NVRAM-aware Logging in Transaction Systems

Jian Huang
jhuang95@cc.gatech.edu

Karsten Schwan
schwan@cc.gatech.edu

Moinuddin K. Qureshi
moin@ece.gatech.edu

Georgia Institute of Technology

ABSTRACT

Emerging byte-addressable, non-volatile memory technologies (NVRAM) like phase-change memory can increase the capacity of future memory systems by orders of magnitude. Compared to systems that rely on disk storage, NVRAM-based systems promise significant improvements in performance for key applications like online transaction processing (OLTP). Unfortunately, NVRAM systems suffer from two drawbacks: their asymmetric read-write performance and the notable higher cost of the new memory technologies compared to disk. This paper investigates the cost-effective use of NVRAM in transaction systems. It shows that using NVRAM only for the logging subsystem (*NV-Logging*) provides much higher transactions per dollar than simply replacing all disk storage with NVRAM. Specifically, for *NV-Logging*, we show that the software overheads associated with centralized log buffers cause performance bottlenecks and limit scaling. The per-transaction logging methods described in the paper help avoid these overheads, enabling concurrent logging for multiple transactions. Experimental results with a faithful emulation of future NVRAM-based servers using the TPCC, TATP, and TPCB benchmarks show that *NV-Logging* improves throughput by 1.42 - 2.72x over the costlier option of replacing all disk storage with NVRAM. Results also show that *NV-Logging* performs 1.21 - 6.71x better than when logs are placed into the PMFS NVRAM-optimized file system. Compared to state-of-the-art distributed logging, *NV-Logging* delivers 20.4% throughput improvements.

1. INTRODUCTION

Byte-addressable, non-volatile memory (NVRAM) is emerging as a promising way forward to substantially enhance future server systems. Its main advantages of near-DRAM speed, lower than DRAM power consumption, and non-volatility suggest its utility both for augmenting memory capacity, and for improving performance of systems with persistence requirement.

Proceedings of the VLDB Endowment, Vol. 8, No. 4

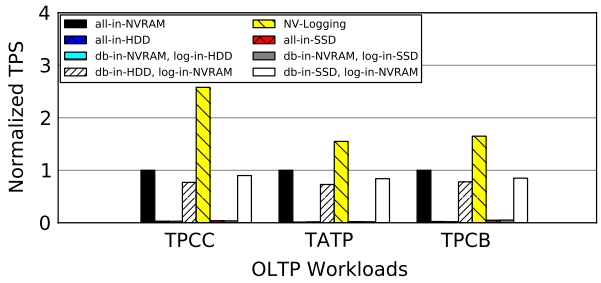


Figure 1: Throughput comparison, taking *all-in-NVRAM* as the baseline.

This paper explores the use of NVRAM for speeding up performance-critical transaction systems. Conventional transaction systems use DRAM as cache to host data pages and log records. Upon transaction commit, log records must be flushed into persistent storage. With the relatively high latency of disk storage, overall system performance, therefore, is constrained by the disk I/O bottleneck. Past work has addressed this issue with write-ahead logging (WAL), made efficient with additional software solutions like log group commit [27], early lock release (ELR) [17], and speculative lock inheritance (SLI) [15], but at the risk of partial data loss or incorrect results due to the inherent delays in placing data on disk.

NVRAM offers much shorter I/O latency compared to disk, promising notable performance improvements. This also implies that when using NVRAM to replace formerly disk-based storage, it will be software overheads that cause performance bottlenecks. This suggests the need to rethink transaction system architectures and their implementation. Pelley et al. [25], for instance, proposed using NVRAM as main memory for in-memory databases, thus leveraging its potentially large memory capacity. The software overheads exposed when using NVRAM include those caused by the barriers used to ensure persistence for in-place updates.

This paper presents a comprehensive study on alternative ways to restructure a transaction system for effective use of NVRAM. As baselines, it considers options in which (1) NVRAM is used as a disk replacement accessed via standard I/O interfaces, termed *NV-Disk*, and (2) NVRAM replaces the entire system's main memory [25], termed *NV-WSP*. These baselines are compared with our improved solution – *NV-Logging* – a cost-effective, high performance, NVRAM-aware transaction system. In *NV-Logging*, improved cost-performance is obtained by reducing the amount of costly NVRAM rather than replacing all disk storage with NVRAM, and implementing a NVRAM-aware logging sub-

system. The implementation acknowledges that (1) NVRAM differs from DRAM in its characteristics – it has asymmetric read-write performance and non-volatility, and (2) it avoids the software overheads that can dominate the performance of NVRAM-based solutions. *NV-Logging*'s technical contributions are as follows:

- Log buffers are placed into NVRAM as data structures directly accessed via hardware-supported memory references vs. via costlier, software-based I/O interfaces.
- *Per-transaction logging* avoids the software overheads of centralized log buffers for block-oriented storage, and enables highly concurrent logging.
- Certain known overheads of NVRAM are avoided with *flush-on-insert* and *flush-on-commit* methods for log object persistence and consistency.

NV-Logging is implemented in the latest open source transaction system Shore-MT [5]. Experimental results with the TPCC, TATP, and TPCB benchmarks demonstrate that *NV-Logging* improves throughput by 1.42 - 2.72x compared to the *NV-Disk* approach. Additional throughput improvements are seen with SLI [15] and ELR [17] enabled, resulting in *NV-Logging* performing 1.18 - 2.66x better than *NV-Disk*(i.e., *all-in-NVRAM*). Furthermore, compared to state-of-the-art distributed logging [29], *NV-Logging* improves the throughput by 8.93 - 26.81%, with a much simpler design. Finally, using Intel Corporation's experimental persistent memory server, which employs a modified CPU and custom firmware to emulate future NVRAM hardware, we compare *NV-Logging* with an alternative implementation of its functionality in which log data structures are supported by the NVRAM-optimized PMFS file system [12]. In this setup, experimental results with TPCB and TATP benchmarks show that the native implementation of *NV-Logging* performs 1.21 - 6.71x better than its PMFS realization, particularly for update-intensive transactions.

The remainder of this paper is organized as follows. Section 2 introduces the conventional disk-based logging solutions. Section 3 describes candidate uses of NVRAM in transaction systems and presents their performance analysis. We describe the *NV-Logging* design in Section 4 and its implementation in Section 5. Section 6 shows experimental results with OLTP benchmarks. We summarize related work in Section 7, and conclude the paper in Section 8.

2. BACKGROUND & MOTIVATION

Logging is an essential means for guaranteeing the ACID (Atomicity, Consistency, Isolation, and Durability) properties of database systems. It can also become a bottleneck, as logs must be made persistent upon transaction commits.

2.1 Architecture of Disk-based Logging

A typical disk-based logging system has the two main components depicted in Figure 2: log buffer and log partitions. The log buffer is located in memory, while log partitions are kept in persistent storage. Centralizing the log buffer makes it possible to group log records, thereby avoiding frequent disk accesses and obtaining sequential disk access patterns. A potential drawback is logging contention when concurrently accessed by multiple requests, which has become evident in transaction systems in the multicore era

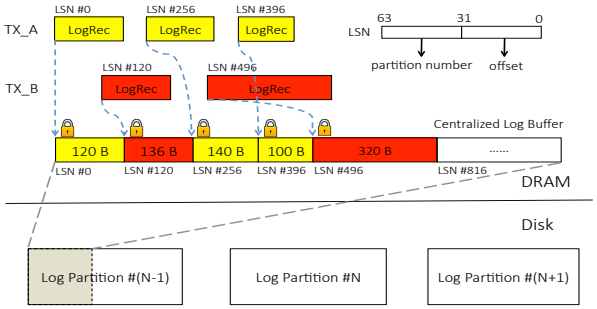


Figure 2: Disk-based logging system. The log file is organized into multiple partitions, where the log buffer is flushed to partitions in sequential order.

[29, 17]. Furthermore, with a synchronous commit strategy, each commit cannot complete until all corresponding log records have been flushed to persistent storage, causing potentially high request delays due to today's high disk access latencies. With an asynchronous commit strategy, a request can continue its execution without waiting until all log records are persistent. This improves transaction throughput dramatically, but at the expense of data durability, since all of the unflushed data in the log buffer is lost if a crash occurs.

Logging operates as follows. Each transaction generates a set of log records. Before inserting these into the log buffer, the transaction must acquire a lock on the buffer to check for available buffer space. If there is no space left, the log buffer must be reclaimed. This requires its dirty pages to be flushed to disk and the associated active transactions to be aborted.

