Scalable Logging through Emerging Non-Volatile Memory

Tianzheng Wang
Department of Computer Science
University of Toronto
tzwang@cs.toronto.edu

Ryan Johnson
Department of Computer Science
University of Toronto
ryan.johnson@cs.utoronto.ca

ABSTRACT

Emerging byte-addressable, non-volatile memory (NVM) is fundamentally changing the design principle of transaction logging. It potentially invalidates the need for *flush-before-commit* as log records are persistent immediately upon write. Distributed logging—a once prohibitive technique for single node systems in the DRAM era—becomes a promising solution to easing the logging bottleneck because of the non-volatility and high performance of NVM.

In this paper, we advocate NVM and distributed logging on multicore and multi-socket hardware. We identify the challenges brought by distributed logging and discuss solutions. To protect committed work in NVM-based systems, we propose *passive group commit*, a lightweight, practical approach that leverages existing hardware and group commit. We expect that durable processor cache is the ultimate solution to protecting committed work and building reliable, scalable NVM-based systems in general. We evaluate distributed logging with logging-intensive workloads and show that distributed logging can achieve as much as ∼3x speedup over centralized logging in a modern DBMS and that passive group commit only induces minuscule overhead.

1. INTRODUCTION

Since its debut in the early 90s, ARIES [28] has been the *de facto* standard of transaction logging. Despite the prevalence of multicore hardware and large main memories, most systems still use a centralized ARIES-style log that buffers log records in DRAM until a commit request forces them to stable storage. Given the volatile, fast DRAM and non-volatile, slow disk, such *flush-before-commit* principle improves performance, without risk of lost work, by replacing the random write-back of all dirty pages in a transaction's footprint with (ideally) a single sequential I/O to harden the log. However, centralized logging has become a significant bottleneck on today's massively parallel hardware [21, 22].

Emerging byte-addressable, non-volatile memory (NVM) technologies, such as phase change memory (PCM) [44] and spin-transfer torque RAM (STT-RAM) [14], are fundamentally changing the design principle of logging. These NVM products promise high performance, good scalability and low power consumption [25]. For example, STT-RAM could achieve ∼10ns write latency [18], which is even faster than DRAM. With proper caching, carefully architected PCM could also match DRAM performance [4]. These memories can be manufactured as DIMMs and placed side by side with DRAM on the memory bus, allowing direct access through ordinary `load` and `store` instructions. The combination of DRAM-like performance and non-volatility makes NVM an attractive candidate for logging: log records persist immediately upon write, so transactions need not force log records to disk at commit. Various simplifications can then be admitted to improve logging performance and reliability, such as removing asynchronous and group commit [12, 37], and unifying the log buffer with backend log storage [10].

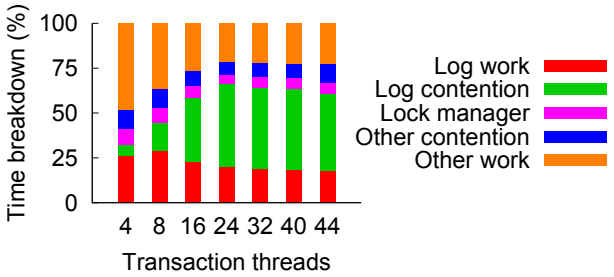

Figure 1: Time breakdown of running the Update Location transaction of TATP on a modern DBMS. Log contention is a major source of overhead.

Although NVM-based logging brings improvements in latency and single-thread performance, contention for the log head still exists because of the centralized design. Figure 1 shows the time breakdown of Shore-MT [20], a prototype database system optimized for modern hardware, when running a logging-intensive workload on a 48-core quad-socket server (with hyper-threading). As system load increases, contention for the centralized log becomes a major overhead. Though rarely used in single node systems, we observe that a distributed log has the potential to remedy this situation. In this paper, we show that NVM-enhanced distributed logging can eliminate the logging bottleneck for modern multicore and multi-socket hardware. We equip each log with an NVM-based buffer, accessed via `load` and `store` instructions. Transactions commit immediately after buffering their commit records, and log records are only de-staged to disk when the log buffer is full.

Proceedings of the VLDB Endowment, Vol. 7, No. 10

Neither adopting distributed logging nor NVM is trivial. Distributed logging poses two main challenges: how to assign log records to logs (log space partitioning), and how to prevent holes in the distributed log, without imposing excessive overhead at commit time. ARIES-style recovery favors partitioning the log either by page or transaction, i.e., assigning all log records involving a given page or transaction to the same log, respectively. As we will show in later sections, adopting either approach involves different design challenges and performance implications.

Byte-addressable NVM is no panacea, either. Modern processors heavily rely on multiple levels of caches to improve performance. Though log records become persistent immediately after reaching NVM, by default they are first buffered in volatile processor caches ("write-back" caching). Data remain cached until some event evicts the value from the caches (e.g., a cache conflict or the OS explicitly requests a cache flush), and a power failure in the meantime would lose log records buffered in the SRAM cache. Spreading log records over multiple logs worsens the situation: a transaction cannot commit until all records from all logs it accessed have reached persistent storage. In case of transaction-level partitioning, the transaction must also ensure that all previous log records for pages in its footprint have become persistent, even if they reside in a log the transaction never accessed. Tracking these dependencies explicitly is prohibitively complex, and forcing log records to disk at every commit imposes unacceptable I/O overheads [21].

Based on existing hardware, we propose *passive group commit*, a lightweight group commit protocol for NVM-based distributed logging, to protect committed work upon failure. Unlike most NVM logging proposals, passive group commit does not rely on NVM-specific hardware support for customized memory barriers [9], and does not issue a barrier after buffering each log write to NVM [10, 35]. Instead, passive group commit builds on commodity processor features—write combining and memory barriers—and leverages group commit to ensure all necessary log records are persistent before a transaction finishes committing.

Passive group commit is very lightweight and offers significant performance benefits for existing hardware equipped with NVM, but is actually a stop-gap solution. For future NVM-based systems, we argue that the ultimate solution is not a new type of memory barrier, but rather a *durable processor cache* [23, 45]. A durable cache ensures that writes are effectively persistent as soon as they leave the CPU's store buffers, and can be implemented either with on-chip NVM or a supercapacitor-based solution that drains caches whenever power is lost. Durable caching is largely transparent to software. The only requirement, on some processors, is to employ memory fences that prevent reordering of stores (x86 does not reorder stores, but other architectures like ARM are more aggressive). Durable caches are also helpful in extending the lifetime of NVM by allowing the cache hierarchy to absorb the vast majority of writes. Although not yet widely available, they could be developed rapidly using existing technology: prototypes have already been built [29, 32]. Durable caches would improve and simplify our proposed design even further, by removing the need for passive group commit and write combining. Due to their ease of use, high performance, and reduced wearout rates for underlying NVM, we expect durable caches to become widely available as NVM-equipped systems become more popular.

In summary, we make the following contributions:

- We show that NVM allows for practical distributed logging, thereby eliminating the scalability challenges posed by a single centralized log.
- We propose *passive group commit*, a lightweight group commit protocol that leverages existing hardware support for write combining to protect committed work upon failures.
- We show that durable cache architectures promise to both simplify distributed logging and improve its performance in NVM-based systems.

Note that the focus of this work is *not* to create another NVM-based log that removes I/O latencies. Rather, we propose to leverage NVM in support of distributed logging, with the goal to alleviate the contention bottleneck that all centralized logging schemes suffer (NVM-based or otherwise).

In the rest of this paper, we provide background on NVM and logging in Section 2. In Section 3, we discuss design challenges brought by distributed logging and propose solutions. Section 4 details approaches to protecting committed work and expected support for NVM from hardware/OS. Section 5 presents evaluation results. We summarize related work and conclude in Sections 6 and 7, respectively.

2. BACKGROUND

In this section, we briefly introduce NVM technologies and discuss how they could be used for database logging. In the end, we give an brief overview of database logging.

