transaction Processing (OLTP), where consistency and durability are paramount, but applications demand high throughput and low latency. Prior work has already demonstrated the potential of these technologies to enhance file systems [10] and persistent data structures [27], but has not considered OLTP. Today, OLTP systems are designed from the ground up to circumvent disk's performance limitations. For example, many popular database systems use Write-Ahead Logging (WAL; e.g., ARIES [17]) to avoid expensive random disk writes by instead writing to a sequential log. Although effective at hiding write latency, WAL entails substantial software overheads.

Proceedings of the VLDB Endowment, Vol. 7, No. 2

NVRAM offers an opportunity to simultaneously improve database forward-processing throughput and recovery latency by rethinking mechanisms that were designed to address the limitations of disk. Figure 1 demonstrates this potential, displaying recovery time and transaction throughput for the TPCB workload running on the Shore-MT storage manager [13] for hypothetical NVRAM devices (see Section 4 for a description of our methodology).

The ARIES/WAL points (black circles) in the Figure show forward-processing throughput (horizontal axis) and recovery time (vertical axis) as a function of device write throughput (annotated alongside each point). As database throughput can greatly outpace existing storage devices (our configuration requires 6,000 page writes/s to bound recovery; measured disk and flash devices provide only 190 and 2,500 page writes/s, respectively) we model recovery performance under faster NVRAM using a RAM disk for log and heap while limiting the page flush rate. As intuition would suggest, greater write bandwidth enables more aggressive flushing, minimizing the number of dirtied pages in the buffer cache at the time of failure, reducing recovery time. With enough write bandwidth (in this case, 127,000 flushes/s, or 0.97 GB/s random writes for 8KB pages) the database recovers near-instantly, but forward-processing performance remains compute bound. Achieving such throughput today requires large, expensive disk arrays or enterprise flash storage devices; future NVRAM devices might enable similar performance on commodity systems.

NVRAM opens up even more exciting opportunities for recovery management if we consider re-architecting database software. Figure 1 shows this additional potential with a design point (red triangle) that removes WAL and asynchronous page flushing—optimizations primarily designed to hide disk latency. Throughput improves due to three effects: (1) threads previously occupied by page and log flushers become available to serve additional transactions, (2) asynchronous page flushing, which interferes with transactions as both flusher and transaction threads latch frequently accessed pages, is removed, and (3) transactions no longer insert WAL log entries, reducing the transaction code path. In aggregate these simplifications amount to a 50% throughput increase over ARIES's best possible NVRAM performance. The key take-away is that database optimizations long used for disk only hinder performance with faster devices. In this paper, we investigate how to redesign durable storage and recovery management for OLTP to take advantage of the low latency and byte-addressability of NVRAM.

NVRAMs, however, are not without their limitations. Several candidate NVRAM technologies exhibit larger read latency and significantly larger write latency compared to DRAM. Additionally, whereas DRAM writes benefit from caching and typically are not on applications' critical paths, NVRAM writes must become persistent in a constrained order to ensure correct recovery. We consider an NVRAM access model where correct ordering of persistent writes is enforced via *persist barriers*, which stall until preceding NVRAM writes are complete; such persist barriers can introduce substantial delays when NVRAM writes are slow. Given the possible NVRAM technologies and their disparate performance characteristics, we propose a range of new NVRAM-specific database designs, evaluating their sensitivity to NVRAM characteristics. We contribute:

- a methodology for analyzing NVRAM read and write performance on existing hardware platforms, utilizing memory trace analysis, code annotation, and precise timing models.
- an analysis of NVRAM read latency's effect on database throughput, quantifying the impact of caching.
- a comparison of ARIES, as *NVRAM Disk-Replacement*, against *In-Place Updates*, wherein the data heap updates synchronously as in in-memory databases.
- a new recovery mechanism, *NVRAM Group Commit*, providing fast recovery and high throughput even in the presence of high-latency persist barriers.

We draw several conclusions. First, even though NVRAM read latency may be slower than DRAM, for the likely latency range, even a small amount of buffer cache (10s of pages) is sufficient to hide the higher latency. Second, when average persist barrier latency is low (below 1µs), *In-Place Updates* performs substantially better than *NVRAM Disk-Replacement*, by as much as 50% greater throughput in the best case. Finally, when average persist barrier latency increases beyond 1µs, *NVRAM Group Commit* retains much of the throughput advantage of *In-Place Updates*, continuing to outperform *NVRAM Disk-Replacement*, while providing a user-controllable latency trade-off.

Our paper is organized as follows. Section 2 provides an overview of NVRAM technologies and ARIES recovery management. Section 3 describes the possible design space of OLTP software leveraging NVRAM. In Section 4 we describe our experimental methodology. Sections 5 and 6 evaluate the performance of OLTP on NVRAM considering read and write performance concerns, respectively. We discuss related work in Section 7. Finally, in Section 8 we conclude.

2. BACKGROUND

This section provides an overview of anticipated NVRAM technologies and database recovery mechanisms.

2.1 Nonvolatile Memories

Nonvolatile memories will soon be commonplace. Technology trends suggest that DRAM and flash memory may cease to scale, requiring new dense memory technologies [15].

Memory technology characteristics. Numerous technologies offer durable byte-addressable access. Examples include phase change memory (PCM), where a chalcogenide glass is heated to produce varying electrical conductivities, and spin-transfer torque memory (STT-RAM), a magnetic memory that stores state in electron spin [5]. Storage capacity increases by storing more than two states per cell in Multi-level Cells (MLC) (e.g., four distinct resistivity levels provide storage of 2 bits per cell).

While it remains unclear which of these technologies will eventually dominate, many share common characteristics. In particular, NVRAMs will likely provide somewhat higher access latency relative to DRAM. Furthermore, several technologies are expected to have asymmetric read-write latencies, where writing to the device may take several microseconds [22]. Write latency worsens with MLC, where slow, iterative writes are necessary to reliably write to a cell.

Resistive NVRAM technologies suffer from limited write endurance; cells may be written reliably a limited number of times. While write endurance is an important consideration, proposed hardware mechanisms (e.g., Start-Gap [21]) are effective in distributing writes across cells, mitigating write endurance concerns. We do not evaluate NVRAM endurance-related concerns in this paper.

NVRAM storage architectures. Future database systems may incorporate NVRAM in a variety of ways. At one extreme, NVRAM can be deployed as a disk or flash SSD replacement. While safe, cost-effective, and backwards compatible, the traditional disk interface imposes overheads. Prior work demonstrates that file system and disk controller latencies dominate NVRAM access times [6]. Furthermore, block access negates advantages of byte addressability.

Recent research proposes alternative device interfaces for NVRAM. Caulfield *et al.* propose Moneta and Moneta Direct, a PCIe attached PCM device [7]. Unlike disk, Moneta Direct bypasses expensive system software and disk controller hardware to minimize access latency while still providing traditional file system semantics. However, Moneta retains a block interface. Condit *et al.* suggest that NVRAM connect directly to the memory bus, with additional hardware and software mechanisms providing file system access and consistency [10]. We adopt the same atomic eight-byte persistent write, enabling small, safe writes even in the face of failure. While exposing NVRAM as a byte-addressable memory raises interesting questions regarding access protection and memory allocation, we assume the operating system and hardware provide the necessary functionality. NVRAM will eventually connect via a memory interface, but it is unclear how NVRAM storage will evolve or what its exact performance characteristics will be.