To track log record location and maintain order across the log records among all transactions, each log record has a unique log sequence number (LSN). In the example shown in Figure 2, the LSN is split into two parts: the high 32 bits represent the partition number, and the low 32 bits indicate its offset in the corresponding partition. When space is available and log buffer space is allocated successfully, a LSN is assigned to the log record, and the global LSN is updated when the completed log record has been inserted into the log buffer. For example, the LSN of TX_A's first log record is 0, the LSN of TX_B's first log record is 120 as one log record whose size is 120 bytes from TX_A has been inserted into the log buffer. The procedure repeats for subsequent log records.

Each log record insertion generally involves one memory copy and one disk I/O write, and during the entire procedure, locks are used to protect the log buffer space and maintain log record orderings. With such a lock-based approach, the average latency of log insertion increases dramatically with an increasing number of threads.

2.2 Software Solutions Alleviating Contention

In a transaction system requiring durability guarantees, logging is on the critical path of transaction processing. With synchronous log commits, Figure 3 shows that the transaction TXN_B cannot make progress until TXN_A has flushed its log record into persistent storage and released the lock. In comparison, asynchronous commits allow the transaction to commit and execute without waiting until log flush completes. This removes log flushes from the critical path, but risks partial data loss, as shown in Figure 4. Log group

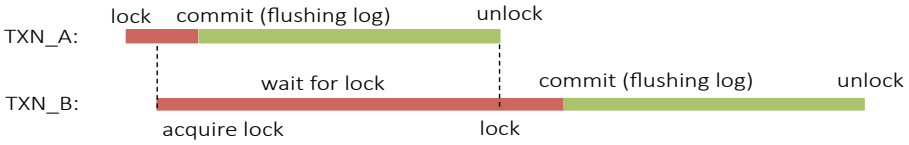


Figure 3: Logging procedure in a disk-based design[14]. Transaction TXN_B has to wait until TXN_A finishes the log commit and releases the lock. Disk I/O is the bottleneck.

commit [27] aggregates several flushes into a single I/O operation to decrease the number of disk accesses, and ELR [17] tries to further reduce overheads by releasing locks before a log record is written to persistent storage. Controlled lock violation [14] is another method enabling unconflicted transactions to continue without first releasing their locks. However, these speculative executions also risk data loss and may result in inconsistencies. Consider two transactions TXN_A and TXN_B, for instance, where TXN_A acquires an exclusive lock to execute an update operation on one record in the database, and releases the lock after the corresponding log record is inserted into the log buffer. Next, TXN_B acquires a lock on the same record, but executes only a read operation, so that a commit record is not needed for TXN_B. The result of TXN_B would be the value written by TXN_A before the log records associated with TXN_A are flushed into persistent storage. If the system crashes at this point, the user may get a value that never existed in the database. Similar issues exist for the two-phase commit protocol used in distributed databases [14].

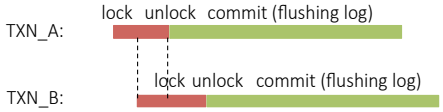


Figure 4: Optimized disk-based solution.

Another factor affecting transaction system performance is lock contention in the lock manager, addressed in recent work with optimizations like SLI [15] and ELR [17]. The improved synchronization support on multicore hardware is also helpful for improving lock manager performance. Taking advantage of such lock manager improvements, this paper focuses on improving the performance-dominant logging subsystem, explained in more detail next.

2.3 Redesigning Logging for NVRAM

The emergence of NVRAM and its potential use for replacing slow disk offers new opportunities for improving logging performance. Key to efficiently using NVRAM, however, is a detailed understanding of the software bottlenecks involved with obtaining durability. Replacing slower hard disk with faster flash drive has already been shown to provide performance advantages [7, 21], but experimental results in [17] also show that even the fastest flash drives still cannot eliminate overheads due to buffer contention, OS scheduling, and software overheads inherent in systems. This will hold even more when using NVRAM with its near-DRAM speeds, giving rise to software bottlenecks like resource contention in the centralized log buffer.

This paper investigates cost-effective methods for leveraging NVRAM, in the logging subsystem. We present a redesign of the logging component of a transaction system, with the goal of alleviating software-related performance bottlenecks when using NVRAM while still providing strong consistency and durability guarantees.

3. NVRAM DATABASE RESTRUCTURING

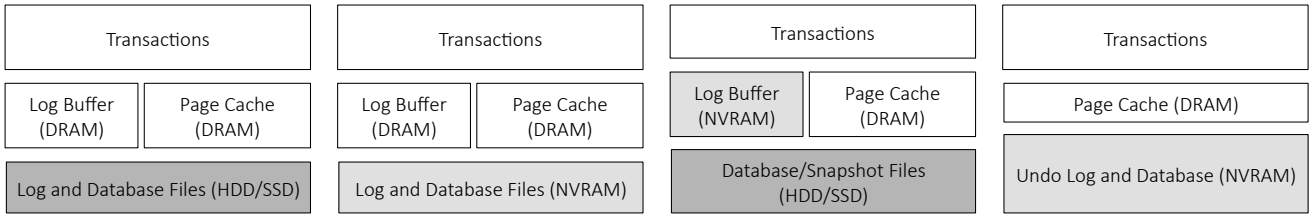
3.1 Candidate NVRAM Uses

As shown in Figure 5(a), a disk-based database system has two main DRAM components: page cache and log buffer. The page cache hosts a fraction of data pages populated with the records stored in database tables. The log buffer stores log records for transactions. Each transaction may generate multiple log records, and once the transaction commits, all corresponding log records must be flushed to log files on persistent storage. This design is appropriate for block devices like hard disks and SSDs, since the centralized log buffer and page cache hosted in DRAM encourage sequential disk access to alleviate disk I/O bottlenecks. Yet, as faster NVRAM devices become available, with their access latencies close to that of DRAM [26, 2], it becomes important to rethink this traditional disk-centric design.

A straightforward way to accelerate the performance of a transaction system is with the *NV-Disk* approach shown in Figure 5(b). This approach replaces the entire disk with NVRAM. Since the speed of NVRAM is much higher than that of hard disks and flash drives, significant performance improvements can be gained, without the need to modify the transaction system's implementation. However, the approach has drawbacks. Notably, the replacement of high capacity low-cost disks with costly NVRAM fails to leverage NVRAM's byte addressability, and its interactions with NVRAM via file system APIs suffer from software overhead. It is also not cost-effective for typical server systems, since the cost of NVRAM will remain substantially higher than that of hard disks and SSDs.

The alternative *NV-Logging* solution explored in our work continues to use disks to store database tables, but uses NVRAM only to maintain logs. Specifically, log records are stored in persistent NVRAM, database tables continue to be cached in DRAM for performance reasons, but their original data source and snapshots are stored on high capacity, low cost disks for recovery and long term use. As shown in Figure 5(c), the *NV-Logging* approach exploits the non-volatility characteristic of NVRAM to overload the functionality of the log buffer, but does not incur the potentially high cost of *NV-Disk*. Note that the approach also applies to today's in-memory systems like RAMCloud [24] and in-memory databases [1], where all of the data pages are in DRAM, and logging is used to back up update operations.

As shown in Figure 5(d), a more aggressive approach is *NV-WSP*, in which all database pages are hosted in NVRAM. Since all updated data will be persistent without being flushed to disk, the redo logs are no longer needed, but undo logs are still required for transaction aborts. Due to the slower speed of the state-of-the-art NVRAM technologies compared



(a) Traditional Design (b) NV-Disk (c) NV-Logging (d) NV-WSP

Figure 5: Candidate ways to use NVRAM in a transaction system.

to DRAM, however, bridging this performance gap still requires caching pages in DRAM. The resulting scale-out, non-volatile memory design may offer sufficient capacity for today's petabyte sized data warehouses [18], but its high cost suggests the need for alternative, more cost-effective approaches [19] like the *NV-Logging* outlined above.

3.2 Performance Gap Analysis

It is clear that using NVRAM can improve the performance of transaction systems, but gaining such improvements implies the need for additional hardware expenses due to the higher cost of NVRAM compared to traditional disk-based storage.