2.1 Byte-addressable Non-Volatile Memories

Several NVM technologies—such as PCM [44], memristor [41] and STT-RAM [14]—are being actively investigated as alternatives to DRAM due to their scalability, energy efficiency and non-volatility. Other efforts focus on NV-DIMMs, or DRAM backed by flash/batteries [1, 43]. Though based on different technologies, these NVMs are all byte-addressable and can be placed side by side with DRAM on the memory bus. Software can use ordinary `load` and `store` instructions to access them. Data are immediately persistent upon write, no constant voltage is needed to maintain the data. We summarize these NVM technologies below.

Maturing/in production NVM. Although most NVM technologies are only beginning to roll out, enterprise-level products are available today—for applications that can justify the cost—in the form of DIMMs backed by batteries or NAND flash [1, 43]. These products are already in mass production and available on the market. They usually buffer data first in DRAM to allow full DDR3 speed, and then periodically persist data on flash. Such NV-DIMMs could be a convenient drop-in solution for distributed logging.

Another maturing technology is PCM, which stores data by changing the state of its underlying material. PCM cells consist of two electrodes and phase change materials that can switch between low- and high-resistance states in response to electrical signals [44]. Raw PCM exhibits longer latency (80ns – 1μs) than DRAM (20ns – 50ns) and limited programming cycles (10^8 – 10^9) [6, 25]; various compensating mechanisms have been proposed [7, 17, 25, 36, 46]. We expect future PCM products to be equipped with a capacitor-backed SRAM buffer (like some disks today) to deliver near-DRAM performance with reasonable lifetimes.

Future promising candidates. Known as the "fourth basic circuit element", the memristor [41] consists of two layers of titanium dioxide and wire electrodes. Its resistance changes as electric current passes through, thus storing information. Reported latency ranged from hundreds of pico-seconds to tens of nanoseconds [38]. 100TB memristor drives may be available by 2018 [27]. STT-RAM [14] is another promising NVM technology that offers 10–25ns latency, making it a good candidate for both on-chip cache and storage [18]. These NVM technologies are still several years from reaching the market, but their high performance and non-volatility will further ease the adoption of NVM as a drop-in solution to scalable logging.

Summary. Flash/battery backed NV-DIMMs and PCM are the most promising NVM candidates to date, as they are already or about to be available on the market. Despite slower writes and limited endurance, it is widely accepted that commercial PCM products will feature a capacitor-backed SRAM/DRAM buffer, both to hide latency and to avoid premature wearout. Moreover, our scheme writes log records sequentially and without partial overwriting, which makes logging an ideal candidate application for NVM; we do not regard endurance as a major problem. NVMs that are fast enough (e.g., flash-backed NV-DIMMs) could be used as a drop-in replacement. Our approach does not rely on an NVM having specific timing or wearout characteristics.

2.2 Write-ahead Logging

First proposed by Mohan et al. [28], ARIES logging is endemic among both commercial and open source database systems. Transactions record all updates in a single, global log before the dirty pages reach disk. To hide the performance difference between fast memory and slow disk, log records are cached in a centralized DRAM buffer and forced to disk only at transaction commit. The log is usually protected by a latch or mutex, which must be obtained by the inserting transaction before writing to the log buffer. Despite the adoption of multicore and multi-socket hardware, centralized logging still dominates database engine designs.

When a transaction commits, its buffered log records must be forced to disk to ensure recoverability. Each log record is uniquely identified by a monotonically increasing log sequence number (LSN). An LSN in each page indicates the latest log record that modified the page. During recovery, the log is analyzed and scanned forward to repeat history ("redo"), applying log records with an LSN greater than its targeted page. After redo, loser transactions are rolled back ("undo"). A compensation log record (CLR) is written after each undo operation to make sure log records are undone exactly once in spite of repeated crashes during recovery.

3. DISTRIBUTED LOGGING

Distributed logging eases the logging bottleneck by spreading log insertions over multiple physical logs. Historically, distributed logging has been prohibitive for single node systems due to dependency tracking and I/O overheads. Most single node systems avoid using a distributed log, and even distributed systems have used a centralized log, hosted on a dedicated node [26]. NVM makes it possible to use distributed logging in single node systems, as it potentially eliminates the need for *flush-before-commit*. Log de-staging is only required when log buffers are full, and uses large sequential writes to maximize disk bandwidth utilization.

Logging is tightly coupled with other core components in database systems (e.g., the transaction manager) and is strongly affected by recovery: ARIES uses physical logging for redo and logical logging for undo (physiological logging). Both redo and undo have their own parallelism models: during redo, modifications to different pages can be replayed in parallel, while undo is parallelized at transaction-level. Resource managers that generated the log records select a log either by page or transaction, and the competing recovery parallelism modes add to the importance of selecting the right partitioning scheme. The rest of this section discusses the design challenges and trade-offs posed by distributed logging, in terms of both forward processing and recovery.

3.1 Forward Processing

Uniqueness of log records. In a centralized log, the log sequence number (LSN) uniquely identifies each log record; records written by the same transaction are chained together using a `prevLSN` field, which identifies the previous log record written by the same transaction. When a transaction aborts, log records are undone by following the chain of `prevLSN` pointers, most-recent first. However, LSNs are not unique identifiers in distributed logging, because they only indicate a log record's position within the log that holds it, saying nothing about the relative ordering of records from different logs. Under page-level log space partitioning, a transaction could touch any page and write to any log, and a `prevLSN` no longer allows a transaction to trace its log records backward. For transaction-level log space partitioning, a transaction only inserts log records to a single log, but page LSNs are no longer unique in the transaction because updates generated by different transactions go to different logs. The lack of ordering among records from different logs potentially causes the recovery manager to skip applying newer log records, or even to apply older log records on top of newer log records that happen to have smaller LSNs.

To uniquely identify log records, we propose a *global sequence number* (GSN) based on logical clock [24]. At runtime, a GSN is maintained in each page, transaction and log. Pages and log records also store a GSN with them permanently. Pinning a page in the buffer pool sets both page and transaction GSNs to `max(tx GSN, page GSN) + 1` if the transaction intends to modify the page; otherwise only the transaction GSN is set to `max(tx GSN, page GSN)`. Inserting a log record will set both transaction and page GSNs to `max(tx GSN, page GSN) + 1`. The log then sets its GSN to `max(tx GSN, page GSN, log GSN)`. The same GSN is also stored in the log record that is being inserted. In addition to the GSN, each log record also stores an LSN, which indicates the offset into an individual log that holds it. Although GSNs are only partially ordered, they uniquely identify log records and provide a total order for log records belonging to any one page, log, or transaction.

GSN solves the uniqueness problem posed by LSN in distributed logging. The recovery manager can follow GSNs to determine if a record should be applied to its targeted page. Note that GSN might not be mandatory for page-level log space partitioning because log records for the same page are always stored in the same log. To allow transaction rollback, each transaction should record not only the LSN, but also the log in which each log record is stored. Though not required, using GSN can avoid such complex bookkeeping and simplify the design of page-level partitioning. Transaction-

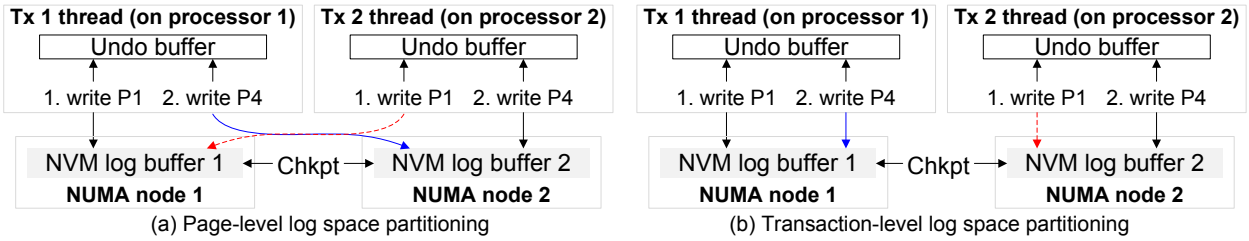

Figure 2: The processor affinity problem brought by distributed logging. (a) Threads may cross NUMA node boundaries to insert log records. (b) Threads only insert to the log that is local to its running processor.

level partitioning relies heavily on GSN to function properly during forward processing as log records for the same page could be stored in arbitrary logs by different transactions.