Instead of focusing on device interface, we investigate the costs of providing durable consistency. We consider two aspects of the storage architecture: enforcing the order in which data persistently writes to the device, and notifying the user that their data are durable (e.g., to commit a transaction), which we collectively call *persist barriers* (similar to *epoch barriers* in [10]). We introduce the term *persist*, meaning "make persistent," to distinguish NVRAM device writes from volatile DRAM writes. Disk access achieves consistency through per-write acknowledgements—the user enforces the order of persists by receiving an acknowledgement or blocking before subsequent writes issue. With NVRAM, we expect such synchronization to incur latencies up to several microseconds. Accessing several memory chips in parallel will enable high access bandwidth (Flash SSD devices currently provide up to 1.5 GB/s), suggesting that persist order constraints will be the predominant performance bottleneck for many applications. Ordering writes in hardware presents opportunities for improved performance, but with some remaining cost, particularly to flush data when transactions commit. We do not consider a specific persist barrier implementation. Rather, in this paper we investigate how persist barrier stall latency affects optimal database design and throughput. Next, we describe recovery mechanisms.

2.2 Database Recovery

Transactions execute with various guarantees on isolation and durability. While relaxing these constraints often improves transaction performance, we focus on providing at least read committed isolation with synchronous durable commit. Atomic and durable transactions commit or abort seemingly instantaneously, despite failures—no transaction may be observed in a partially executed state, or may be observed at all if that transaction eventually aborts. While concurrency control plays a large role in ensuring these guarantees, separate mechanisms provide recovery to maintain data integrity and consistency to withstand failures. We describe ARIES [17], a popular Write Ahead Logging (WAL) system that provides atomic durable transactions.

ARIES. ARIES uses a two-level store (disk and volatile buffer cache) alongside a centralized log. Transactions' writes coalesce in the buffer cache while persisting to the log, transforming random accesses into sequential accesses. Whereas such logging is a necessary optimization when using disk, it is also a proven technique for providing durable consistency. The log contains both redo and undo entries for each update. Redo logs record actions performed on heap pages so that they can be replayed if data has not yet persisted in-place. Undo logs provide roll back operations necessary to remove aborted and uncommitted transaction updates during recovery. Recovery begins at the latest checkpoint, taken periodically and marked in the log. The redo log replays to its end, reproducing the state at failure in the buffer cache. Afterwards, incomplete transactions are removed using the appropriate undo log entries. Undo log entries contain a transaction number and refer to the previous entry generated from the same transaction.

While a centralized log orders all database updates, the software additionally enforces that the log persists before the heap page for every operation. Transactions commit by generating a commit log entry, which must necessarily persist after the transaction's other log entries. This process guarantees that no transaction commits, or page persists, without a durable history of its modifications in the log.

Though complex, ARIES improves database performance with disks. First, log writes appear as sequential accesses to disk, maximizing device throughput. Additionally, aside from reads resulting from buffer cache misses, each transaction depends on device access only at commit to flush log entries. All disk writes to the heap may be done at a later time, off of transactions' critical paths. In this way ARIES is designed from the ground up to minimize the effect of large disk access latencies.

ARIES, and more generally, WAL, provides a number of features aside from failure recovery. The log contains a complete redo-history of the database, and in conjunction with database snapshots allows arbitrary point-in-time recovery of the database. Further, logs support profiling and debugging for database applications. Finally, database replication and hot-standby commonly rely on communicating logs as a means to keep two instances of the same database consistent. Although we recognize their importance, we believe it is worth evaluating the performance cost WAL incurs for these features and consider removing WAL entirely.

Whereas disk relies on ARIES and related WAL schemes to improve performance while ensuring correctness, we expect NVRAM to provide sufficiently fast access that centralized logging is no longer useful for performance, and presents only unnecessary overheads. Specifically, the log maintains a total order of updates across all transactions, requiring frequent synchronization to serialize log entries. While multithreaded logging can be made scalable [14], large overheads remain. In this paper, we focus on providing atomic durable transactions, as in ARIES, without the associated overheads and complexity of centralized logging.

	NVRAM Disk-Replacement	*In-Place Updates*	*NVRAM Group Commit*
Software buffer	Traditional WAL/ARIES	Updates both buffer and NVRAM	Buffer limits batch size
Hardware buffer	Impractical	Slow uncached NVRAM reads	Requires hardware support
Replicate to DRAM	Provides fast reads and removes buffer management, but requires large DRAM capacity		

Table 1: NVRAM design space. Database designs include recovery mechanisms (top) and cache configurations (left).

3. NVRAM DATABASE DESIGN

Near-future NVRAM devices will undoubtedly be faster than both disk and flash. However, compared to DRAM many NVRAM technologies impose slower reads and significantly slower persistent writes. We must consider both in redesigning OLTP for NVRAM.

3.1 NVRAM Reads

While the exact read performance of future NVRAM technologies is uncertain, many technologies and devices increase read latency relative to DRAM. Current databases and computer systems are not equipped to deal with this read latency. Disk-backed databases incur sufficiently large read penalties (on the order of milli-seconds) to justify software-managed DRAM caches and buffer management. On the other hand, main-memory databases rely only on the DRAM memory system, including on-chip data caches. Increased memory latency and wide-spread data accesses may require hardware or software-controlled DRAM caches even when using byte addressable NVRAM.

We consider three configurations of cache management; these alternatives form the three rows of Table 1 (we consider the recovery management strategies, forming the three columns, and the resulting design space in subsequent subsections). The first option, *Software Buffer*, relies on software to manage a DRAM buffer cache, as in conventional disk-backed databases. Second, we may omit the cache or rely solely on a *Hardware Buffer*, as in main-memory databases. Hardware caches are fast (e.g., on-chip SRAM) and remove complexity from the software, but provide limited capacity. Third, we might *replicate to DRAM* all data stored in NVRAM—writes update both DRAM and NVRAM (for recovery), but reads retrieve data exclusively from DRAM. Replicating data ensures fast reads by avoiding NVRAM read latencies (except for recovery) and simplifies buffer management, but requires large DRAM capacity.

We investigate how NVRAM read latency will drive storage management design. Our experiments explore the first design option—a software-controlled cache—and vary cache capacity and NVRAM access latency to determine the effect of caching on transaction throughput.

3.2 NVRAM Writes

Persistent writes, unlike reads, do not benefit from caching; writes persist through to the device for recovery correctness. Additionally, NVRAM updates must be carefully ordered to ensure consistent recovery. We assume that ordering is enforced through a mechanism we call a *persist barrier*, which guarantees that writes before the barrier persist before any dependant operations after the barrier persist.