To make these performance opportunities concrete, we run the Shore-MT [5] transaction system with the TPCC benchmark. We evaluate transaction throughput with different memory and storage configurations. Details about the experimental environment are described in Setup-A in Section 6.1. Using the memory-based *tmpfs* file system, with NVRAM I/O latency conservatively set to be 5 microseconds following an approach similar to that used in [17], we compare the four alternative configurations shown in Figure 5. As shown in Figure 6, *all-in-HDD/SSD* is the default configuration, where both the log and the database file are stored on the ext4 file system on hard disk. For the case of *all-in-NVRAM*, the log and database files are stored in the NVRAM-based *tmpfs* file system. Option *db-in-HDD/SSD, log-in-NVRAM* stores the database file on disk and the log in NVRAM, and option *db-in-NVRAM, log-in-HDD/SSD* stores the database file in NVRAM and the log files on disk. File system APIs are used in all cases, to avoid the modifications of the transaction system's implementation.

Figure 6 shows the throughput of the TPCC benchmarks with varying numbers of threads. As expected, performance of the options hosting log partitions in memory is dramatically better than those hosting log partitions on disk. More interestingly, the throughput gained with option *db-in-HDD, log-in-NVRAM* comes close to the throughput of the much more expensive *all-in-NVRAM* option, reaching an average 74% of the performance of *all-in-NVRAM*. The performance gap is further reduced with *db-in-SSD, log-in-NVRAM*, reaching an average 82.5% of *all-in-NVRAM*'s performance. Similar performance trends are seen when we vary I/O latency from 5 to 50 μs and run the TATP benchmark. Note that TPS stops increasing beyond 12 threads on our machine (it has 16 logical cores in Setup-A). This lack of increase is also due to increased logging overheads, as evident from the fact that as we increase the page cache size to 12 GB (the working set size of TPCC is 9 GB), essentially making the transaction system work as an 'in-memory' system, throughput does not increase further. We will discuss the logging overheads in detail in the following section.

3.3 Logging Overheads

To further analyze intrinsic behaviors in transaction systems and reason about the performance gap reported in Section 3.2, we use *perf* [4] to collect stack traces and report on the timing behavior of Shore-MT's different components. We break down the software stack into five categories: (1) log operations, such as flushing log buffer, populating log record, etc.; (2) log contention, such as contention for buffer allocation, lock overhead on log buffer, etc.; (3) lock manager, including database locking, lock management, etc.; (4) DB operations, including index lookup/update, fetch record, etc.; and (5) Others. As shown in Figure 7, log operations take 60.73 - 85.46% of the total time as logs are placed on HDD/SSD, demonstrating that the I/O bottleneck dominates overall performance. This situation becomes worse if we also place database files on disk, not shown in Figure 7. In contrast, when placing logs into NVRAM, log operations perform much better, but log contention overhead gradually increases as we increase the number of threads. The overheads of log operations and log contention occupy 34.1 - 39.19% of the total execution time, while the overhead in the lock manager is almost eliminated with SLI and ELR.

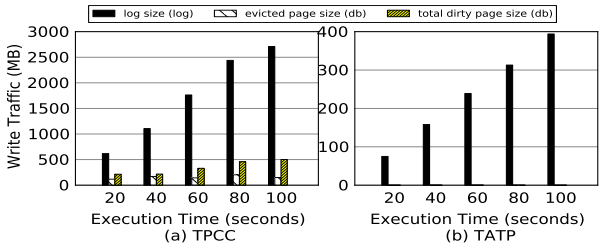


Figure 8: Write traffic of TPCC-mix and TATP-mix benchmark running in 20 - 100 seconds.

The measurements above show that the logging component strongly influences transaction performance due to log buffer contention and log-induced lock contention. Furthermore, logging also dominates the I/O traffic in transaction systems. We demonstrate this fact by collecting the write traffic of the log buffer and page cache in Shore-MT over different time intervals, varying from 20 to 100 seconds. The log size represents the traffic written to log partitions via the log buffer. The evicted page size indicates the traffic caused by the page evictions from the page cache during the specific time intervals. As shown in Figure 8, the logging traffic is 2.88 - 5.43x larger than the total size of dirty pages for TPCC. For TATP, the log size is dramatically larger than that of dirty pages (about several MB).

Given the importance of logging, we argue that applying NVRAM to logging is a good choice in terms of cost-performance. This is not a trivial endeavour, however, as de-

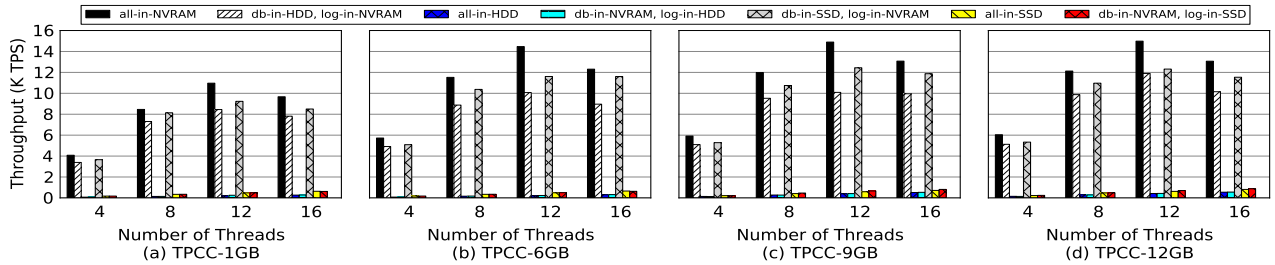


Figure 6: Throughput of the TPCC-mix workload with varied page cache size. The average throughput of *db-in-HDD/SSD,log-in-NVRAM* is up to 74%/82.5% of the throughput of *all-in-NVRAM*.

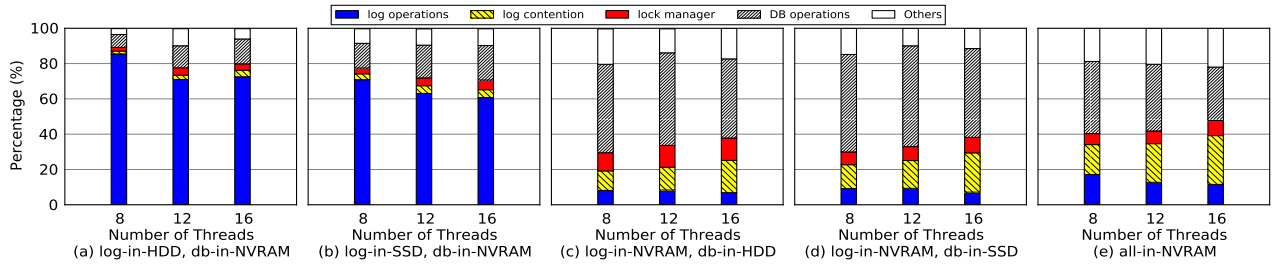


Figure 7: Time breakdowns for TPCC-mix benchmark.

scribed in the remainder of the paper where we discuss optimizations necessitated by NVRAM's non-DRAM-like characteristics. For clarity, we refer to *all-in-NVRAM* as *NV-Disk* in the rest of the paper.

4. LOGGING IN NVRAM

4.1 Bottlenecks Shifted from I/O to Software

Replacing traditional disk with faster flash devices and NVRAM can reduce I/O latency and improve throughput, but performance challenges formerly posed by disk-related overheads shift to the software stacks being used.

According to the analysis in [6], the number of instructions executed for a simple database transaction ranges from 20K to 100K based on transaction logic, database structures, its implementation and compiler optimizations. The number of I/O operations involved in each transaction is typically in the range from 4 to 10. On a fast system, each transaction executes 50K instructions and 4 I/O operations (two for database IOs, two for log writes) on average. With the optimistic assumption of 1 instruction per CPU cycle on average, the execution time of transaction logic will be 0.02 ms on a 2.4 GHz mainstream server processor. Since it takes considerably more time to commit the transaction to hard disk (e.g., 10 ms) or flash device (e.g., 0.2 ms), the resulting I/O latency is an order of magnitude higher than the execution time of the transaction logic. On the other hand, even for NVRAM accesses conservatively estimated to be 4 - 8 times slower than DRAM writes (i.e., a 60 ns latency), the latency of I/O operations to NVRAM is smaller than that of the transaction logic itself, thus demanding improvements in the software stack of transaction systems.