Cross-log aborts. GSN only partially solves the problem of tracking log records in a distributed log. Compared to LSN, GSN is also monotonically increasing, however, it does not identify the log a particular log record came from, nor give the record's byte offset within that log. A naïve implementation will scan multiple logs during forward processing to find the correct log record with a specified GSN. To avoid expensive log scans, we maintain a *private DRAM undo buffer* which records log writes in each transaction. Similar approaches have already been implemented in both commercial and open source systems to simplify transaction rollback [40]. The buffer is managed as a stack (with abort repeatedly popping records), and discarded after the transaction is ended. Log records generated during rollback are never undone, and recorded in the NVM log buffer only.

Processor affinity. The shift to multicore and multi-socket systems brings non-uniform memory access (NUMA), where memory is either "local" or "remote", with the latter being much more expensive to access. Transaction threads could be scheduled on arbitrary processors, spreading insertions over any log, which in turn could be allocated in any NUMA node (or striped over all of them). If the log buffer and the transaction thread are not within the same NUMA node, as shown in Figure 2(a), log insertion will involve remote memory accesses. Since page-level log space partitioning allows transactions to write to any log, accessing remote memory is inevitable, unless transactions are scheduled in a way that restricts threads to particular sets of pages (e.g. using physiological partitioning [34]). Transaction-level partitioning, on the contrary, fits directly with multi-socket hardware as each transaction's writes go to one log. As shown in Figure 2(b), a log could be allocated in each NUMA node, with transactions assigned to logs based on the CPU they run on. This approach eliminates remote memory accesses when inserting log records, improving logging performance and freeing up interconnect bandwidth for other uses.

Checkpointing. To accelerate recovery, ARIES mandates periodic checkpoints of all dirty pages and in-flight transactions during forward processing. The recovery manager can load these tables directly rather than scanning the entire log to reconstruct them. A distributed log requires the checkpointing thread to be aware of individual logs, especially for page-level log space partitioning: either page GSN must be recorded, or a dirty pages table should be generated for each log. The analysis pass can then process log records from different logs in parallel. As the active transactions table is global and records only transaction status, it could be stored in a predefined log (e.g., the first one), and recovered with a pre-analysis step. Another approach is to generate multiple active transactions tables, one for each log, to remove the pre-analysis pass.

Log space utilization and skew. Under page-level partitioning, transactions leave log records in multiple logs during forward processing, but all records for a given page are stored in the same log. If certain pages are especially hot, the corresponding logs could become over utilized, while leaving other logs under utilized. In contrast, transaction-level partitioning stores each transaction's log records in a single log, so log accesses should be distributed uniformly as long as the OS spreads load evenly across cores. Transactions are usually short enough, and migrations rare enough, that we can assign each transaction a log based on the core where it first runs, with little impact on performance.

3.2 Recovery

ARIES separates recovery into three passes: analysis, redo and undo. The analysis pass extracts information (e.g., lists of dirty pages and active transactions) logged before the system failed, to determine how redo and undo should proceed. With a distributed log, the analysis pass could be parallelized at both page or transaction levels, though a pre-analysis step might be needed first, to retrieve the dirty page and/or active transactions tables (as discussed previously). Once analysis completes, ARIES naturally supports parallel redo and undo: redo of different pages can proceed in parallel, as can undo of different transactions. Ideally, a distributed log should recover fully in parallel, but neither log partitioning choice aligns perfectly with both redo and undo. We discuss how distributed logging impacts recovery under page and transaction level partitioning in this section.

Page-level log space partitioning. A page-partitioned distributed log suits parallel redo nicely. The only difference is that each redo thread should work with pages from a single log, preferably residing on the same socket. However, the undo pass requires analysis to build up the same per-transaction log buffers that would have existed during forward processing, which is potentially expensive given that we expect the vast majority of transactions to have committed with only a few losers left to undo. Further, if the per-transaction buffers store pointers rather than copies of log records, a parallel undo pass would have to randomly access multiple logs, which is expensive for disk based systems (recall that log records are de-staged to disk as the NVM buffer fills). The overheads can be mitigated by copying log records to the per-transaction buffers, or (if transactions are too large to cache this way) by using efficient checkpointing to ensure that most log records to be undone are in NVM.

Table 1: Comparing page-level and transaction-level log space partitioning.

	Page-level	Transaction-level
Record uniqueness	LSN is adequate, but GSN is more convenient	GSN is required
Checkpointing	Per-log dirty pages tables without GSN	Per-log transaction tables for simpler recovery
Recovery	Easy parallel redo, expensive transactions-level parallel undo (need to hop among multiple logs)	Log synchronization needed for parallel redo; cheap parallel undo by transactions
CPU affinity	Cannot utilize CPU affinity since a transaction could potentially insert to any log	Logs can be pinned to specified NUMA nodes to serve requests exclusively for the node
Log utilization	May be unbalanced, depending on workload	Balanced if transactions are scheduled properly

Tracking dummy CLRs written by nested top actions also complicates undo for page-level partitioning. ARIES uses nested top actions and dummy CLRs to denote (usually physical) operations that will never be undone once complete, even if the requesting transaction eventually aborts. For example, B-Tree page splits only affect physical representations (not logical contents in the database) and can safely persist across transaction rollback [28]. A dummy CLR is written after the nested top action (e.g., B-Tree split) completes, with a `prevLSN` field pointing directly to the log record before the nested top action started. Since a dummy CLR could cover multiple log records which are spread over multiple logs under page-level partitioning, tracking back to the log record before the nested top action involves jumping across different logs. Even though the log records written by nested top actions are never undone, they still have to be examined (and potentially cached) during the analysis pass.

To simplify undo, we replace nested top actions with *system transactions* [13]. A system transaction only modifies physical representations of the database, and will not change any logical contents visible to the user. Therefore, system transactions acquire no locks, and can inherit latches from the invoking transaction. The user transaction can start a system transaction when physical-only changes are required, and resumes after it finishes.[1] System transactions also generate log records like user transactions do, and the rollback of an interrupted system transaction is identical to that of a user transaction: it can safely go through its log records in the private DRAM undo buffer to undo them one by one.

Transaction-level log space partitioning. Parallel undo is straightforward for a transaction-partitioned log as all the log records to undo a transaction reside in the same log. However, redo becomes more complex because dependencies among pages can reside in any log. A naïve replay of all logs in parallel results in log records for the same page being applied in arbitrary order, potentially skipping certain records. For example, consider a transaction `Tx 1` that modified page `P1` and logged it on `Log 1` with GSN `50`. Another transaction `Tx 2` modified the same page but logged it on `Log 2` with GSN `60`. Though the log record generated by `Tx 2` had a larger GSN, its relative position (byte offset) in `Log 2` could be well before the log record generated by `Tx 1` in `Log 1`, causing the log record generated by `Tx 2` to be applied first (assuming we start redoing both logs at the same time and they proceed at the same speed). As the redo pass will not apply a log record with a GSN less than the page GSN, the record generated by `Tx 1` will be skipped.

[1]The system transaction can use the resources (e.g., latches) of its invoking thread, which therefore is essentially "waiting" for a *function call* to return, instead of another thread.

A two-step redo pipeline, reminiscent of map/reduce, can solve this problem efficiently: the first stage uses `N` threads to scan the logs up to some target GSN, partitioning log records into `M` buckets by page ID. `M` threads can then sort each bucket and apply the changes in parallel, without risk of skipped records. Full parallelism can be achieved using this approach, with the available parallelism balanced between the `N` partitions and `M` redo threads.

3.3 Page vs. Transaction-Level Partitioning

We compare and summarize page-level and transaction-level log partitioning in this section. Figure 2 clarifies the basic ideas of both approaches based on an example on a dual-socket machine. In Figure 2(a), each NUMA node has an NVM log buffer allocated from local memory. Two transaction threads (`Tx 1` and `Tx 2`) run on two processors. The example assumes all pages with odd page numbers are covered by `Log 1`, and all even numbered pages are covered by the other log. Under page-level partitioning, both transactions access remote memory for log insertions, while transaction-level partitioning generates only local writes. For both designs, checkpointing records could be kept in a predefined log or generated for specific logs to ease recovery: page-level partitioning could use individual dirty pages tables for each log, while transaction-level partitioning could utilize per-log active transactions tables.