Persist barriers may be implemented in several ways. The easiest, but worst performing, is to delay at persist barriers until all pending NVRAM writes successfully persist. More complicated mechanisms improve performance by allowing threads to continue executing beyond the persist barrier and only delaying thread execution when persist conflicts arise (i.e., a thread reads or overwrites shared data from another thread that has not yet persisted). BPFS provides an example implementation of this mechanism [10]. Regardless of how they are implemented, persist barriers can introduce expensive synchronous delays on transaction threads; as we show, the optimal recovery mechanism depends on how expensive, on average, persist barriers become. To better understand how persist barriers are used and how frequently they occur, we outline operations to atomically update persistent data using persist barriers, and use these operations to implement three recovery mechanisms for NVRAM.

```
persist_wal(log_buffer, nvram_log)
  for entry in log_buffer:
    nvram_log.force_last_lsn_invalid(entry)
    nvram_log.insert_body(entry) # no lsn
  persist_barrier()
  nvram_log.update_lsns()
  persist_barrier()

persist_page(page_v, page_nv, page_log)
  page_log.copy_from(page_nv)
  persist_barrier()
  page_log.mark_valid()
  persist_barrier()
  page_nv.copy_from(page_v)
  persist_barrier()
  page_log.mark_invalid()
  persist_barrier()
```

Figure 2: Durable atomic updates. `persist_wal()` appends to the ARIES log using two persist barriers. `persist_page()` persists pages with four persist barriers.

Atomic durable updates. Figure 2 shows two operations to atomically update NVRAM data. The first, `persist_wal()`, persists log entries into an ARIES log. Shore-MT log entries are post-pended with their Log Serial Number (LSN – log entry file offset). Log entries are considered valid only if the tail LSN matches the location of the entry. We persist log entries atomically by first persisting an entry without its tail LSN and only later persisting the LSN, ordered by a persist barrier. Additionally, we reduce the number of persist barriers by persisting entries in batches, writing several log entries at once (without LSNs), followed by all their LSNs, separated by a single persist barrier. Log operations introduce two persist barriers—one to ensure that log entries persist before their LSNs, and one to enforce that LSNs persist before the thread continues executing.

The second operation, `persist_page()`, atomically persists page data using a persistent undo page log. First, the page's original data is copied to the page log. The page log is marked valid and the dirty version of the page is copied to NVRAM (updated in-place while locks are held). Finally, the log is marked invalid. Four persist barriers ensure that each persist completes before the next, enforcing consistency at all points in execution. Recovery checks the valid flags of all page logs, copying valid logs back in-place. The log is always valid while the page persists in-place, protecting against partial NVRAM writes. Together, `persist_wal()` and `persist_page()` provide the tools necessary to build our recovery mechanisms. We discuss these mechanisms next, describing their implementation and performance.

NVRAM Disk-Replacement. NVRAM database systems will likely continue to rely on ARIES/WAL at first, using NVRAM as *NVRAM Disk-Replacement*. WAL provides recovery for disk by keeping an ordered log of all updates, as described in Section 2.2. While retaining disk's software interface, NVRAM block accesses copy data between volatile and nonvolatile address spaces. *NVRAM Disk-Replacement* in Shore-MT persists the log and pages with `persist_wal()` and `persist_page()`, respectively. Persists occur on log and page flusher threads, and transaction threads do not delay (except when waiting for commit log entries to persist). *NVRAM Disk-Replacement* provides low recovery latency by aggressively flushing pages, minimizing recovery replay. While requiring the least engineering effort, *NVRAM Disk-Replacement* contains large software overheads to maintain a centralized log and asynchronously flush pages. We can leverage NVRAM's low latency to reduce these overheads.

In-Place Updates. Fast, byte-addressable NVRAM allows data to persist in-place, enforcing persist order immediately in a design we call *In-Place Updates*. *In-Place Updates* allows us to remove the centralized log by providing redo and undo log functionality elsewhere. We remove redo logs by keeping the database's durable state up-to-date. In ARIES terms, the database maintains its replayed state—there is no need to replay a redo log after failure. Undo logs need not be ordered across transactions (transactions are free to roll back in any order), so we distribute ARIES undo logs by transaction. Such private logs are simpler and impose fewer overheads than centralized logs. Transaction undo logs remain durable so that in-flight transactions can be rolled back after failure. Page updates (1) latch the page, (2) insert an undo entry into the transaction-private undo log using `persist_wal()`, (3) update the page using `persist_page()` (without an intermediate volatile page), and (4) release the page latch. This protocol ensures all updates to a page, and updates within a transaction, persist in-order, and that no transaction reads data from a page until it is durable.

Persisting data in-place removes expensive redo logging and asynchronous page flushing, but introduces persist barriers on transactions' critical paths. For sufficiently short persist barrier delays *In-Place Updates* outperforms *NVRAM Disk-Replacement*. However, we expect transaction performance to suffer as persist barrier delay increases. We next introduce *NVRAM Group Commit*, a recovery mechanism designed to minimize the frequency of persist barriers while still removing WAL.

NVRAM Group Commit. The two previous recovery mechanisms provide high throughput under certain circumstances, but contain flaws. *NVRAM Disk-Replacement* is insensitive to large persist barrier delays. However, it assumes IO delays are the dominant performance bottleneck and trades off software overhead to minimize IO. *In-Place Updates*, on the other hand, excels when persist barriers delays are short. As persist barrier latency increases performance suffers, and *NVRAM Disk-Replacement* eventually performs better. We provide a third option, coupling *NVRAM Disk-Replacement*'s persist barrier latency-insensitivity with *In-Place Updates*'s low software overhead: *NVRAM Group Commit*.

NVRAM Group Commit operates by executing transactions in batches, whereby all transactions in the batch commit or (on failure) all transactions abort. Transactions quiesce between batches—at the start of a new batch transactions stall until the previous batch commits. Each transaction maintains a private ARIES-style undo log, supporting abort and roll-back as in *In-Place Updates*, but transaction logs are no longer persistent. As batches persist atomically, transactions no longer roll back selectively during recovery (rather, aborted transactions roll back before any data persists), obviating the need for persistent ARIES undo logs. Instead, recovery relies on a database-wide undo log and staging buffer to provide durable atomic batches.

NVRAM Group Commit leverages byte-addressability to reduce persist barrier frequency by ordering persists at batch rather than transaction or page granularity. Batch commit resembles `persist_page()`, used across the entire database, once per batch. Because undo logging is managed at the batch level, transactions' updates may not persist in-place to NVRAM until all transactions in the batch complete. Rather, transactions write to a volatile staging buffer, tracking dirtied cache lines in a concurrent bit field. Once the batch ends and all transactions complete, the pre-batch version of dirtied data is copied to the database-wide persistent undo log, only after which data is copied from the staging buffer in-place to NVRAM. Finally, the database-wide undo log is invalidated, transactions commit, and transactions from the next batch begin executing. On failure the log is copied back, aborting and rolling back all transactions from the in-flight batch. The key observation is that *NVRAM Group Commit* persists entire batches of transactions using four persist barriers, far fewer than required with *In-Place Updates*. Note, however, that it enables recovery only to batch boundaries, rather than transaction boundaries.