For transaction systems, a known cause of software overhead is the centralized log buffer, inhibiting the parallel execution of multiple in-flight transactions [17, 29]. When maintained in NVRAM, the log buffer can be accessed as a block-based device via file system interfaces within *NV-Disk*, or as directly addressable memory within *NV-Logging*. To understand the performance tradeoffs seen for both, a statistical analysis with the TPCC benchmark shows log object sizes typically range from ~64 bytes to 6 KB. Further, with *group-commit* committing log objects in batches, flush size reaches 64 - 754 KB. Figure 9 illustrates the overhead comparison when using the file system vs. memory APIs for different log object sizes. Compared to the *memcpy* operation for a normal log object, the latency of using *tmpfs* with its fils system APIs is 49 - 154x larger, and using *mmap* with synchronous (MS_SYNC) writes is 5 - 18.3x larger. For larger flushing sizes, *tmpfs* and *mmap* still perform 5 - 9.5x and 1.4 - 2.9x slower than *memcpy* operations respectively. These experimental results demonstrate that bypassing the file system API can reduce software overheads dramatically, thus motivating our *NV-Logging* solution described next.

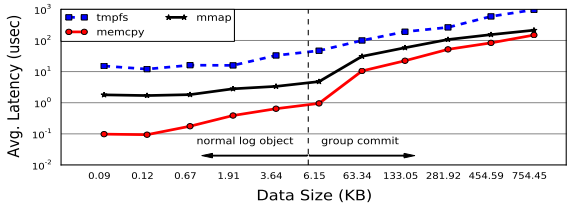


Figure 9: Write latency comparison of file System and memory APIs.

4.2 Decentralized Logging

The experimental results shown in prior sections motivate *NV-Logging*'s use of NVRAM as memory, exploiting both its byte-addressability and persistence characteristics, while continuing to use file system APIs for the conventional hard disk drives or flash devices employed to store cold data for backup purposes. To avoid bottlenecks in the centralized log buffer, *NV-Logging* leverages per-transaction logging for decentralized operation. With this solution, because each transaction maintains its own private log buffer for storing log records, their creation, population, and persistence properties can be obtained in a scalable manner. We continue to use global LSN to track the order of logs, but since

NV-Logging does not need log partitions and no offset is required, assigning a LSN to a log and updating its value are straightforward, efficient operations.

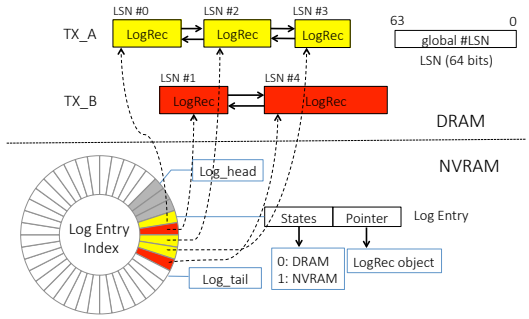


Figure 10: NV-Logging system design.

The per-transaction log buffer depicted in Figure 10 is structured as a set of log entries organized as a circular buffer in NVRAM. Each log entry consists of one state bit and a 4-byte pointer to the generated log object. A log object is first created in DRAM, with its state bit set to 0. Once it is flushed into NVRAM and becomes persistent, the state bit is set to 1. The pointer *log_head* always points at the start entry of the circular buffer. It will point to a new location after log truncation (Section 4.5). The pointer *log_tail* always points at the first available log entry of the circular buffer, and moves forward after a log entry is allocated.

Each transaction manages its own log records, including maintaining their global order, as shown in Figure 10. All log objects are tracked via a doubly linked list, but rather than storing its adjacent log records' LSNs and using LSN to calculate file offsets to locate log object, NVRAM's byte addressability affords the use of pointers to retrieve log objects for transaction abort and recovery.

Transaction conflicts are handled with fine-grained, row-granularity locks. Additional optimizations enabled in our experiments include SLI and ELR. The problems of data loss and inconsistent results caused by these optimizations, as discussed in Section 2.2, can be avoided with the logging persistence techniques described in Section 4.3.

To summarize, our logging implementation reduces software overheads (1) by using per-transaction logging, thus decentralizing the log buffer and reducing potential lock contention, and (2) by simplifying certain implementations in lieu of the logging's straightforward structure and use.

4.3 Logging Persistence

Like the persistence primitives discussed in Mnemosyne [28] and NVHeap [9], *NV-Logging* leverages hardware primitives and software instructions for writing data persistently and for providing consistency guarantees. Compared to these works, however, *NV-Logging* benefits from simplified persistence and consistency mechanisms due to the straightforward nature of its data structures: the log object, and a few variables (i.e., global LSN, *log_head*, *log_tail* and log entry). The log object, for instance, is initially created in DRAM, and is only later made persistent via its placement into NVRAM, thus benefiting from DRAM's high performance. Further, until it is cleaned within log truncation, it will not be updated after it has been populated, thus avoiding exposure to the higher write latency of NVRAM.

Consistency issues with NVRAM-resident data arise from the fact that today's processor cache hierarchies are designed for DRAM rather than NVRAM. Specifically, with caching, writes may be reordered, resulting in potential inconsistencies in the presence of application and system failures. To avoid the need for additional cache-level hardware support [10] or the need to replace volatile with non-volatile caches [32], *NV-Logging* takes advantage of well-known hardware instructions to implement its consistency and persistence mechansims: (1) the *clflush* instruction supported in most processors flushes specified cache lines out to memory; (2) the *mfence* instruction is a hardware memory barrier that enforces the ordering of memory operations. An alternative solution is whole-system persistence [23], which can make the entire memory persistent upon failure. With hardware that has sufficient backup power sources, *NV-Logging* can also achieve high performance with a *flush-on-failure* policy. Such an approach complements this paper's goal to create cost-effective ways of using NVRAM.

NV-Logging uses *clflush*, but not *mfence*, as the latter is not required because append-only logs never update a log object once it has been written. Further, by first creating a log object in DRAM, NVRAM writes are necessary only when the object is fully populated after all of its data structures have been assigned with proper values. Fully populating log object content in DRAM before flushing it to NVRAM simplifies matters, as only completed log object with fixed size is flushed, thus only a single *clflush* or *write-through store* instruction is needed to complete each such write. Finally, previous work has argued that in-memory data copying can cause high levels of memory pressure for large database transactions [8]. This problem can be addressed by multiplexing log objects, an approach that is feasible with the less than tens of log objects generated by each transaction on average, based on our statistics obtained from well-known transaction workloads.

To obtain high performance, *NV-Logging* offers two persistence policies, *flush-on-insert* and *flush-on-commit*.

flush-on-insert: it is similar to *in-place update*, but the log object is initially allocated from volatile memory. As stated earlier, once the log object is entirely populated, it is flushed into the location in NVRAM to which its corresponding index entry points. To ensure consistency, the *clflush* instruction is called to make sure that all log data is in NVRAM and no longer resides in volatile cache or DRAM. The state bits in both the log object and log index entry are set to indicate the log object is persistent. After the transaction commits, all of its log objects in volatile memory are cleared but not deallocated, so that they can be reused to reduce object allocation overhead.

flush-on-commit: the log object is created as in *flush-on-insert*, but instead of flushing the object immediately after it is populated, the log object is asynchronously copied to NVRAM. Such copying is performed by a daemon process that checks the states of DRAM-resident log objects, persists the fully populated ones, and sets their state bits accordingly. Since this may cause delays on transaction commit, when log objects are finally flushed, we first scan the state bit in each log object. If the bit indicates that the object has not yet been asynchronously persisted, persistence is ensured by calling the procedure *flush-on-insert*. By using this policy, persistence overhead is removed from the critical path, without damaging the order of persistent log objects.

In order to reduce the frequency of persistence operations, Pelley et al. propose *Group Commit* [25] that orders persists in batch instead of in transaction or page granularity. However, all transactions from the in-flight batch have to be rolled back or aborted on failure. No such constraint exists for *flush-on-commit*, as only uncommitted transactions have to execute the rollback procedure.

4.4 Transaction Abort

Following the principles introduced in ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [30], the log used in *NV-Logging* consists of both the undo and redo log objects for all of the updated records in each transaction. Undo logs provide necessary data to roll back uncommitted transaction updates and recover the corresponding records to original values. Redo logs contain the records of update operations on data pages in the volatile page cache in case the updates have not yet been persisted when failures or crashes occur, so that these update operations can be replayed. As with ARIES, *NV-Logging* ensures that log objects are persistent before the corresponding values are updated in corresponding volatile data pages, with reduced overheads compared to *NV-Disk*.