Table 1 compares the two approaches in terms of design challenges. In general, a transaction-level partitioned distributed log features straightforward undo and tracking of log records. However, it needs more complex redo because of dependencies among pages and transactions. Page-level log space partitioning is implicitly supported by ARIES. It allows easy parallel redo, but complicates with transaction-level parallel undo and generates more remote accesses. As we will show in our evaluation results, transaction-level partitioning achieves higher performance during forward processing for multi-socket hardware as it avoids crossing NUMA boundaries to insert log records.

4. NVM-BASED LOGGING

Modern processors rely heavily on caching for good performance. When accessing NVM through the memory interface, log records are first cached by the CPU. Log records that are not yet persistent in NVM could be lost upon failures, which risks losing committed work. ARIES truncates the log at the first hole, and transaction-level partitioning could leave the database unrecoverable if earlier updates to a page in one log are lost and the system attempts to apply later updates to the same page from a different log. Worse, the transactions generating such log records may not have committed before a crash. So it is not sufficient to track

which logs a committed transaction touched. Instead, we must ensure that no transaction can commit until every log is durable up to the transaction's commit GSN. The overhead of flushing every log at every commit (or every time a page moves to a new log) is a principal reason why implementations have avoided distributed logging when possible.

It is straightforward to store log records directly in NVM by completely disabling caching for the NVM address range. However, this approach adds significant latency to the critical path (log insertion) and threatens to cancel out any benefits brought by a distributed log. Prior NVM research has proposed augmenting the processor with a new memory epoch barrier [9]. Although such a barrier would be complex to implement, and would have high runtime overhead, some NVM logging proposals rely on this primitive. Fang et al. [10] propose to issue an epoch barrier after writing each log record to persist the record. To tolerate holes in the log, an epoch barrier is issued as soon as the log header is written, leaving transactions to populate log records at their leisure. Pelley et al. [35] proposed NVRAM Group Commit to mitigate the overhead of the epoch barrier. All these approaches are tailored for centralized logging, however. With a distributed log, the issuing processor would have to not only flush its own records to NVM, but also ensure that other processors have done the same (in case any dependent log records are not yet durable). The proposed epoch barrier introduces more complexity in both processor cache and the memory controller, and it is not clear whether this approach can scale in multicore environments. Instead of relying on hypothetical hardware primitives, we develop a group commit protocol that protects committed work by leveraging existing hardware support.

4.1 Passive Group Commit

This section presents *passive group commit*, a lightweight group commit protocol for reliable and high performance NVM-based distributed logging. Passive group commit leverages existing hardware primitives to protect committed work while still providing good performance.

Leveraging existing hardware primitives. Modern processors allow the OS to control how the cache interacts with accesses to a given physical address range. Addresses can be marked as `uncacheable` (important for memory mapped I/O), `write-through` (serving reads from the cache but propagating all writes immediately), `write-back` (the normal caching mode), or `write-combining` (WC). The latter is a compromise for uncacheable memory in cases where write coalescing is tolerable. Accesses to WC memory bypass caches for both reads and writes, but can buffer writes inside the CPU for a short time. For example, the Intel Core series processors have eight WC buffers per core [16]. WC buffers can coalesce repeated/adjacent writes (replacing several small memory transfers with a larger one) and allow store instructions to complete before the write actually completes (to be drained lazily); with judicious use of normal memory fences, write combining provides comparable performance to cached writes, while guaranteeing that those writes still reach the memory system in a timely manner [2, 15]. Events such as interrupts and context switches also drain WC buffers, avoiding the need to worry about thread migrations. Though WC only provides uncached (slow) reads, it has minuscule effect on logging, which is write-mostly. In particular, our distributed log does not read from NVM during forward processing (abort is supported by the private DRAM undo buffer). During recovery, we map the NVM in `write-back` mode before the undo pass for fast reads, because no log records are generated during the analysis or redo passes. Thus, write combining is a natural fit for NVM-based logging buffers. Whenever writing sensitive log records (e.g., commit), a memory barrier (e.g., `mfence`) is issued to empty the WC buffers and persist log records in NVM. When the user receives a return value from the log manager, the records are guaranteed to reside in NVM. However, simply issuing memory barriers when committing the transaction does not solve the dependency problem, as discussed in the beginning of this section.

Protecting committed work. Log records are spread over multiple logs by transaction threads running on different processors. Before a transaction can safely commit, all of its log records, along with all the log records that logically precede them, must become persistent. For transaction-level partitioning, log records affecting pages touched by a committing transaction could have been created by other, as-yet-uncommitted transactions running on other processors. The committing transaction's own memory barrier only empties the worker thread's own WC buffers, potentially leaving (parts of) other dependent log records in other cores' WC buffers; a memory barrier from the running processor will only make its writes globally visible from a cache coherence perspective, and does not guarantee that other WC buffers have drained. We thus introduce a variant of group commit to coordinate draining of WC buffers on other cores.

Group commit was originally proposed to avoid excessive random writes to the disk. It gathers and writes multiple commit records in a single disk I/O. Passive group commit relies on a similar mechanism to ensure that all log records written by the committing transactions are persistent in NVM before notifying the client that commit succeeded. In each thread, we maintain a thread-local variable called `dgsn` to record the GSN of the last log record the thread can guarantee to be persistent. We reserve one bit as a dirty flag, to indicate whether the thread has written any log records that might not be persistent yet. Whenever a thread writes a log record, it sets the dirty flag. When a transaction commits, it will issue a memory fence, clear the dirty bit and store the record's GSN in `dgsn`. The transaction is then added to a group commit queue. The group commit daemon periodically examines the `dgsn` of each worker thread and records the smallest `dgsn` it sees (`min_dgsn`), as the guaranteed upper bound of persistent log records. Log records written by all transactions on any processor with a GSN $\leq$ `min_dgsn` are guaranteed to be persistent. The passive group commit daemon dequeues transactions once their commit record precedes `min_dgsn` and notifies the client. If any thread takes too long to update its `dgsn` (perhaps due to a long read-only query), the daemon sends a signal to the straggler. Signal delivery implicitly drains the WC buffers of the processor the thread is running on, and the thread's signal handler then clears the dirty bit so the commit daemon can proceed. Note that operations on `dgsn` need not be atomic, because the owning thread makes all writes and the daemon thread reads a monotonically increasing GSN.

Figure 3 shows an example of passive group commit in action. Here, each transaction thread runs on a physical processor. Transactions `Tx 3` and `Tx 4` update pages `P1` and `P2` before commit. At steps (3) and (4), both transactions

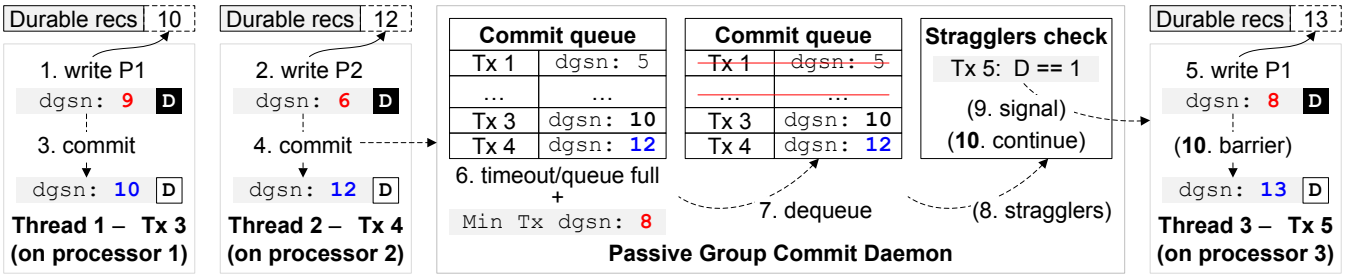

Figure 3: Protecting committed work with passive group commit. The passive group commit daemon keeps dequeuing committed transactions with GSN $\leq$ the minimum `dgsn` and signaling stragglers to make progress.

issue an `mfence`, clear their dirty bit, and join the commit queue. Another transaction `Tx 5` sets its dirty bit and inserts a log record for updating `P1` (right of the figure). The daemon reads `dgsn` of all threads and obtains the minimum, which is 8 from `Tx 5`. Note that `Tx 5` has not committed yet, therefore it still has the dirty bit set. The daemon then dequeues all transactions with `dgsn` $\leq$ 8 in the queue and notifies the clients. After step (7), `Tx 3` and `Tx 4` are left in the queue since they have `dgsn` $>$ 8. Normally the daemon will start another round of checking for the smallest `dgsn` and repeat (1) – (7). However, if there is any straggler which is not updating its `dgsn`, the commit queue will become full gradually and no more transactions could be removed from it due to low `min_gsn`. In such cases, the daemon will signal through `pthread_kill` all threads whose dirty bit is still set. Upon receiving the signal, the thread's signal handler will set the correct `dgsn` and clear the dirty bit, as shown in steps (8) – (10). Since the signal delivery is asynchronous, the call to `pthread_kill` may return before the signal is delivered. Therefore the daemon will return to step (6) and start checking again, as shown in step (10).