We briefly outline two implementation challenges: long transactions and limited staging buffers. Long transactions force other transactions in the batch to defer committing until the long transaction completes. Limited staging buffers, not large enough to hold the entire data set, may fill while transactions are still executing. We solve both problems by resorting to persistent ARIES-style undo logs, as in *In-Place Updates*. Long transactions persist their ARIES undo log (previously volatile), allowing the remainder of the batch to persist and commit. The long transaction joins the next batch, committing when that batch commits. At recovery the most recent batch rolls back, and the long transaction's ARIES undo log is applied, removing updates that persisted with previous batches. Similarly, if the staging buffer fills, the current batch ends immediately and all outstanding transactions persist their ARIES undo logs. The batch persists, treating in-flight transactions as long transactions, reassigning them to the next batch. This mechanism requires additional persistent data structures to allow transaction and batch logs to invalidate atomically.

NVRAM Group Commit requires fewer persist barriers than *In-Place Updates* and avoids *NVRAM Disk-Replacement*'s logging. Batches require four persist barriers regardless of batch length. Persist barrier delays are amortized over additional transactions by increasing batch length, improving throughput. Batch length must be large enough to dominate time spent quiescing transactions between batches. However, increasing batch length defers commit for all transactions in the batch, increasing transaction latency.

3.3 Design Space

We describe the space of possible designs given our choices regarding NVRAM read and write performance. Our discussion ignores possible uses of hard disk to provide additional capacity. Each design works alongside magnetic disk with additional buffer management and the constraint that pages persist to disk before being evicted from NVRAM. Many modern OLTP applications' working sets are small, fitting entirely in main-memory.

Table 1 lists the possible combinations of caching architectures and recovery mechanisms. The left column presents *NVRAM Disk-Replacement*, which we see as the obvious and most incremental use for NVRAM. Of note is the center-left cell, *NVRAM Disk-Replacement* without the use of a volatile buffer. WAL, by its design, allows pages to write back asynchronously from volatile storage. Removing the volatile cache requires transactions to persist data in-place, but do so only after associated log entries persist, retaining the software overheads of *NVRAM Disk-Replacement* as well as the frequent synchronization in *In-Place Updates*. We find this design impractical.

The middle-column recovery mechanism, *In-Place Updates*, represents the most intuitive use of NVRAM in database systems, as noted in several prior works. Agrawal and Jagadish explore several algorithms for atomic durable transactions with an NVRAM main-memory [2]. They describe the operation and correctness of each mechanism and provide an analytic cost model to compare them. Their work represents the middle column, middle row of Table 1 (*In-Place Updates* with no volatile buffer). Akyürek and Salem present a hybrid DRAM and NVRAM buffer cache design alongside strategies for managing cache allocation [23]. Such *Partial Memory Buffers* resemble the middle-top cell of our design space table (*In-Place Updates* with a software-managed DRAM buffer), although their design considers NVRAM as part of a hybrid buffer, not the primary persistent store. Neither of these works considers alternative approaches (such as *NVRAM Group Commit*), to account for large persist barrier latency and associated delays. Additionally, we extend prior work by providing a more precise performance evaluation and more detailed consideration of NVRAM characteristics.

The right column presents *NVRAM Group Commit.* Each of our three recovery mechanisms may replicate all data between NVRAM and DRAM to ensure fast read accesses, manage a smaller DRAM buffer cache, or omit the cache altogether. In Section 5 we consider the importance of NVRAM caching to transaction throughput. Then, in Section 6 we assume a DRAM-replicated heap to isolate read performance from persist performance in evaluating each recovery mechanisms's ability to maximize transaction throughput. The next section describes our evaluation methodology.

Operating System	Ubuntu 12.04
CPU	Intel Xeon E5645 2.40 GHz
CPU cores	6 (12 with HyperThreading)
Memory	32 GB

Table 2: Experimental system configuration.

4. METHODOLOGY

This section details our methodology for benchmarking transaction processing and modeling NVRAM performance. Our experiments use the Shore-MT storage manager [13], including the high performance, scalable WAL implementation provided by Aether [14]. While Aether provides a distributed log suitable for multi-socket servers, the distributed log exists as a fork of the main Shore-MT project. Instead, we limit experiments to a single CPU socket to provide a fair comparison between WAL and other recovery schemes, enforced using the Linux *taskset* utility. Our experiments place both the Shore-MT log and volume files on an in-memory *tmpfs*, and provide sufficiently large buffer caches such that all pages hit in the cache after warmup. The intent is to allow the database to perform data accesses at DRAM speed and introduce additional delays to model NVRAM performance. Table 2 shows our system configuration.

Modeling NVRAM delays. Since NVRAM devices are not yet available, we must provide a timing model that mimics their expected performance characteristics. We model NVRAM read and write delays by instrumenting Shore-MT with precisely controlled assembly-code delay loops to model additional NVRAM latency and bandwidth constraints at 20ns precision. Hence, Shore-MT runs in real time as if its buffer cache resided in NVRAM with the desired read and write characteristics.

We introduce NVRAM read and write delays separately. Accurately modeling per-access increases in read latency is challenging, as reads are frequent and the expected latency increases on NVRAM are small. It is infeasible to use software instrumentation to model such latency increases at the granularity of individual reads; hardware support, substantial time dilation, or alternative evaluation techniques (e.g., simulation) would be required, all of which compromise accuracy and our ability to run experiments at full scale. Instead, we use offline analysis with PIN [16] to determine (1) the reuse statistics of buffer cache pages, and (2) the average number of cache lines accessed each time a page is latched. Together, these offline statistics allow us to calculate an average number of cache line accesses per page latch event in Shore-MT while considering the effects of page caching. We then introduce a delay at each latch based on the measured average number of misses and an assumed per-read latency increase based on the NVRAM technology.

We model NVRAM persist delays by annotating Shore-MT to track buffer cache writes at cache line granularity—64 bytes—using efficient "dirty" bitmaps. Depending on the recovery mechanism, we introduce delays corresponding to persist barriers and to model NVRAM write bandwidth contention. Tracking buffer cache writes introduces less than a 3% overhead to the highest throughput experiments.

We create NVRAM delays using the x86 RDTSCP instruction, which returns a CPU-frequency-invariant, monotonically increasing time-stamp that increments each clock tick. RDTSCP is a synchronous instruction—it does not allow other instructions to reorder with it. Our RDTSCP loop delays threads in increments of 20ns (latency per loop iteration and RDTSCP) with an accuracy of 2ns.

In addition to NVRAM latency, we model shared NVRAM write bandwidth. Using RDTSCP as a clock source, we maintain a shared *next_available* variable, representing the next clock tick in which the NVRAM device is available to be written. Each NVRAM persist advances *next_available* to account for the latency of its persist operation. Reservations take the maximum of *next_available* and the current RDTSCP and add the reservation duration. The new value is atomically swapped into *next_available* via a Compare-And-Swap (CAS). If the CAS fails (due to a race with a persist operation on another thread), the process repeats until it succeeds. Upon success, the thread delays until the end of its reservation. The main limitation of this approach is that it cannot model reservations shorter than the delay required to perform a CAS to a contended shared variable. This technique models reservations above 85ns accurately, which is sufficient for our experiments.