In *NV-Disk*, each log object contains the LSN of its previous log object. On transaction abort, the incomplete transaction reverses the updates from the latest log object one by one. This may incur a number of disk seeks. For *NV-Logging*, we use back pointers to obtain previous log objects. For incomplete transactions, all copies of their log objects also exist in volatile memory, although some of them may have been persisted with *flush-on-commit*. In such cases, to maintain consistency, we still persist the unflushed log objects and set their state bits, but then insert an additional log object to indicate that this transaction has been aborted. These log objects for aborted transactions may not be needed, which will be cleaned later in log truncation.

4.5 Log Truncation

Log truncation is needed in part to limit the server's total NVRAM capacity and its consequent cost. It works collaboratively with checkpointing in transaction system. For those logs whose associated data pages have been made persistent, they are not needed for recovery.

With the disk-based solution *NV-Disk*, log truncation works at the granularity of log partitions. The two-level store hierarchy (volatile centralized log buffer and disk) creates a sequential write pattern for transactions logs, but this also means that one partition cannot be truncated until all of its corresponding transaction updates have been persisted. For *NV-Logging*, log truncation is not constrained to partition boundaries. After checkpointing, the *log_head* moves ahead to the checkpoint, and the state bits in the log entries passed in that move are cleared. The memory resources used by log index entries are not reclaimed, as they can be reused. This may waste some memory resource when log object sizes vary, but the total allocated memory size is limited since checkpointing runs periodically. For disk-based solution, checkpointing is activated when the centralized log buffer is full or transaction commits. In *NV-Logging*, a threshold for the number of consumed log entries is defined. Once the threshold is reached, checkpointing will be awakened. As many of the log objects created recently have been become persistent, the logging persistence will not delay the checkpointing. And only the pages associated with these recent logs are involved in the checkpointing, transactions that are not updating to these pages can proceed as normal until next threshold is reached. In addition, dirty page cleaner threads running in the background will traverse the whole page cache and write out dirty pages asynchronously [16]. This is similiar to the *adaptive flushing policy* [20, 22] as dirty page flushing can be interleaved with transaction processing. It is rare that log entries are exhausted, as the log truncation procedure wakes up intermittently and corresponding log entries are reclaimed accordingly.

4.6 Recovery

The redo logs contain the history of all transaction updates since the last checkpoint. With checkpoint or snapshot files, the database can accomplish point-in-time recovery. Each page has a LSN to indicate the transaction that updated it most recently. During recovery, the LSN in the page is compared with the LSN stored in log entries to check if the page needs to be updated with redo logs. With decentralized logging, the challenge is how to rapidly perform in-order system recovery. We use the log entry index for reconstructing the order of logs. As each entry in the structure has a pointer that points to the address of the log object, the object's retrieval is simple, requiring no complex offset calculations or file operations. For pages that were modified by transactions but not made persistent, redo logs are applied to roll the database forward. For modified pages that contain updates but have not been committed, a roll back procedure is executed. Both the roll back and roll forward procedures rely on LSNs to retrieve log objects. In summary, we follow ARIES rules for database recovery, but provide atomic and durable logging while avoiding associated software overheads.

5. IMPLEMENTATION

NV-Logging is implemented based on the latest open-source Shore-MT [5] transaction system, providing ARIES-based logging and recovery.

5.1 Memory Management

As shown in the system design for *NV-Logging* (Figure 10), a set of log index entries are allocated upon system startup. All state bits in the log entries are initially cleared, with pointers set to pre-allocated space in NVRAM. A 64-bit LSN is automatically increased as log objects are inserted. Without the overhead of resource allocation, acquiring a global LSN can be done quickly. The *log_head* always points to the beginning of the circular log buffer structure, and records the LSN of the log object to which this entry points. The *log_tail* always points to the first available log entry, and records the pre-allocated LSN. With the two recorded LSNs, a log entry can be easily located if a LSN is within the range, otherwise an error may occur during LSN allocation and log insertion. All of the data structures include the global LSN, *log_head*, *log_tail*, log entries, with pre-allocated space are in NVRAM. Atomic updates are applied to guarantee consistency.

For data structures in volatile memory like the initial existing log objects, atomic update rules must be obeyed only when interacting with NVRAM, an example being log object flushing. This substantially reduces the overheads of maintaining log objects. Additional reductions are obtained by

avoiding allocation overheads: threads that execute transactions pre-allocate a number of log objects in advance, and these log objects are reused after transaction commits. This reduces the memory footprint of log objects in volatile memory and context switch overheads from allocation calls.

In addition, *NV-Logging* has asynchronous log backup to dump logs from NVRAM to disk for freeing NVRAM space. This is done asynchronously so that it does not affect transaction performance. By storing cold logs on disk, more costly NVRAM space is preserved to maintain logs related to in-flight transactions.

5.2 Consistency and Atomic Updates

Similar to file system inode management, the update order for log object persistence must be maintained if failures occur when a log object is flushed from volatile to persistent memory: object content is flushed first, followed by *clflush*, then the state bit is set to indicate it has been made persist. Violation of this order can result in what appear to be successfully persisted log objects, but with meaningless NVRAM pointers. Note that pointers from one space (i.e., pointers to either volatile or non-volatile memory locations) used in *NV-Logging* never point to addresses in the other space. Specifically, the pointer in each log index entry always points to a pre-allocated non-volatile log object, and its state bit indicates whether the value of the log object is valid or not. This design reduces the complexity of consistency maintenance and avoids the happening of dangling pointers. Atomic updates are guaranteed with small, atomic eight-byte persistent writes offered by hardware (for pointer updates, LSN updates, and etc.), along with the state bits in log objects and log entries to detect failures during updates.

6. EVALUATION

This section evaluates the performance of *NV-Logging*, as compared to *NV-Disk* and *Distributed Logging* [29]. A challenge in this evaluation is the lack of NVRAM hardware, with previous work typically resorting to the use of simulation. In contrast, following recent work like [12], we evaluate our solutions on emulated NVRAM hardware, using two different emulations, described in detail as Setup-A and Setup-B in Table 1. First, experiments with OLTP benchmarks are run in an environment approximating expected NVRAM performance (Setup-A). In order to further verify and examine our solutions, we then run experiments on a hardware platform precisely emulating future NVRAM memory developed by Intel Corporation (Setup-B). Using the latter, we also compare *NV-Logging* with an alternative solution PMFS file system that specifically designed to efficiently exploit NVRAM's benefits [12]. The purpose of this comparison is to assess the value of re-implementing components of database systems with solutions like *NV-Logging* vs. reusing systems' existing file-based interfaces layered on top of NVRAM, as done by PMFS.

6.1 Experimental Setup

Setup-A: To emulate NVRAM's slower writes relative to DRAM, we add latency to NVRAM writes. Since NVRAM writes may be cached in volatile processor cache, we add these delays after executing the *clflush* instruction. We do not add any latency to NVRAM reads as the asymmetric read-write performance of NVRAM indicates that its read performance will be close to DRAM's. We use the *rdtscll* for timing and then compute the latency for write slowdowns.

Setup-A	
CPU	Intel Xeon X5550, 2.67 GHz
CPU cores	4 (16 with Hyper-threading)
Processor cache	32KB/32KB L1, 256KB L2, 8MB L3
DRAM	48 GB
NVRAM	emulated with slowdown additional latency varies from ∼1 to 8 μs
Disk	512 GB HDD two 128 GB OCZ-VERTEX 4 SSDs
Operating system	RHEL 6, kernel version 2.6.32
Setup-B	**Intel's PMEP**
CPU	Intel64 Xeon-EP platform, 2.6 GHz modified CPU & custom firmware
CPU cores	16 (Hyper-threading disabled)
Processor cache	32KB/32KB L1, 256KB L2, 20MB L3
DRAM	64 GB (DDR3 Channels 0-1)
NVRAM	256 GB (DDR3 Channel 2-3) configurable latency
Disk	4 TB HDD
Operating system	Ubuntu 13.04

Table 1: Experimental setup.

Setup-B: Intel's Persistent Memory Emulation Platform (PMEP) is a system-level performance emulator for persistent memory, offering configurable memory latencies and bandwidth. The platform provides up to 4 DDR3 Channels, and partitions memory between DRAM and emulated NVRAM, the former is unmodified and the latter is modified as per expected NVRAM properties. PMEP models latency and bandwidth by introducing additional stall cycles and using a programmable feature in the memory controller. Details about the emulation appear in [12].