Note that it is unnecessary to issue a barrier for every processor. We only need to make sure that all the processors that have written log records issue a barrier to empty their WC buffers. This observation makes passive group commit lightweight: signaling is only required if stragglers arise.

4.2 Durable Processor Cache

Recent advances in ferroelectric RAM(FeRAM) [3] and STT-RAM [23] have made it possible to build processors with durable caches. For instance, real prototypes [32] have been built using FeRAM to realize "normally off, instant on" computing. Kiln [45] is a recent proposal which adopts both non-volatile cache and main memory to build a persistent memory system. Another approach is augmenting the processor with a supercapacitor [9] that can drain the SRAM cache back to NVM if power is lost. As a beneficial side effect, durable caches improve NVM lifetime by allowing large L2 and L3 caches to absorb the vast majority of writes without risking data loss. When a durable cache is available, programming NVM becomes much simpler: NVM can be treated like normal memory by the OS and caches, and software need only issue memory fences, when it is necessary to force ordering of writes. Based on these observations, we expect that durable processor cache (built with capacitor-backed SRAM or NVM such as FeRAM) is the ultimate solution to reliable and high performance NVM-based systems. Though durable cache is still in its infancy, we believe it will become the most convenient drop-in solution in the future. In the context of a distributed log, passive group commit is no longer necessary since all writes, on all processors, are immediately durable as long as each log insertion includes a (now inexpensive) memory fence to ensure write ordering on certain architectures that are aggressive on reordering store operations (e.g., ARM and SPARC). Therefore, in later sections we also measure the performance that could be achieved with a durable cache, in addition to evaluating passive group commit.

4.3 Hardware/OS Support

To use NVM in existing systems, the memory controller should distinguish DRAM and NVM, so that the OS can manage both of them and enable applications to make requests for both types of memory explicitly. Moreover, transaction level log space partitioning requires directly allocating NVM from specified NUMA nodes. It is therefore desirable to have a certain amount of NVM on each NUMA node. We use NVM for buffering log records, and use DRAM as main memory for the buffer manager, etc.

At the software level, the memory allocator should provide interfaces for explicitly allocating NVM from a specified NUMA node for transaction-level log space partitioning. For passive group commit to work, the OS should expose interface for user space to specify the caching mode. However, currently most systems expose such interfaces in kernel space or only allow very limited control in user space (e.g., via MTRR registers). NVM-based systems would benefit strongly if the OS could expose more interfaces for user space to directly control the behavior of NVM. These changes are straightforward and easy to make in the OS. For example, caching-mode-related functions are already available for Linux device drivers. Even without current OS support, our kernel module for NUMA-aware write combining memory allocation has only ~250 LoC. After changes in the OS, such functionality can be exposed to user space as system calls easily. Given historical trends, however, we suspect that durable caches will become widely available before operating systems incorporate such support.

5. EVALUATION

We implement both flavors of distributed logging and passive group commit based on Shore-MT [20], an open source DBMS prototype designed specifically to remove bottlenecks on multicore hardware[2]. Since different NVM products have varied latencies and most of them provide near-DRAM (or

[2]Downloaded from `https://bitbucket.org/shoremt`.

faster, such as STT-RAM) performance as discussed in Section 2, we first use DRAM in our experiments, and then add delays to emulate the latencies for worst-case analysis. The rest of this section presents experimental setup and results.

5.1 Experimental Setup

We conduct experiments on a quad-socket Linux server with 64GB main memory and four Intel Xeon E7-4807 processors, each with six physical cores. With hyper-threading, the OS sees a total of 48 "CPUs". The data used in our experiments are completely memory-resident using a `tmpfs` file system, to eliminate I/O effects unrelated to logging. Buffer flushing and checkpointing are still enabled, however.

Experimental variants. We evaluate distributed logging based on two dimensions: (1) log space partitioning: transaction (TX) or page (PG) level, and (2) processor cache type: durable (D) or volatile (V). Variants that use volatile caches rely on passive group commit to protect committed work. Variants that emulate a durable cache machine disable passive group commit as it is not needed. Based on existing work on non-volatile caches, similar performance to SRAM is expected, giving us the opportunity to measure the performance on existing hardware. We use the default `write-back` mode for durable cache variants, and `write-combining` for volatile cache variants. To focus on logging performance, we minimize other bottlenecks by enabling speculative lock inheritance (SLI) [19] and early lock release (ELR) [21] in Shore-MT for all distributed logging variants. Both flavors of distributed logging are compared to Aether with SLI and ELR enabled (denoted as "Aether"). Results from another Aether variant without SLI and ELR (denoted as "Baseline") are also provided for comparison. For both Aether and Baseline, group commit is enabled and the log buffer size is set to 80MB. For all distributed logging variants we use 20 logs, each with a 64MB NVM-based log buffer. TX variants have five logs per socket and insert log records only to local logs. PG variants insert log records into the log with `logID = pageID modulo 20`. Records carrying no page ID are inserted into a predefined log. These configurations are completely adjustable, and we did not observe significant difference by varying them as other contention becomes the major bottleneck.

NVM performance. As we discussed in Section 2, some NVM technologies (e.g., flash backed NV-DIMMs) already promise at least comparable speed to DRAM. For such candidates, we use DRAM in our experiments. PCM is the only maturing byte-addressable NVM with longer latency. Since it is not widely available, we add a conservative delay of 1.2μs, longer than the maximum ($\sim$1μs) reported in most existing literature [6, 25], on top of DRAM latency for emulation. The final, mature PCM DIMMs are still yet to reach the enterprise market and they are constantly changing anyway. Therefore, by using DRAM and artificial delays, we only provide a lower bound for NVM latency based on existing hardware. Our scheme is general enough that better performance is expected with faster-than-DRAM NVM. The extra latency is added through a loop that busy-waits for a specified number of CPU cycles. We first convert clock time latency to CPU cycles according to our CPU's frequency. Then for each log insertion, we embed an `RDTSC` instruction in the loop to read the number of cycles passed since we entered the loop, so it can break after the specified number of cycles have passed. Since our NVM log buffers are write-only—a private DRAM undo buffer is maintained in each transaction—we do not introduce read delays. Recent research has shown that carefully architected DDR PCM modules could provide a write bandwidth comparable to at least 50% of that of DRAM [4]. Pelley et al. [35] also showed that bandwidth is not a concern for OLTP. Thus we do not specifically model NVM bandwidth in our experiments.

Log buffer allocation. For Baseline, Aether and PG-D variants, we use the default `malloc` to allocate log buffers and use write-back caching. We build a Linux device driver for PG-V variants to allocate write combining memory. We export it to user space via the `mmap` interface by reserving memory at the end of the physical address space (i.e., the last NUMA node in our experimental system). We develop the driver this way due to the difficulty of mimicking the behavior of `malloc` and the need for write combining. In Section 5.3 we show the impacts of this design in detail.

For TX variants, we implement another driver which is capable of allocating memory from a specified NUMA node and marking it as write-back or write combining as requested. Libraries such as `libnuma` also provide similar functionality (e.g., `numa_alloc_onnode`). However, Linux does not provide convenient ways to mark specific ranges of virtual memory as write combining from user space. We therefore build this dedicated driver to change caching modes directly and then export the memory to user space through the `mmap` interface. Using the driver, we allocate write-back memory for TX-D variants, and write combining memory for TX-V variants.