We choose on-line timing modeling via software instrumentation in lieu of architectural simulations to allow our experiments to execute at full scale and in real time. While modeling aspects of NVRAM systems such as cache performance and more precise persist barrier delays require detailed hardware simulation, we believe NVRAM device and memory system design are not sufficiently established to consider this level of detail. Instead, we investigate more general trends to determine if and when NVRAM read and write performance warrant storage management redesign.

Recovery performance. Figure 1 displays recovery latency vs transaction throughput for the TPCB workload, varying page flush rate. We control page flush rate by maintaining a constant number of dirty pages in the buffer cache, always flushing the page with the oldest volatile update. Experiments run TPCB for one minute (sufficient to reach steady state behavior) and then kill the Shore-MT process. Before starting recovery we drop the file system cache. Reported recovery time includes only the recovery portion of the Shore-MT process; we do not include system startup time nor non-recovery Shore-MT startup time.

Workloads We use three workloads and transactions in our evaluation: TPCC, TPCB, and TATP. TPCC models order management for a company providing a product or service [26]. TPCB contains one transaction class and models a bank executing transactions across branches, tellers, customers, and accounts [25]. TATP includes seven transactions to model a Home Location Registry used by mobile carriers [1]. Table 3 shows our workload configuration. We scale workloads to fit in a 12GB buffer cache. Persist performance experiments use a single write-heavy transaction from each workload while read performance experiments use each workload's full mix. All experiments report throughput as thousands of Transactions Per Second (kTPS). Experiments perform "power runs" – each thread generates and executes transactions continuously without think time – and run an optimal number of threads per configuration (between 10 and 12).

5. NVRAM READ PERFORMANCE

We first evaluate database performance with respect to NVRAM reads. Many candidate NVRAM technologies exhibit greater read latency than DRAM, possibly requiring additional hardware or software caching. We wish to determine, for a given NVRAM read latency, how much caching is necessary to prevent slowdown, and whether it is feasible to provide this capacity in a hardware-controlled cache (otherwise we must resort to software caches).

Workload	Scale factor	Size	Write transaction
TPCC	70	9GB	New order
TPCB	1000	11GB	
TATP	600	10GB	Update location

Table 3: Workloads and transactions.

5.1 NVRAM Caching Performance

Traces. Our NVRAM read-performance model combines memory access trace analysis with our timing model to measure transaction throughput directly in Shore-MT. Traces consist of memory accesses to the buffer cache, collected running Shore-MT with PIN for a single transaction thread for two minutes. We assume concurrent threads exhibit similar access patterns. In addition, we record all latch events (acquire and release) and latch page information (i.e., table id, table type – index, heap, or other). We analyze traces at cache line (64 bytes) and page (8KB) granularity.

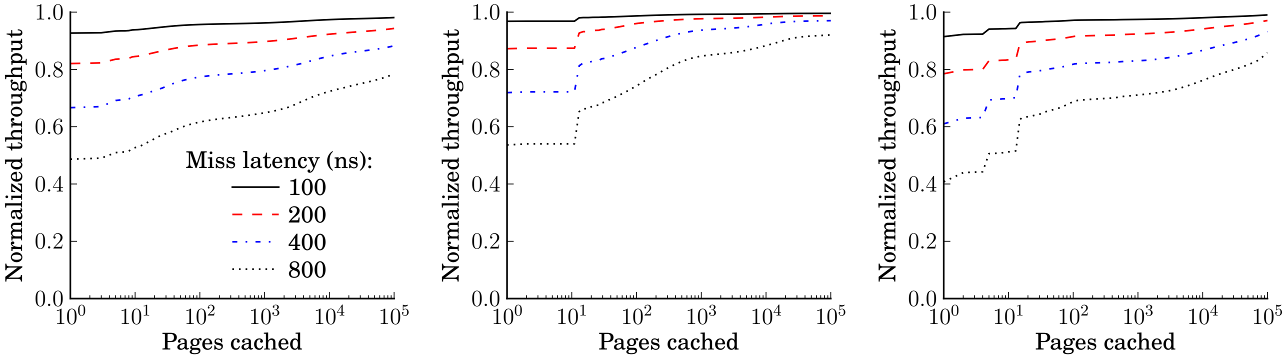
These traces provide insight into how Shore-MT accesses persistent data, summarized in Table 4. Index accesses represent the great majority of cache line accesses, averaging 85% of accesses to NVRAM across workloads. Any caching efforts should focus primarily on index pages and cache lines. We also note that indices access a greater number of cache lines per page access than other page types (average 11.48 vs 4.85 for heap pages and 4.77 for other page types), suggesting that uncached index page accesses have the potential to introduce greater delays.

Throughput. We create a timing model in Shore-MT from our memory traces. Given traces, we perform cache analysis at page granularity, treating latches as page accesses and assuming a fully associative cache with a least-recently-used replacement policy (LRU). Cache analysis produces an average page miss rate to each table. We conservatively assume that every cache line access within an uncached page introduces an NVRAM stall—we neglect optimizations such as out-of-order execution and simultaneous multi-threading that might hide some NVRAM access stalls. Our model assumes the test platform incurs a 50ns DRAM fetch latency, and adds additional latency to mimic NVRAM (for example, a 200ns NVRAM access adds 150ns delay per cache line). We combine average page miss rate and average miss penalty (from lines/latch in table 4) to compute the average delay incurred per latch event. This delay is inserted at each page latch acquire in Shore-MT, using *In-Place Updates*, to produce a corresponding throughput.

Figure 3 shows throughput achieved for the three workloads while varying the number of pages cached (horizontal axis) and NVRAM miss latency (various lines). The vertical axis displays throughput normalized to DRAM-miss-latency's throughput (no additional delay inserted). Without caching, throughput suffers as NVRAM miss latency increases, shown at the extreme left of each graph. A 100ns miss latency consistently achieves at least 90% of potential throughput. However, an 800ns miss latency averages only 50% of the potential throughput, clearly requiring caching. TPCB and TPCC see a 10-20% throughput improvement

	TATP		TPCB		TPCC		Average	
	% lines	lines/latch	% lines	lines/latch	% lines	lines/latch	% lines	lines/latch
Heap	7.26%	4.19	15.47%	4.25	15.27%	6.10	12.66%	4.85
Index	92.45%	11.82	81.18%	11.17	81.54%	11.44	85.06%	11.48
Other	0.29%	3.00	3.36%	3.00	3.19%	8.31	2.28%	4.77
Total		11.24		9.83		10.52		10.48

Table 4: NVRAM access characteristics. "% lines" indicates the percentage breakdown of cache line accesses. "lines/latch" reports the average number of cache line accesses per page latch. Indices represent the majority of accesses.



(a) TATP

(b) TPCB

(c) TPCC

Figure 3: Throughput vs NVRAM read latency. 100ns miss latency suffers up to a 10% slowdown over DRAM. Higher miss latencies introduce large slowdowns, requiring caching. Fortunately, even small caches effectively accelerate reads.

for a cache size of just 20 pages. As cache capacity further increases, each workload's throughput improves to varying degrees. A cache capacity of 100,000 (or 819MB at 8KB pages) allows NVRAMs with 800ns miss latencies to achieve at least 80% of the potential throughput. While too large for on-chip caches, such a buffer might be possible as a hardware-managed DRAM cache [22].