Workload	Scale factor	Data size	Transaction type
TPCC	70	9 GB	Mix
TPCB	1000	11 GB	Account updates
TATP	1000	15 GB	Mix

Table 2: Benchmarks used in experiments.

Benchmarks: Shore-Kits is an open-source suite of OLTP benchmarks implemented on top of Shore-MT storage manager. Our experiments use TPCC, TPCB, and TATP benchmarks as shown in Table 2. TPCC models a retailer's online transaction processing database receiving orders, payments, and deliveries for items. We populate a database of 70 warehouses with 9 GB data size. The TPCB benchmark simulates a banking workload against a database that contains branches, tellers, and accounts records. A database with 1000 warehouses is populated, its size is 11 GB. The TATP benchmark models a home location registry database used by a mobile carrier. We populate its database with 1000 warehouses, of 15 GB size.

Configurations: the page cache size of Shore-MT is configured as 6 GB, and the default page size is 8 KB. For *NV-Disk*, the log buffer is set to 80 MB by default, the quota for log partitions is 3 GB for 8 partitions. For *NV-Logging*, the threshold of consumed log entries for checkpointing is 80 K by default, its average log data size is smaller than 80 MB. In Setup-A, we vary the relative slowdown for NVRAM up to 8x compared to DRAM, and vary NVRAM write latency from DRAM's latency to 500 ns in Setup-B. All benchmarks

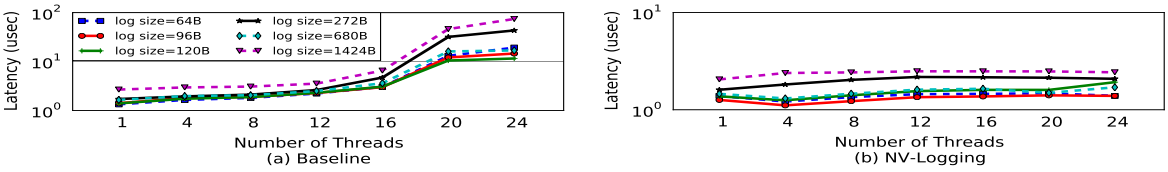


Figure 11: Average latency of log insertion.

are executed 5 times, each of which lasts 30 seconds, and the average throughput is reported.

Schemes	Centralized log buffer	FS APIs	Sync/Async Commit
NV-Disk	✓	✓	Async
NV-Disk+decentralized-logging	×	✓	Async
NV-Logging+flush-on-insert	×	×	Sync
NV-Logging+flush-on-commit	×	×	Async
Distributed Logging [29]	×	✓	Async

Table 3: Schemes for the comparison.

As shown in Table 3, we compare these schemes to understand the impacts of different designs on transaction throughput. In *NV-Disk+decentralized-logging*, the log buffer is decentralized, but logs are still placed on file system via the same way as implemented in *NV-Disk*. In *NV-Logging*, we evaluate both the logging persistence mechanisms *flush-on-insert* and *flush-on-commit*. For all the schemes, we use 'OL' (Optimized Locking) to represent enabled SLI and ELR.

We also compare *NV-Logging* with the state-of-the-art *Distributed Logging* [29], in which both log buffers and log partitions are distributed. Without durable processor cache support, this method has to employ memory fences to prevent reordering of stores for multicore processors. Modifications on other components in transaction system are also required to process recovery, transaction abort and forward processing with the distributed logging. In our experiments, we use 16 logs, each with 192 MB NVRAM-based log buffer for transaction-level partitioning.

6.2 Log Insertion Performance

Using Setup-A, this section evaluates the scalability of the performance-dominant logging component, not yet considering the effects of other components like the lock manager. NVRAM latency is first configured to be the same as DRAM. Figure 11 shows the average latency of log insertion with a varying number of threads. Measurements are based on collected values for the typical sizes of log objects generated in OLTP benchmarks: our statistics indicate these sizes range from ∼64 bytes to several KBs.

As shown in Figure 11(a), the average latency of log insertion increases dramatically (up to 74.3 μs) with an increasing number of threads in *NV-Disk*. This is due to log contention in the centralized log buffer. In contrast, for *NV-Logging*, the average latency of log insertion remains at a consistent level (up to 2.5 μs), as shown in Figure 11(b). There is no resource allocation overhead and correlated contention in *NV-Logging*, as each transaction only needs to obtain a global LSNs for its log objects. With LSNs, transactions can easily locate the corresponding log entries.

6.3 OLTP Workloads

With Setup-A, we next vary the write latency of NVRAM to show how overall transaction performance will be impacted with *NV-Disk*, *NV-Logging* and *Distributed Logging*. In these experiments, database files are always placed on the ext4 file system based on disk. In *NV-Disk*, we place the log partitions on *tmpfs*, whereas in *NV-Logging*, we place the log objects directly into NVRAM, bypassing file system APIs, and no centralized log buffer is required.

TPCC: Figure 12 shows the throughput of Shore-MT, for varying NVRAM write latencies and numbers of threads. Increasing the number of threads, *NV-Logging* shows increased benefits, performing 1.62 - 2.72x more throughput than *NV-Disk*. Note that overall transaction throughput will also be affected by other transaction system components like the lock manager, but such optimizations and improvements are out of scope for our work. We enable both SLI and ELR to eliminate overheads in the lock manager.

Additional experiments evaluate the effects of log buffer size on overall performance in *NV-Disk*. We increase the log buffer size from 80 MB to 256 MB, the latter being close to its maximum limit, as log buffer size depends on the number of partitions and the quota for log files. Experimental results show that the TPS is only increased by 1.8 - 13.6%. Since this demonstrates that enlarging the centralized log buffer is not a good method for performance improvement, other experiments in this paper forego the use of larger log buffers.

Figure 12 also demonstrates that *NV-Disk + decentralized-logging* performs worse than *NV-Disk*. This is because while the decentralized log buffer design could perform better than the centralized log buffer, as resource allocation and lock contention overheads can be avoided, this also hurts the ability of grouping logs, and increases the frequency of disk accesses. Leveraging only the decentralized log buffer, therefore, cannot improve the throughput of *NV-Disk*, particularly for update-intensive workloads.

Additional measurements examine the performance of *flush-on-insert* and *flush-on-commit*. As NVRAM write slowdown is increased, the performance of *flush-on-insert* drops slightly, while *flush-on-commit* performs at a consistent level. If NVRAM write performs 4 - 8x slower than DRAM, the *flush-on-commit* performs 7.89 - 13.79% better than *flush-on-insert*, since log objects can be persisted asynchronously.

Compared to *Distributed Logging*, *NV-Logging* has 13.8 - 26.81% higher TPS. This is because *NV-Logging* uses a simpler design: (1) without calling memory barriers and fences frequently to maintain the ordering for distributed logs, and (2) using a simple lookup procedure. A more detailed analysis of software overhead appears in Section 6.5.

In the following sections, we only present the results with larger numbers of threads due to space limitations. Experiments with smaller numbers of threads show similar trends.

TATP: Figure 13 (a) and (b) illustrate the throughput of TATP with mixed workloads. *NV-Logging* performs 1.12 - 1.62x better than *NV-Disk*. With SLI and ELR enabled, the performance of both *NV-Logging* and *NV-Disk* are increased. *NV-Logging* still processes 1.10 - 1.38x more TPS than disk-based solution. As expected, *NV-Logging* with *flush-on-commit* performs better than with *flush-on-insert*

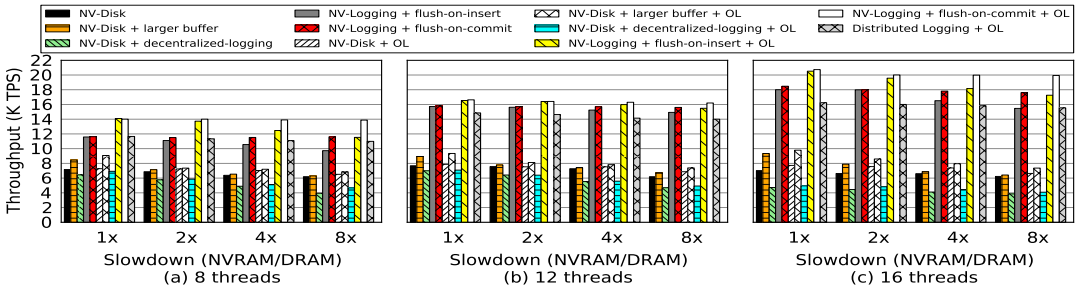


Figure 12: Throughput of TPCC benchmark with varied slowdown configurations. *NV-Logging* performs 1.62 - 2.72x better than *NV-Disk*, even when SLI and ELR are enabled.