Benchmarks. We run the TATP [30], TPC-B and TPC-C [42] benchmarks to test the scalability of different variants. TATP models a cell phone service database with very small transactions, stressing the locking and logging subsystems. TPC-B models an update-intensive banking workload and we run its only transaction "Account Update". For TATP and TPC-C, we run both the full mix and write intensive transactions ("Update Location" in TATP and "Payment" in TPC-C) to show the general and stressed performances.

For each benchmark and variant, we report the average throughput (in thousand transactions per second, or kTPS) of five 30-second runs under different system loads (by varying the number of transaction threads). All workloads have a scaling factor of 48 to saturate the system and avoid deadlocks, so internal storage engine bottlenecks are responsible for any observed performance differences. We use `perf`[3] to collect stack traces and turn them into time breakdowns to identify the CPU cycles consumed by different components.

5.2 Distributed vs. Centralized Logging

As we have shown in Figure 1, the locking bottleneck is mostly eliminated by SLI and ELR, but logging gradually becomes a bottleneck as system load increases, even with a state-of-the-art centralized log. Figure 4 compares the scalability of centralized and distributed logging when running write-intensive workloads. The y-axis shows the throughput as a function of core count on the x-axis. Due to heavy log contention, neither Baseline nor Aether scales well as the system approaches saturation.

The other lines in Figure 4 plot the throughput of both distributed logging variants, assuming a durable cache system. We use durable cache variants to highlight the scalability improvement brought by distributed logging (Section 5.3 analyzes the performance impacts of passive group commit

[3]Details at `http://perf.wiki.kernel.org`.

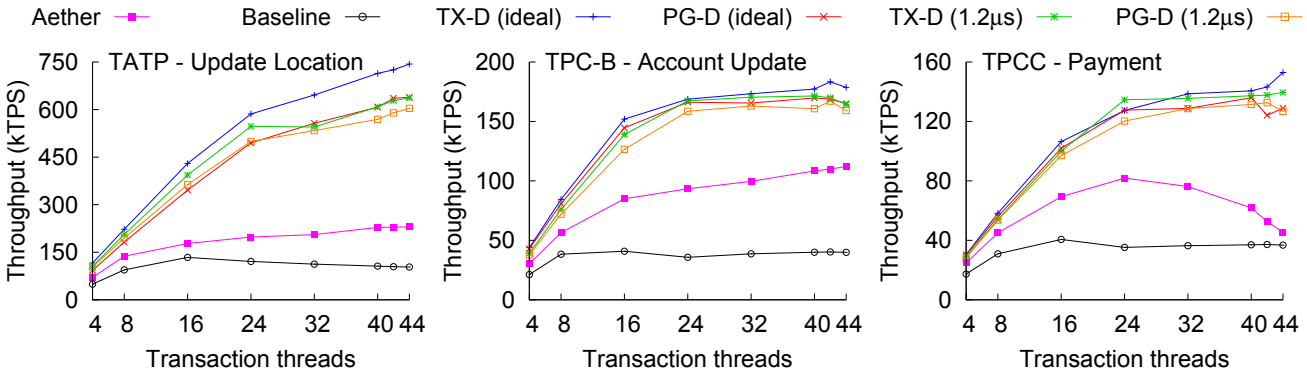

Figure 4: Scalability of distributed logging with write-intensive workloads. Distributed logging scales better as system load increases. Transaction-level partitioning outperforms page-level due to local memory access.

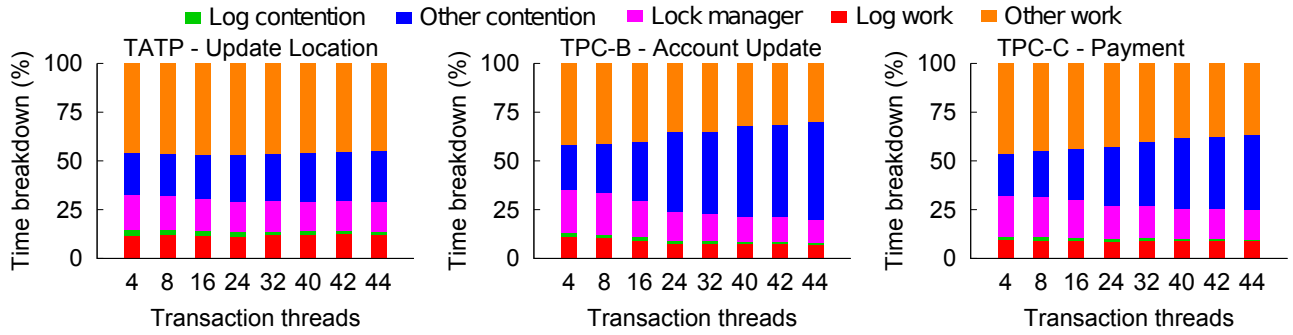

Figure 5: Time breakdowns for write-intensive transactions with transaction-level log partitioning.

for existing volatile-cache systems). For comparison, we also include the variants with NVM delays (labeled as "1.2μs", while those without these delays as "ideal"). However, in future real durable cache systems, we do not expect any such delays, even if the NVM is slower than DRAM, as the non-volatile CPU cache (in write-back mode) will be fast enough to hide the latency and provide at least near-DRAM performance. The performance numbers for these variants with extra delays show a lower bound and are for reference only. For TATP, Figure 4 shows that both TX-D and PG-D (with and without the delay) achieve near-linear scalability up to 24 threads, the number of physical cores in our system. TX-D (ideal) achieves a speedup from 1.6x – 3.2x over Aether. The numbers for PG-D (ideal) are slightly lower, ranging from 1.3x – 2.7x. Figure 4 shows a similar trend for TPC-B and TPC-C. As system load increases, throughput scales up more slowly due to bottlenecks outside the log manager, especially after 24 threads. The corresponding time breakdowns for TX-D (ideal) in Figure 5, clearly show that logging continues to perform well, but other sources of contention (e.g., contention in the buffer manager) gradually become larger for TPC-B and TPC-C. The growth of throughput in Figure 4 also becomes slower or even stopped. The same breakdowns for TATP do not show significant increase in contention as system load increases, explaining the superior scalability of the microbenchmark vs. TPC-B and TPC-C. In most cases, the imposed NVM delay shows an overhead of up to 15% for all the three workloads. However, in other cases the "1.2μs" variants actually achieved higher throughput (e.g., TPC-C with 24 threads) because the delay slowed down the system and thus reduced contention in components outside the log manager. Threads will spend more time in the log manager, resulting in fewer threads contending for other components at the same time. For example, running TPC-C – Payment under TX-D (1.2μs) with 24 threads increased 4.09% of log work, and reduced contention for the log and other components by 0.99% and 1.16%, respectively.

To show the performance of distributed logging and compare it with centralized counterparts in more ordinary cases, we repeat the same experiment with the full TATP and TPC-C benchmark mix (TPC-B only has one transaction, which is also write-intensive). Figure 6 shows the result for running the these workloads which have less contention and more read operations. For clarity, we only show the "ideal" variants. With fewer writes, Aether scales almost equally well as both distributed logging variants for TATP, though the throughput is slightly lower. Baseline keeps a similar trend as before, and cannot scale as system load increases. For TPC-C, both Baseline and Aether scaled up to 16 threads, after which only distributed logging variants can scale. Similar to the write-intensive case we discussed earlier in this section, after 32 threads, throughput numbers stop growing or even start to drop as the system approaches full saturation. Clearly passive group commit has an effect on this phenomenon as both TX-V and PG-V have lower throughput after 32 threads. However, PG-D also sees a drop at 40 threads. As we will show in Section 5.3, passive group commit is not the major cause. We plot the time breakdown graph for TX-V in Figure 7, in which "other work" increased by ~4% from 32 to 44 threads with de-