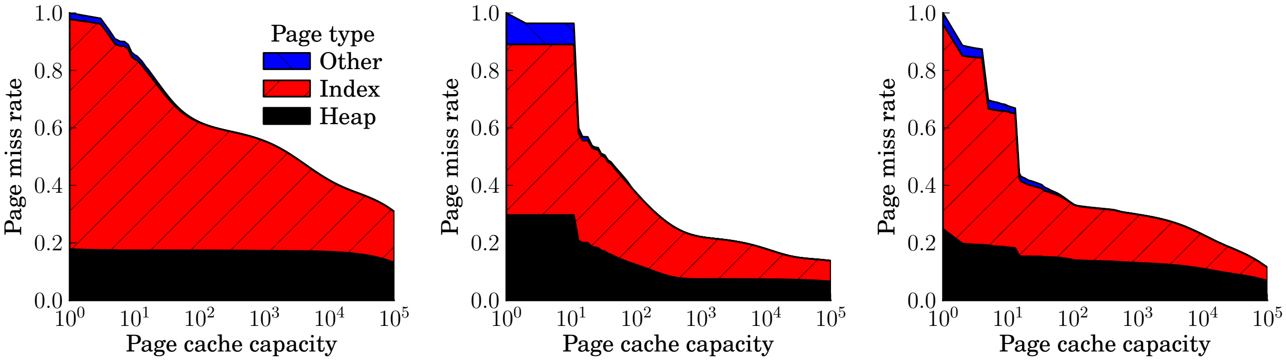
5.2 Analysis

We have shown that modest cache sizes effectively hide NVRAM read stalls for our workloads, and further analyze caching behavior to reason about OLTP performance more generally. Figure 4 shows the page miss rate per page type (index, heap, or other) as page cache capacity increases. Each graph begins at 1.0 at the left – all page accesses miss for a single-page cache. As cache capacity increases, workloads see their miss rates start to decrease between cache capacity of five and 20 pages. TATP experiences a decrease in misses primarily in index pages, whereas TPCB and TPCC see a decrease across all page types.

While cache behavior is specific to each workload, our results represent trends applicable to many databases and workloads, specifically, index accesses and append-heavy tables. First, all workloads see a decrease in index page misses as soon as B+Tree roots (accessed on every traversal) successfully cache. The hierarchical nature of B+Tree indices allows high levels of the tree to cache effectively for even a small cache capacity. Additionally, TPCB and TPCC contain history tables to which data are primarily appended. Transactions append to the same page as previous transactions, allowing such tables to cache effectively. Similarly, extent map pages used for allocating new pages and locating pages to append into are frequently accessed and likely to cache. The remaining tables' pages are accessed randomly and only cache as capacity approaches the size of each table. In the case of TPCB and TPCC, each transaction touches a random tuple of successively larger tables (Branch, Teller, and Account for TPCB; Warehouse, District, Customer, etc. for TPCC). Our analysis suggests that various page types, notably index and append-heavy pages, cache effectively, accelerating throughput for high-latency NVRAM misses with small cache capacities.

Main-memory databases. While we use Shore-MT (a disk-based storage manager) as a research platform, we believe main-memory database optimizations (e.g., [11, 4, 19]) apply to byte-addressable NVRAM. Main-memory databases assume heap data resides solely in byte-addressable memory, improving throughput relative to traditional disk-backed storage by removing expensive indirection (i.e., using memory pointers instead of buffer translation tables), reducing overheads associated with concurrency control and latches, and optimizing data layout for caches and main-memory, among other optimizations. While such optimizations will increase transaction throughput, removing non-NVRAM overheads will amplify the importance of read-miss latency (an equal increase in NVRAM read-miss latency will yield a relatively greater drop in performance). At the same time, data layout optimizations will reduce the number of cache lines and memory rows accessed per action (e.g., per latch), minimizing NVRAM read overheads. Investigating main-memory database optimizations for NVRAM remains future work.

Bandwidth. Finally, we briefly address NVRAM read bandwidth. For a worst-case analysis, we assume no caching. Given the average number of cache line accesses per page latch, the average number of page latches per transaction, and transaction throughput (taken from Section 6), we compute worst-case NVRAM read bandwidth for each workload, shown in Table 5 Our workloads require at most 1.168 GB/s (TPCC). Since this is lower than expected NVRAM band-



(a) TATP **(b)** TPCB **(c)** TPCC

Figure 4: Page caching. B+Tree pages and append-heavy heap pages cache effectively.

Workload	Bandwidth (GB/s)
TATP	0.977
TPCB	1.044
TPCC	1.168

Table 5: Required NVRAM read bandwidth. Workloads require up to 1.2 GB/s read bandwith.

width and caching reduces the required bandwidth further, we conclude that NVRAM read bandwidth for persistent data on OLTP is not a concern.

5.3 Summary

NVRAM presents a new storage technology for which modern database systems have not been optimized. Increased memory read latencies require new consideration for database caching systems. We show that persistent data for OLTP can be cached effectively, even with limited cache capacity. We expect future NVRAM software to leverage hardware caches, omitting software buffer caches. Next, we turn to write performance for storage management on NVRAM devices.

6. NVRAM PERSIST SYNCHRONIZATION

Whereas NVRAM reads benefit from caching, persists must always access the device. We are particularly interested in the cost of ordering persists via persist barriers. Several factors increase persist barrier latency, including ordering persists across distributed/NUMA memory architectures, long latency interconnects (e.g., PCIe-attached storage), and slow NVRAM MLC cell persists. We consider the effect of persist barrier latency on transaction processing throughput to determine if and when new NVRAM technologies warrant redesigning recovery management.

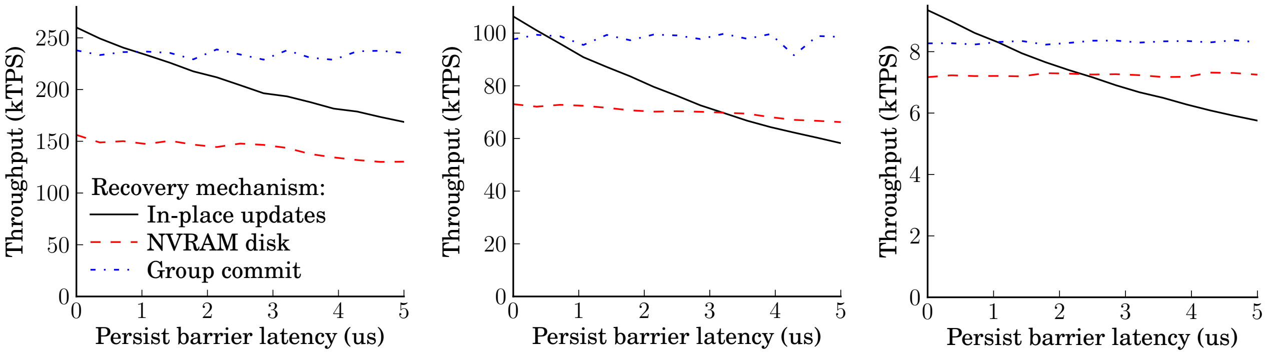
Refer to Sections 3 and 4 for a more thorough description of our recovery mechanisms and experimental setup. All experiments throttle persist bandwidth to 1.5GB/s, which we believe to be conservative (already possible with PCIe-attached Flash). Ideally, NVRAM will provide low latency access, enabling *In-Place Updates*. However, we expect *In-Place Updates*'s performance will suffer at large persist barrier latencies, requiring either *NVRAM Disk-Replacement* or *NVRAM Group Commit* to regain throughput.