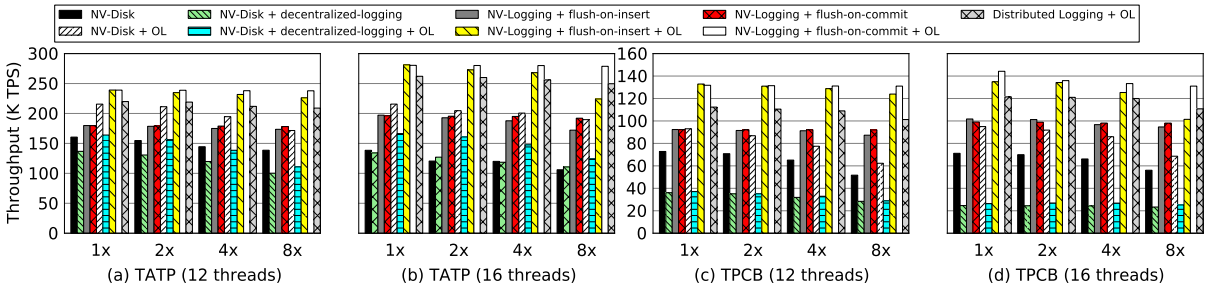


Figure 13: Throughput of TATP and TPCB benchmark with varied slowdown configurations. *NV-Logging* performs 1.10 - 1.62x and 1.26 - 1.99x better than *NV-Disk* respectively, even when SLI and ELR are enabled.

by up to 24.32%. *NV-Logging* performs 8.94 - 10.97% better than *Distributed Logging*. Overall, the results with TATP show similar trends as those with TPCC.

TPCB: when running TPCB, as shown in Figure 13 (c) and (d), the performance trends are similar to those in other two benchmarks. Throughput is increased by 1.26 - 1.69x with *NV-Logging*, as compared to *NV-Disk*. With SLI and ELR enabled, *NV-Logging* performs 1.43 - 1.99x and 1.15 - 1.21x better than *NV-Disk* and *Distributed Logging* respectively. As we increase NVRAM slowdown, *flush-on-commit* can perform up to 29.18% better than *flush-on-insert*.

In summary, *NV-Logging* improves transaction throughput, as compared to *NV-Disk* and *Distributed Logging*, particularly for update-intensive transactions. When NVRAM writes are much slower than DRAM writes, *flush-on-commit* performs better than *flush-on-insert*.

6.4 Experiments on Intel's PMEP

To further evaluate the performance of *NV-Logging*, we redeploy Shore-MT and its Shore-Kits benchmarks on Intel's PMEP, described in Setup-B. This platform has been used to evaluate the system-level performance of persistent memory software, including for the PMFS file system expressly developed to exploit NVRAM's byte addressability [12].

We leverage PMFS in our experiments by using it to maintain log files for *NV-Disk*. We also modify *NV-Logging* and *Distributed Logging* with *libnuma* interfaces, so that NVRAM can be allocated from the persistent memory node in PMEP. As shown in Figure 14 (a) and (b), *NV-Logging* performs 1.21 - 3.17x more TPS than *NV-Disk* with PMFS. For update-intensive workloads as shown in Figure 14 (c) and (d), the throughput of TPCB increases by 3.86 - 6.71x with *NV-Logging*, compared to *NV-Disk* with PMFS. With SLI and ELR enabled, *NV-Logging* performs 4.45 - 7.95x better than *NV-Disk*. Compared to *Distributed Logging*, *NV-Logging* increases TPS by 11.9 - 20.4%. Further, transaction performance does not drop dramatically as NVRAM latency is increased from that of DRAM to 500 ns, matching the trends shown in [12]. The reason for the lack of sensitivity to NVRAM latency is that software overheads are the primary performance determinants.

6.5 Software Overhead Analysis

As shown in Figure 7, logging overheads increase as we place log partitions on NVRAM, while locking bottlenecks in the lock manager are reduced dramatically with SLI and ELR enabled. Compared to *NV-Disk*, *NV-Logging* decreases the log operations overhead from 11.55% to 3.2%, and reduces the log contention overhead from 27.64% to 5.14% (Figure 15). The execution time on DB operations is greatly increased as more transactions are processed. This causes the slight increment in lock manager overhead, but the overall performance is improved.

As shown in Figure 15, *NV-Logging* and *Distributed Logging* can both reduce log contention overhead. However, *NV-Logging* has a simpler design that requires no memory barriers and fences, it reduces the overhead of maintaining the ordering of logs, and also the overhead of log lookups. Figure 15 demonstrates that log operations overhead in *Distributed Logging* is 10.65%, larger than *NV-Logging*'s, while their log contention overheads are similar. Also note that in order to deploy distributed logging in transaction systems, other techniques are needed to solve associated problems like cross-log aborts, imbalanced log space utilization, and log ordering. We believe that our simplified solution can obtain comparable performance with less complexity.

6.6 Cost-Effectiveness Analysis

The cost-effectiveness analysis depicted in Figure 16 illustrates dramatic differences in the potential cost/performance for the different options. The analysis uses overall throughput to represent performance, with costs including the memory and storage cost for hosting all relevant data for the workloads. The prices used for DRAM, flash, and disk are

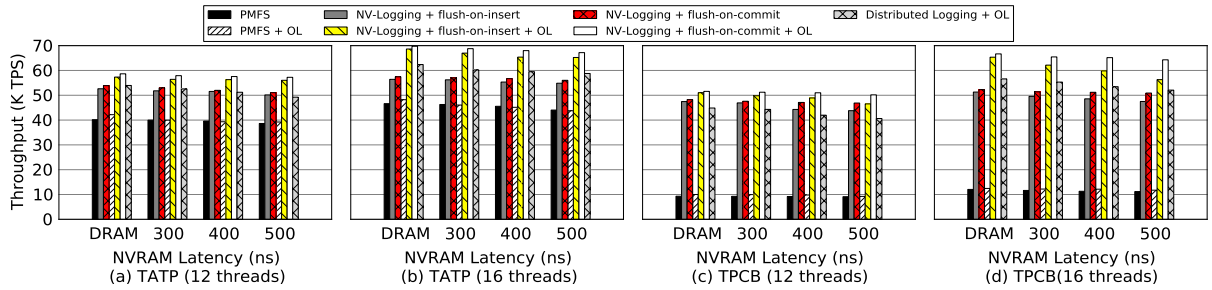


Figure 14: Throughput of TATP and TPCB benchmark running on Intel's PMEP. *NV-Logging* performs 1.21 - 3.17x and 3.86 - 7.95x better than *NV-Disk* with PMFS respectively, even when SLI and ELR are enabled.

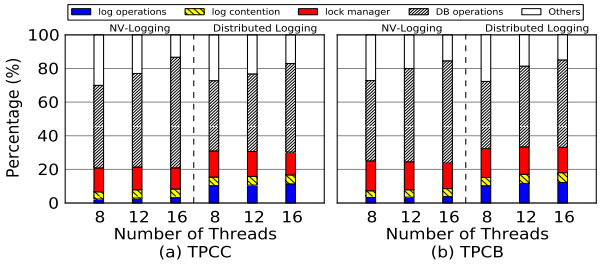


Figure 15: Time breakdowns for TPCC and TPCB with NV-Logging and Distributed Logging.

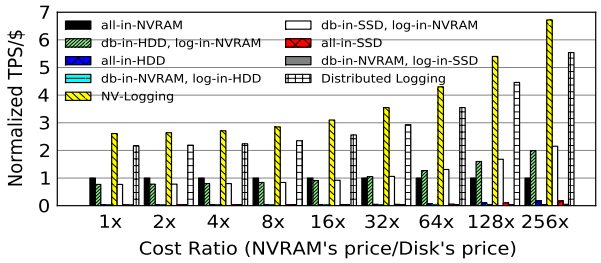


Figure 16: Cost-effectiveness analysis. It shows the normalized TPS/$ for TPCC benchmark, taking *all-in-NVRAM* as the baseline.