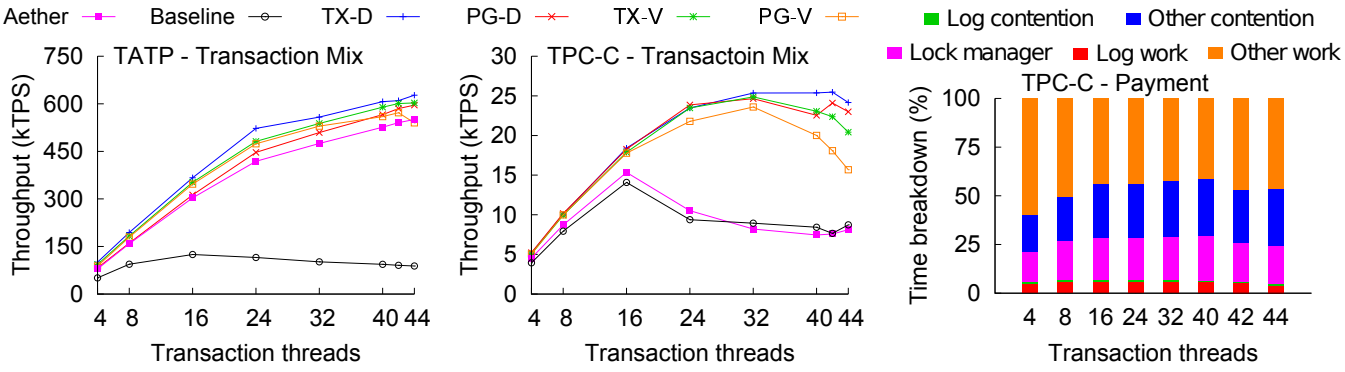

Figure 6: Scalability of distributed logging with more realistic TATP (left) and TPC-C (right) workloads (transaction mixes).

Figure 7: Time breakdown for TPC-C full mix with TX-V.

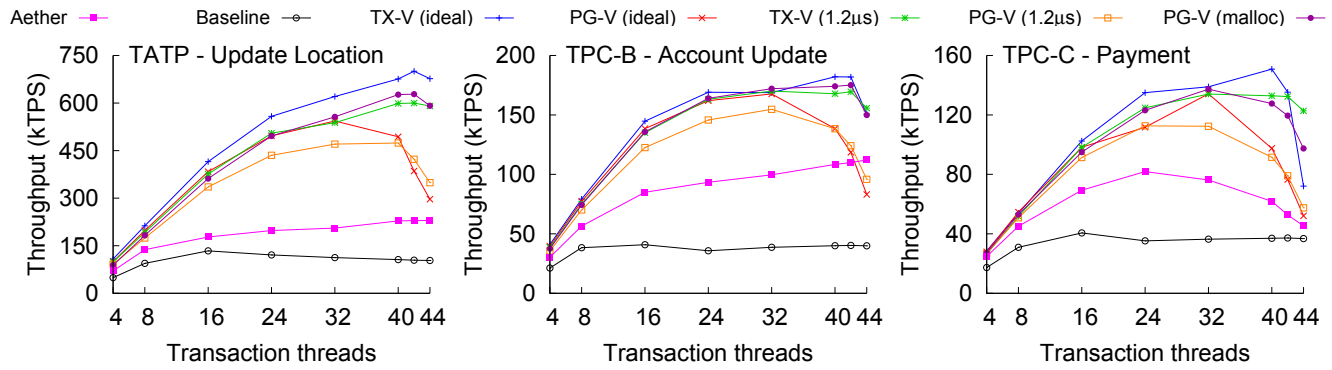

Figure 8: Scalability of distributed logging with passive group commit. PG-V cannot scale after 32 threads due to single-node log buffer allocation. However, PG-V with `malloc` shows similar scalability to TX-P.

creased contention. More time is spent on other work which is responsible for the slowdown, but log related contention is low and does not change much as system load increases.

Compared to TX-D (ideal), PG-D (ideal) achieved lower throughput in all benchmarks for all experiments we have conducted so far, because PG-D accesses log buffers in remote memories. Profiling both TX-D and PG-D, when running the Update Location transaction from TATP with 24 threads, we observe nearly 9% more off-core requests with PG-D (by comparison, Baseline produces 18% more requests). Based on these results, we conclude that a distributed log can improve performance on multi-socket under both high and low contentions, and transaction-level partitioning can avoid NUMA effects for even higher throughput.

5.3 Impact of Passive Group Commit

We explore the performance impact of passive group commit, which is required to protect committed work in today's volatile-cache machines. Passive group commit imposes overhead mostly when it signals stragglers, though we expect the signaling should be rare.[4] We show the throughput in Figure 8 when running distributed logging variants with passive group commit. Similar to durable cache variants, TX variants scale well as core count increases. When running TATP's Update Location transaction, both TX-V and PG-V lose throughput when compared to the durable cache variants in Figure 4. The overhead ranges from 4% to 10% for TX-V (ideal). A similar trend is observed for PG-V (ideal) until the core count reaches 32, after which the performance could drop as much as 50%. TPC-B and TPC-C results show the similar trend. Compared to the TX-V variant in Figure 6, when running TPC-C, TX-V in Figure 8 does not show a performance drop as core count increases. Note that in Figure 8 TX-V runs a more write-intensive workload. Thus the extra other work incurred by the transaction mix workload is the real major cause of the overhead, not passive group commit. Note that PG-V (ideal) and PG-V (1.2μs) use our simplistic driver to allocate WC memory from a single NUMA node, concentrating log traffic to a single node and taxing the machine's interconnect. The impact becomes more severe when we impose NVM delays. A more sophisticated strategy, such as the one used by `malloc`, would stripe the allocation over all nodes to distribute memory traffic more evenly. In Figure 8 we also show a PG-V variant using `malloc`, ("PG-V (malloc)") to estimate the real performance that passive group commit could achieve if the driver were improved to stripe the log across all nodes.

Compared to PG-V (ideal), PG-V (malloc) in Figure 8 shows similar scalability to TX-V (ideal), except that the numbers are slightly lower, which is in line with our results on TX-V (ideal) and PG-V (ideal). Note that as the system approaches full saturation (e.g., with 42 threads), daemon threads including the passive group commit daemon and buffer flusher start to affect performance. Therefore perfor-

[4]In our experiments, fewer than one in 20,000 commits requires a signal, though this could vary with workload.

mance numbers after 42 threads are only for completeness. Results from PG-V (malloc) also indicate that the performance difference between write-back and write combining on write-only access is minuscule as the throughput numbers are very close before 32 threads. After 32 threads, PG-V (malloc) maintained the same trend as TX-V (ideal) for both workloads. Since we also use `malloc` to allocate log buffers for PG-D variants, the scalability of PG-D is similar to that of TX-D, as shown in Figure 4.

Based on the above results and analysis, we have shown the real overhead of passive group commit, and conclude that passive group commit is very lightweight and is suitable for current multi-socket systems. We also verify that write combining imposes little overhead when compared to write-back in terms of writes [2, 15]. The performance of write combining is not a major concern in distributed logging.

5.4 Log Space Partitioning

We summarize the differences between and implications of page and transaction level partitioning. As discussed in Sections 5.2 and 5.3, transaction-level partitioning generally has higher throughput on multi-socket machines. With the high performance interconnect, OS scheduling and NUMA-aware memory allocation, page-level partitioning only induces a small amount of overhead. However, without NUMA-aware memory allocation, page-level partitioning cannot scale as core count reaches the system limit due to the NUMA effect. Compared to transaction-level, page-level partitioning is more suited for systems which also partition data by processor locations [33, 34]. Several logs could be allocated for each data partition, turning log insert into operations local to the socket on which the thread is running.

Page-level partitioning is more naturally supported by ARIES. The implementation is also simpler. For example, GSN is not required unless passive group commit is needed. In terms of individual components, page-level partitioning requires more changes in the transaction manager, as a transaction will touch multiple logs. On the contrary, transaction-level partitioning requires more changes in the buffer manager, while changes in the transaction manager are more straightforward with the help of GSN. Based on Aether in Shore-MT, we added, modified and removed 9644 LoC in total to implement page-level partitioning, while the number for transaction-level is 10280. These numbers include GSN and passive group commit implementations. The net changes required without GSN will be fewer. The majority of code changes were simply refactoring to replace "lsn" with "gsn" in variable names; the actual implementation of distributed logging only occupied about 2kLoC (much of which was deletions from the removal of Aether).

Based on the above analysis, we argue that on multicore and multi-socket hardware, transaction-level is more preferred over page-level log space partitioning. Though the implementation is somewhat invasive, due to refactoring involved, it does not require significantly more design effort.