6.1 Persist Barrier Latency

Figure 5 shows throughput for write-heavy transactions as persist barrier latency increases from 0µs to 5µs, the range we believe encompasses realistic latencies for possible implementations of persist barriers and storage architectures. A persist barrier latency of 0µs (left edge) corresponds to no barrier/DRAM latency. For such devices (e.g., battery-backed DRAM), *In-Place Updates* far out-paces *NVRAM Disk-Replacement*, providing up to a 50% throughput improvement. The speedup stems from a combination of removing WAL overheads, removing contention between page flushers and transaction threads, and freeing up (a few) threads from log and page flushers to run additional transactions. *In-Place Updates* also outperforms *NVRAM Group Commit*, providing an average 10% throughput improvement across workloads.

As persist barrier latency increases, each recovery mechanism reacts differently. *In-Place Updates*, as expected, loses throughput. *NVRAM Disk-Replacement* and *NVRAM Group Commit*, on the other hand, are both insensitive to persist barrier latency; their throughputs see only a small decrease as persist barrier latency increases. TATP sees the largest throughput decrease for *NVRAM Disk-Replacement* (14% from 0µs to 5µs). The decrease stems from *NVRAM Disk-Replacement*'s synchronous commits, requiring the log flusher thread to complete flushing before transactions commit. During this time, transaction threads sit idle. While both *NVRAM Disk-Replacement* and *NVRAM Group Commit* retain high throughput, there is a large gap between the two, with *NVRAM Group Commit* providing up to a 50% performance improvement over *NVRAM Disk-Replacement.* This difference, however, is workload dependent, with WAL imposing a greater bottleneck to TATP than to TPCB or TPCC.

Of particular interest are persist barrier latencies where lines intersect—the break-even points for determining the optimal recovery mechanism. Whereas all workloads prefer *In-Place Updates* for a 0µs persist barrier latency, *NVRAM Group Commit* provides better throughput above 1µs persist barrier latency. When only considering *In-Place Updates* and *NVRAM Disk-Replacement* the decision is less clear. Over our range of persist barrier latencies TATP always prefers *In-Place Updates* to *NVRAM Disk-Replacement* (the break-even latency is well above 5µs). TPCB and TPCC see the two mechanisms intersect near 3.5µs and 2.5µs, re-



(a) TATP – Update Location

(b) TPCB

(c) TPCC – New Order

Figure 5: Throughput vs persist barrier latency. *In-Place Updates* performs best for zero-cost persist barriers, but throughput suffers as persist barrier latency increases. *NVRAM Disk-Replacement* and *NVRAM Group Commit* are both insensitive to increasing persist barrier latency, with *NVRAM Group Commit* offering higher throughput.

Workload	Full mix	Single transaction
TATP	25	12
TPCB	3.2	3.2
TPCC	3.6	2.4

Table 6: Break-even persist latency Persist barrier latency (µs) where *NVRAM Disk-Replacement* and *In-Place Updates* achieve equal throughput. Latencies reported for full transaction mixes and single write-heavy transaction per workload.

spectively, above which *NVRAM Disk-Replacement* provides higher throughput. TATP, unlike the other two workloads, only updates a single page per transaction. Other overheads tend to dominate transaction time, resulting in a relatively shallow *In-Place Updates* curve.

The previous results show throughput only for a single transaction from each workload. Table 6 shows break-even persist barrier latency between *NVRAM Disk-Replacement* and *In-Place Updates* for these transactions and full transaction mixes. Full transaction mixes contain read-only transactions, reducing log insert and persist barrier frequency (read-only transactions require no recovery). *NVRAM Disk-Replacement* sees improved throughput at 0 µs and *In-Place Updates*'s throughput degrades less quickly as persist barrier latency increases. As a result, the break-even persist barrier latency between these two designs increases for the full transaction mix relative to a single write-heavy transaction and the opportunity to improve throughput by optimizing recovery management diminishes—improved recovery management does not affect read-only transactions and actions.

Our results suggest different conclusions across storage architectures. NVRAM connected via the main memory bus will provide low latency persist barriers (less than 1µs) and prefer *In-Place Updates*. Other storage architectures, such as distributed storage, require greater delays to synchronize persists. For such devices, *NVRAM Group Commit* offers an alternative to *NVRAM Disk-Replacement* that removes software overheads inherent in WAL while providing recovery. However, *NVRAM Group Commit* increases transaction latency, which we consider next.

6.2 Transaction Latency

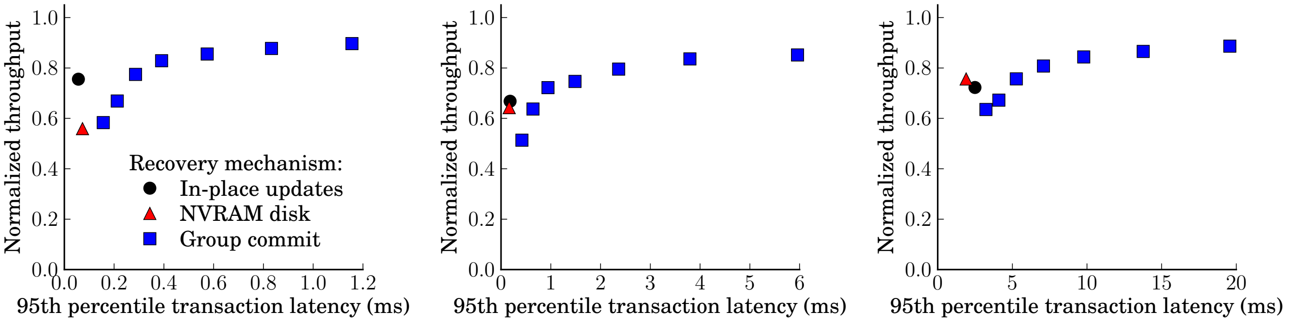
NVRAM Group Commit improves transaction throughput by placing transactions into batches and committing all transactions in a batch atomically. Doing so minimizes and limits the number of inserted persist barriers. However, deferring transaction commit increases transaction latency, especially for the earliest transactions in each batch. To achieve reasonable throughput, batches must be significantly longer than average transaction latency (such that batch execution time dominates batch quiesce and persist time). The batch period acts as a knob for database administrators to trade off transaction latency and throughput. We use this knob to measure the relationship between throughput and high-percentile transaction latency.

Figure 6 shows throughput, normalized to *In-Place Updates* at 0µs persist barrier latency. Our results consider a 3µs persist barrier latency, where *NVRAM Group Commit* provides a throughput improvement over other recovery mechanisms. The different *NVRAM Group Commit* points represent different batch periods, and we report the measured 95th percentile transaction latency for all recovery mechanisms. We measure transaction latency from the time a transaction begins to the time its batch ends (Shore-MT does not model any pre-transaction queuing time).