$5.5/GB, $0.7/GB, and $0.05/GB, respectively [3, 31]. Concerning NVRAM, since actual NVRAM devices are not yet on the market, we follow the assumptions in Kim et al. [19], who conservatively assume NVRAM device to be 24x more expensive than HDD, based on expert opinions and their investigations. To strengthen our analysis, we explore variations in the cost ratio of NVRAM to HDD, from 1 to 256.

As depicted in Figure 16, *NV-Logging* offers the best TPS/$ compared to other schemes. With *NV-Logging*, we gain 2.61 - 6.72x more TPS/$ than the baseline *all-in-NVRAM*. This illustrative analysis shows that *NV-Logging* is a cost-effective solution, even when NVRAM's cost reaches the same level as that of disk devices (which is highly unlikely). With the same amount of NVRAM used for logging, *NV-Logging* performs 21.13% more TPS/$ than *Distributed Logging*, because of the throughput improvement by *NV-Logging* as described in Section 6.3. Similar trends are seen within TATP and TPCB workloads.

6.7 Discussion

Contention on the centralized log buffer and log-induced lock contention contribute a significant portion to transaction execution times. These software overheads become conspicuous when replacing relatively slow disks with fast NVRAM. The consequent need for restructuring certain transaction system components, in particular the logging subsystem, is shown important by the experiments described in this section. Experiments also show that our proposed solution, *NV-Logging*, reduces these software overheads dramatically, including by using per-transaction logging to exploit NVRAM's byte-addressability. Further, addressing the performance gap between DRAM and NVRAM, for slower NVRAM, we show that *flush-on-commit* offers better performance than *flush-on-insert*, as the former's asynchronous nature bypasses persistence overheads. For logging that generates data in an orderly manner, the persist barrier used to enforce persist order can be simplified or avoided. This could further reduce the cost on persistence and consistency.

7. RELATED WORK

Disk-based Solutions. Assuming traditional memory hierarchies, recent studies have created innovative software to reduce overheads. The Early Lock Release [11, 17] scheme is based on the observation that a transaction's locks can be released before the corresponding log records are written to disk. Johnson et al. [17] identify four bottlenecks related to write-ahead logging: log buffer contention, lock contention, I/O delay, and excessive content switching. Flush pipelining is proposed to avoid context switches when transactions commit. Further performance improvements can be obtained with group commit logging [27], where multiple requests are aggregated into one log flush, thereby reducing I/O overheads. Our work differs from all such efforts in its complementary focus on non-traditional memory hierarchies, for systems able to leverage future NVRAM memory technologies offering both byte addressability and non-volatility.

Flash-based Solutions. Recent studies [7, 21] have shown the performance benefits brought by replacing hard disks with faster flash drives. Interestingly, the experimental results from these studies [17] show that even when using the fastest flash disk drives, one cannot eliminate all of the software overheads associated with logging, such as those due to buffer contention and OS scheduling. Extrapolating from those studies, we posit that such overheads become even more dominant for faster NVRAM.

NVRAM-based Solutions. As NVRAM is nearing its market deployment, researchers have begun to study its use in database systems. Pelley et al. [25] propose to leverage NVRAM as main memory to host all of the data sets for in-memory databases. Our concern with this solution is its practicality, in terms of its cost-effectiveness not considered in that work. We demonstrate those concerns with a cost-effectiveness study evaluating alternative ways to employ NVRAM in transaction systems. Wang et al. [29] propose

distributed logging, in which both log buffers and log partitions are distributed to alleviate logging overheads, but this design has associated issues such as cross-log abort, recovery from distributed partitions and imbalanced log space utilization. With *NV-Logging*, we offer a simplified design exploiting NVRAM's both byte-addressability and non-volatility, and gaining improved performance through reduced overheads. Fang et al. [13] exploit SCM (Storage Class Memory) to reduce the logging overheads of transaction systems. Their approach is to use SCM as cache for disks, but this solutions still suffers from the software overheads introduced by structures like the centralized log buffer. In comparison, we propose a decentralized logging approach that avoids these overheads and provides a more scalable logging solution. With both software (lower-level interfaces) and hardware support, Coburn et al. [8] implemented atomic write operations to exploit the benefits of parallelism of NVRAM-based storage. We can take advantage of their solutions, applying them to the specific case of log buffer access in transaction systems.

8. CONCLUSION

This paper describes cost-effective ways to use NVRAM technology to improve the performance of transaction systems. By implementing these solutions and evaluating them with emulated NVRAM hardware, detailed insights are provided on how to best leverage NVRAM in future systems. In particular, we show that it is not necessary or cost-effective to replace all disk with NVRAM to gain high transaction throughput. Instead, it suffices to use NVRAM to hold the transcaction system's logs, resulting in performance comparable to that obtained for in-memory databases.

When using NVRAM vs. disk-based logging, however, careful attention must be paid to the software overheads involved in logging, in contrast to previous implementations benefiting from the relatively slow nature of disk-based devices. This inspires us to re-design logging – *NV-Logging* – to use per-transaction logging that efficiently exploits the byte-addressability of NVRAM and supports highly concurrent operation. Experimental results with the OLTP benchmarks show that this design substantially outperforms previous disk-based implementations – *NV-Disk* – by up to 6.71x.

Acknowledgements

We thank anonymous reviewers for their feedback and comments. We would like to thank Ling Liu, Ada Gavrilovska, Xuechen Zhang for discussions, and Sanjay Kumar for his support on Intel's persistent memory server. We also thank Alexander M. Merritt for proofreading an earlier version of this manuscript. This work was supported in part by the Intel URO program on software for persistent memories, and by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

9. REFERENCES

[1] In-memory Databases. http://en.wikipedia.org/wiki/In-memory_database.
[2] Micron Technology, Inc. http://us.micron.com/products-support/phase-change-memory.
[3] Newegg. http://www.newegg.com/.
[4] perf. https://perf.wiki.kernel.org.
[5] Shore-MT. https://sites.google.com/site/shoremt/.
[6] Anon et al. A measure of transaction processing power. In *Datamation*, 1985.
[7] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD'09*, Providence, Rhode Island, USA, 2009.
[8] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *SOSP'13*, Famington, Pennsylvania, 2013.
[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 *ASPLOS'11*, Newport Beach, California, USA, 2011.
[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *SOSP'09*, Big Sky, Montana, 2009.
[11] D. J. Dewitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84*, New York, NY, USA, 1984.
[12] S. R. Dulloor, S. K. Kumar, A. K. Keshavamurthy, P. Lantz, D. Subbareddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys'14*, Amsterdam, The Netherlands, 2014.
[13] R. Fang, H.-I. Hsiao, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE'11*, 2011.
[14] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *SIGMOD'13*, New York, USA, 2013.
[15] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. In *VLDB'09*, Lyon, France, 2009.
[16] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *EDBT'09*, Saint Petersburg, Russia, 2009.
[17] R. Johnson, I. Pandis, R. Stoica, and M. Athanassoulis. Aether: A scalable approach to logging. In *VLDB'10*, Singapore, 2010.
[18] P. Kieun. Database Technology for Large Scale Data. http://www.cubrid.org/blog/dev-platform/database-technology-for-large-scale-data/.
[19] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *FAST'14*, Santa Clara, CA, USA, 2014.
[20] R. Lawrence. Early hash join: A configurable algorithm for the efficient and early production of join results. In *VLDB'05*, Trondheim, Norway, 2005.
[21] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD'08*, Vancouver, BC, Canada, 2008.
[22] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE'04*, Boston, USA, 2004.
[23] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS'12*, London, UK, 2012.
[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP'11*, Cascais, Portugal, 2011.
[25] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the nvram era. In *VLDB'14*, Hangzhou, China, 2014.
[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA'09*, Austin, Texas, USA, 2009.
[27] A. Rafii and D. DuBois. Performance tradeoffs of group commit logging. In *CMG Conference*, 1989.
[28] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS'11*, Newport Beach, California, USA, 2011.
[29] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. In *VLDB'14*, 2014.
[30] Wikipedia. ARIES. http://en.wikipedia.org/wiki/Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics.
[31] J. H. Yoon, H. C. Hunter, and G. A. Tressler. Flash and dram si scaling challenges, emerging non-volatile memory technology enablement-implications to enterprise storage and server compute systems. In *Flash Memory Summit*, 2013.
[32] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistent support. In *MICRO-46*, Davis, CA, 2013.