6. RELATED WORK

Most related work focused on adopting new storage technologies and utilizing massively parallel hardware.

Leveraging new storage technologies. There has been active research on using NVM to reduce logging overhead, e.g., remove the disk and use NVM as the sole logging device [10]. It requires two (expensive) epoch barriers [9] per log write and partial overwriting to tolerate holes in the log (necessary to improve parallelism), and is thus not implementable on current hardware; the partial overwriting also accelerates device wear-out. PCMLogging [11] employs PCM as both data buffer and log space. Pelley et al. [35] demonstrated different uses of NVM in database systems, but also relies on the epoch barrier. MARS [8] removes LSN and checkpointing, and moves most functionality of logging into hardware. FlashLogging [5] uses multiple USB flash sticks to boost logging performance. Flag Commit [31] embeds transaction status into flash pages by a chain of commit flags to minimize the need of logging. Although these proposals differ in how they achieve improved logging performance, they all remain thoroughly centralized and thus subject to the kinds of contention we propose to eliminate.

Utilizing parallel hardware. Group and asynchronous commit [12, 37] are two widely used techniques for reducing logging overhead. Group commit aggregates multiple commit requests into a single I/O. It improves throughput at the expense of transaction response times. Asynchronous commit allows transactions to complete without waiting for the log records to be de-staged to disk. It completely eliminates log flush delays, but risks losing committed work upon failure due to the volatility of DRAM. Recent research focused more on parallel hardware. Aether [21] allows locks to be released earlier to reduce lock contention due to long log flush waits, reducing the impact of policies like group commit. Aether also reduces log contention, but those mechanisms rely on shared caches and do not perform as well in multi-socket NUMA environments [22]. C-ARIES [39] enhances ARIES with multi-threaded recovery, without changing log insertion (or the contention that accompanies it) in any way.

7. CONCLUSION

NVM is fundamentally changing the landscape of transaction logging. Its non-volatility and byte-addressability potentially invalidate the assumption of *flush-before-commit* and thus enable distributed logging to ease the logging bottleneck. However, the volatile nature of existing processor caches vetoes the option of blindly replacing the centralized log with a distributed one. The increasing popularity of multi-socket hardware today also poses challenges in adopting a distributed log. In this paper, we have discussed and proposed solutions to these challenges, in particular log space partitioning. We have shown that a distributed log could provide near-linear scalability and up to more than 3x speedup over state-of-the-art centralized logging on multi-socket hardware. In particular, transaction-level partitioning is more favorable by allowing us to allocate log buffers from local memory, avoiding cross-socket log buffer access. Leveraging existing hardware, we have proposed passive group commit which is very lightweight to protect committed work. Finally, we expect that durable processor caches will be the ultimate solution to building reliable and high performance NVM-based systems.

8. REFERENCES

[1] AgigaTech. AGIGARAM non-volatile system, 2013.

[2] Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. Implications of CPU caching on byte-addressable non-volatile memory programming. *HP Technical Report HPL-2012-236*, 2012.

[3] Buck, D. A. Ferroelectrics for digital information storage and switching. *MIT Report R-212*, 1952.

[4] Caulfield, A. M., et al. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. *SC*, pp. 1–11, 2010.

[5] Chen, S. FlashLogging: exploiting flash devices for synchronous logging performance. *SIGMOD*, pp. 73–86, 2009.

[6] Chen, S., Gibbons, P. B., and Nath, S. Rethinking database algorithms for phase change memory. *CIDR*, pp. 21–31, 2011.

[7] Cho, S. and Lee, H. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. *MICRO*, pp. 347–357, 2009.

[8] Coburn, J., et al. From ARIES to MARS: Transaction support for next-generation, solid-state drives. *SOSP*, pp. 197–212, 2013.

[9] Condit, J., et al. Better I/O through byte-addressable, persistent memory. *SOSP*, pp. 133–146, 2009.

[10] Fang, R., et al. High performance database logging using storage class memory. *ICDE*, pp. 1221–1231, 2011.

[11] Gao, S., et al. PCMLogging: reducing transaction logging overhead with PCM. *CIKM*, pp. 2401–2404, 2011.

[12] Gawlick, D. and Kinkade, D. Varieties of concurrency control in IMS/VS fast path. *HP Technical Report TR-85.6*, 1985.

[13] Graefe, G. A survey of b-tree logging and recovery techniques. *TODS*, 37(1):1:1–1:35, 2012.

[14] Hosomi, M., et al. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. *IEDM*, pp. 459–462, 2005.

[15] Intel. *Write Combining Memory Implementation Guidelines*, 1998.

[16] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2007.

[17] Jiang, L., et al. Improving write operations in MLC phase change memory. *HPCA*, pp. 1–10, 2012.

[18] Jog, A., et al. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. *DAC*, pp. 243–252, 2012.

[19] Johnson, R., Pandis, I., and Ailamaki, A. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, pp. 479–489, 2009.

[20] Johnson, R., et al. Shore-MT: a scalable storage manager for the multicore era. *EDBT*, pp. 24–35, 2009.

[21] Johnson, R., et al. Aether: a scalable approach to logging. *PVLDB*, pp. 681–692, 2010.

[22] Johnson, R., et al. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDBJ*, pp. 239–263, 2012.

[23] Kawahara, T. Scalable spin-transfer torque RAM technology for normally-off computing. *IEEE Design Test of Computers*, 28(1):52–63, 2011.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM*, pp. 558–565, 1978.

[25] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. Architecting phase change memory as a scalable DRAM alternative. *ISCA*, pp. 2–13, 2009.

[26] Lomet, D., Anderson, R., Rengarajan, T., and Spiro, P. *How the Rdb/VMS data sharing system became fast*, 1992.

[27] Mellor, C. HP 100TB Memristor drives by 2018 if you're lucky, admits tech titan. *The Register*, 2013.

[28] Mohan, C., et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial roll backs using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[29] Narayanan, D. and Hodson, O. Whole-system persistence. *ASPLOS*, pp. 401–410, 2012.

[30] Neuvonen, S., Wolski, A., Manner, M., and Raatikka, V. Telecom Application Transaction Processing Benchmark.

[31] On, S. T., et al. Flag Commit: Supporting efficient transaction recovery in flash-based DBMSs. *TKDE*, 24(9):1624–1639, 2012.

[32] Ooishi, M. Rohm demonstrates nonvolatile CPU, power consumption cut by 90%. *Tech-On!*, 2007.

[33] Pandis, I., Johnson, R., Hardavellas, N., and Ailamaki, A. Data-oriented transaction execution. *PVLDB*, pp. 928–939, 2010.

[34] Pandis, I., Tözün, P., Johnson, R., and Ailamaki, A. PLP: Page latch-free shared-everything OLTP. *PVLDB*, pp. 610–621, 2011.

[35] Pelley, S., Wenisch, T. F., Gold, B. T., and Bridge, B. Storage management in the NVRAM era. *PVLDB*, pp. 121–132, 2014.

[36] Qureshi, M. K., et al. Enhancing lifetime and security of PCM-based main memory with Start-gap wear leveling. *MICRO*, pp. 14–23, 2009.

[37] Ramakrishnan, R. and Gehrke, J. *Database Management Systems*. McGraw-Hill, 3rd edn., 2002.

[38] Saadeldeen, H., et al. Memristors for neural branch prediction: A case study in strict latency and write endurance challenges. *CF*, pp. 26:1–26:10, 2013.

[39] Speer, J. and Kirchberg, M. C-ARIES: A multi-threaded version of the ARIES recovery algorithm. *Database and Expert Systems Applications*, pp. 319–328, 2007.

[40] Stonebraker, M., et al. The end of an architectural era: (it's time for a complete rewrite). *PVLDB*, pp. 1150–1160, 2007.

[41] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[42] Transaction Processing Performance Council. TPC benchmarks B and C.

[43] Viking Technology. ArxCis-NV NVDIMM, 2013.

[44] Wong, H. S. P., et al. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[45] Zhao, J., et al. Kiln: closing the performance gap between systems with and without persistence support. *MICRO*, pp. 421–432, 2013.

[46] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. A durable and energy efficient main memory using phase change memory technology. *ISCA*, pp. 14–23, 2009.