The results illustrate that *NVRAM Group Commit* is capable of providing equivalent throughput to the other recovery mechanisms with reasonable latency increases (no more than 5×). Further, high-percentile transaction latencies fall well below the latency expectations of modern applications. TPCC, the highest latency workload, approaches optimal throughput with a 95th percentile transaction latency of 15ms—similar to latencies incurred by disk-backed databases. For latency sensitive workloads, the batch period can be selected to precisely control latency, and *In-Place Updates* and *NVRAM Disk-Replacement* remain alternatives.

6.3 Summary

Persist barriers used to enforce persist order pose a new obstacle to providing recoverable storage management with NVRAM. We show that, for memory bus-attached NVRAM devices, ensuring recovery using *In-Place Updates* is a viable strategy that provides high throughput and removes the overheads of WAL. For interconnects and NVRAM technolo-



(a) TATP – Update Location

(b) TPCB

(c) TPCC – New Order

Figure 6: 95th percentile transaction latency. Graphs normalized to *In-Place Updates* 0µs persist latency. Experiments use 3µs persist latency. *NVRAM Group Commit* avoids high latency persist barriers by defering transaction commit.

gies that incur larger persist barrier delays, *NVRAM Group Commit* offers an alternative that yields high throughput and reasonable transaction latency. *NVRAM Group Commit*'s batch period allows precise control over transaction latency for latency-critical applications.

7. RELATED WORK

To the best of our knowledge, our work is the first to investigate NVRAM write latency and its effect on durable storage and recovery in OLTP. A large body of related work considers applications of NVRAM and reliable memories.

Ng and Chen place a database buffer cache in battery-backed DRAM, treating it as reliable memory [18]. However, the mechanisms they investigate are insufficient to provide ACID properties under any-point failure or protect against many types of failure (e.g., power loss).

Further work considers NVRAM in the context of file systems. Baker *et al.* use NVRAM as a file cache to optimize disk I/O and reduce network traffic in distributed file systems, yet continue to assume that disk provides the bulk of persisted storage [3]. More recently, Condit *et al.* demonstrate the hardware and software design necessary to implement a file system entirely in NVRAM as the Byte-Addressable Persistent File System (BPFS) [10]. While we assume similar hardware, we additionally consider a range of NVRAM performance and focus instead on databases.

Other work develops programming paradigms and system organizations for NVRAM. Coburn *et al.* propose NV-Heaps to manage NVRAM within the operating system, provide safety guarantees while accessing persistent stores, and atomically update data using copy-on-write [9]. Volos *et al.* similarly provide durable memory transactions using Software Transactional Memory (STM) and physical redo logging per transaction [28]. While these works provide useful frameworks for NVRAM, they do not investigate the effect of NVRAM persist latency on performance, nor do they consider OLTP, where durability is tightly coupled with concurrency and transaction management.

Recently, researchers have begun to focus specifically on databases as a useful application for NVRAM. Chen *et al.* reconsider database algorithms and data structures to address NVRAM's write latency, endurance, and write energy concerns, generally aiming to reduce the number of modified NVRAM bits [8]. However, their work does not consider durable consistency for transaction processing. Venkataraman *et al.* demonstrate a multi-versioned log-free B-Tree for use with NVRAM [27]. Indices are updated in place, similarly to our *In-Place Updates*, without requiring any logging (physical or otherwise) and while providing snap shot reads. Our work considers durability management at a higher level, user transactions, and consistency throughout the entire database. Finally, Fang *et al.* develop a new WAL infrastructure for NVRAM that leverages byte addressable and persistent access [12]. Fang aims to improve transaction throughput but retains centralized logging. We distinguish ourselves by investigating how NVRAM write performance guides database design.

Prior work (e.g., H-Store [24]) has suggested highly available systems as an outright replacement for durability. We argue that computers and storage systems will always fail, and durability remains a requirement for many applications.

8. CONCLUSION

New NVRAM technologies offer an alternative to disk that provides high performance while maintaining durable transaction semantics, yet existing database software is not optimized for such storage devices. In this paper, we consider redesigning OLTP software to optimize for NVRAM read and persist characteristics. We find that even small caches effectively reduce NVRAM read stalls. We also consider database performance in the presence of persist barrier delays. Treating NVRAM as a drop-in replacement for disk, *NVRAM Disk-Replacement* retains centralized logging overheads. *In-Place Updates* reduces these overheads, but for large persist barrier latencies suffers from excessive synchronization stalls. We propose a new recovery mechanism, *NVRAM Group Commit*, to minimize these stalls while still removing centralized logging. While *NVRAM Group Commit* increases high-percentile transaction latency, latency is controllable and within modern application constraints.

9. REFERENCES

[1] Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net.

[2] R. Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. In *Proceedings of the Sixth International Workshop on Database Machines*, pages 269–285, 1989.

[3] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. Non-volatile memory for fast, reliable file

systems. In *Proc. of the 5th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, 1992.

[4] C. Ballard, D. Behman, A. Huumonen, K. Laiho, J. Lindstrom, M. Milek, M. Roche, J. Seery, K. Vakkila, J. Watters, and A. Wolski. IBM solidDB: Delivering data with extreme speed. 2011.

[5] 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. of Research and Development*, 52:449–464, 2008.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proc. of the 43rd International Symp. on Microarchitecture*, pages 385–395, 2010.

[7] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. of the 17th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 387–400, 2012.

[8] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proc. of the 5th Bienniel Conf. on Innovative Data Systems Research*, pages 21–31, 2011.

[9] 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 *Proc. of the 16th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[10] 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 *Proc. of the 22nd Symp. on Operating Systems Principles*, pages 133–146, 2009.

[11] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[12] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Proc. of the 27th International Conf. on Data Engineering*, pages 1221–1231, 2011.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th International Conf. on Extending Database Technology*, pages 24–35, 2009.

[14] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, pages 681–692, Sept. 2010.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[17] 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.*, pages 94–162, Mar. 1992.

[18] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. In *Proc. of the International Conf. on Very Large Data Bases*, pages 76–85, 1997.

[19] Oracle America. Extreme performance using Oracle TimesTen in-memory database, an Oracle technical white paper. July 2009.

[20] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be SSD-aware? In *Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures*, 2011.

[21] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proc. of the 42nd International Symp. on Microarchitecture*, pages 14–23, 2009.

[22] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of the 36th International Symp. on Computer Architecture*, pages 24–33, 2009.

[23] K. Salem and S. Akyürek. Management of partially safe buffers. *IEEE Trans. Comput.*, pages 394–407, Mar. 1995.

[24] 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 *Proc. of the 33rd International Conf. on Very Large Data Bases*, pages 1150–1160, 2007.

[25] Transaction Processing Performance Council (TPC). TPC-B Benchmark. http://www.tpc.org/tpcb/.

[26] Transaction Processing Performance Council (TPC). TPC-C Benchmark. http://www.tpc.org/tpcc/.

[27] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th Usenix Conference on File and Storage Technologies*, pages 61–75, 2011.

[28] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.