

Transaction Logging Unleashed with NVRAM*

Tianzheng Wang Ryan Johnson

University of Toronto

{tzwang, ryan.johnson}@cs.toronto.edu

Most single node database systems such as PostgreSQL use *centralized* write-ahead logging [4]: log records are first buffered in DRAM and then flushed to stable storage upon commit to ensure durability and consistency. On today’s massively parallel hardware, however, this single log buffer design has become a bottleneck: as shown in Figure 1, on a 4-socket, 24-core server with hyper-threading (the OS sees 48 “CPUs”) logging gradually became the worst bottleneck with increasing parallelism: ~45% of the CPU cycles are spent on contending for the log buffer (in green color).

In this paper, we advocate using byte-addressable, non-volatile memory (NVRAM) as *log buffers*. The non-volatility and byte-addressability of NVRAM resurrect distributed logging—a once prohibitive technique for single node systems—to scale databases on modern hardware. We discuss the challenges and solutions brought by NVRAM and distributed logging. All of our techniques have been implemented based on the Shore-MT [1] storage manager. Since NVRAMs built with new materials (e.g., PCM) are not yet widely available, we consider flash-backed DRAM¹ in our evaluation. We run a write-intensive telecommunications transaction (TATP Update Location) and the TPC-C benchmark. Evaluation results show that our approach can significantly ease the logging bottleneck and achieve ~3x speedup over Aether [2], a state-of-the-art centralized logging scheme.

1. Resurrecting distributed logging with NVRAM

Under centralized logging, transaction threads contend on and acquire the lock (mutex) that protects the log buffer, insert to it, and then release the mutex. A natural way to ease this contention is via distributed logging: use multiple log buffers, instead of one (note that group commit does not help ease the contention: it only makes log flushing faster). However, distributed logging was not practical due to the needs for (1) dependency tracking, and (2) flushing log buffers upon transaction commit, i.e., *flush-before-commit*.

Consider two transactions T1 and T2 with their own log buffers L1 and L2, respectively. T1 updates a record R1 in page P, logs it in L1, and continues to do other operations. After this, T2 updates R2 which also resides in P, and logs it in L2. When T2 commits, not only L2 has to be flushed to storage, but also L1, as T1 modified the same page earlier than T2 did. For consistency and correct recovery, both log buffers have to be flushed, even though only T2 wants to commit. With more threads and log buffers—the usual case on modern hardware—the overhead of such dependency tracking becomes much higher. The slow storage (compared to DRAM) further worsens this situation due to the flush-before-commit requirement. Therefore, although distributed logging can significantly reduce contention, it is deliberated avoided in most systems [2].

* This work was originally published in VLDB 2014 [5].

¹ Flash-backed DRAM works the same as normal DRAM when the system is running. Upon failures, data are flushed to flash memory with the energy provided by a supercapacitor. Data are loaded back to DRAM upon restart.

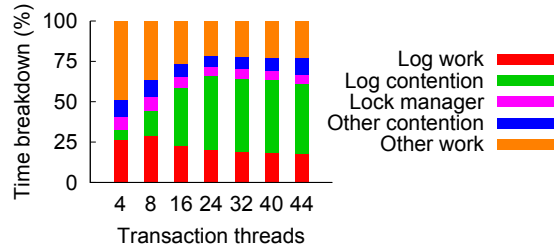


Figure 1. Time breakdown of running a write-intensive workload on a modern database system. Log contention is a major bottleneck.

Using NVRAM as *log buffers* solves these problems of distributed logging: NVRAM’s non-volatility invalidates *flush-before-commit*, as log records are durable once they are written in NVRAM. However, it is not straightforward to adopt either NVRAM or distributed logging. Safely and correctly adopting NVRAM requires all relevant log records hit NVRAM upon commit, instead of being buffered in the CPU cache; choosing the way of distributing log records also has performance implications, especially with NUMA hardware. We next highlight these challenges and solutions.

2. Database design and performance implications

Distributing the log poses two major challenges: how to assign log records to multiple logs (log space partitioning) and recover the system from multiple logs. Log records could be distributed by database page or transaction: storing all log records for a page or transaction in the same log. Redo can be parallelized by pages, while undo favors partitioning by transactions, which can be rolled back in parallel. We next discuss the major impacts on database design and performance implications brought by these challenges.

Uniqueness of log records. In a centralized log, the log sequence number (LSN) uniquely identifies each log record. But LSNs are not unique in a distributed log, as they only indicate a record’s position within the log that holds it. Such lack of ordering among records from different logs can cause errors such as applying older records on top of newer ones that have smaller LSNs.

Our solution is to maintain a *global sequence number* (GSN) based on logical clock [3] in each page, transaction and log. Pinning a page sets the page and transaction GSNs to $\max(tx\ GSN, page\ GSN) + 1$ if the transaction intends to modify the page; otherwise we omit the “+1”. Inserting a log record sets the *tx*, *page*, and *log* GSNs to $\max(tx\ GSN, page\ GSN, log\ GSN) + 1$. GSNs are partially ordered, but they uniquely identify and provide a total order for log records belonging to any one page, log, or transaction.

NUMA effects. The shift to multi-socket systems brings non-uniform memory access (NUMA), where memory is either “local” or “remote”, and accessing the latter is much more expensive. Threads could run on arbitrary CPUs and spread insertions over any

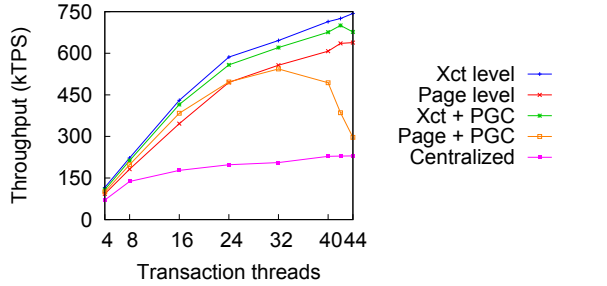


Figure 2. Throughput for a write-intensive telecommunications transaction. Transaction-level partitioning outperforms page-level because of local memory access.

log buffer, which could be allocated in any NUMA node (or striped over all of them). If the log buffer and the thread are not within the same node, log insertion will involve remote accesses, which are almost inevitable in if we partition the log by page. Partitioning by transaction fits directly with NUMA as each transaction’s writes go to one log: a log could be allocated in each node, with transactions assigned to logs based on the CPU they run on. This approach eliminates the NUMA effects during log insertion, improving logging performance and freeing up interconnect bandwidth.

We have evaluated the impact of NUMA effects with a write-intensive telecommunications transaction (TATP Update Location) which consists of frequent and small updates, giving enough stress on logging. As shown in Figure 2, both ways of partitioning scale and outperform Aether (“Centralized”), and transaction-level achieves higher throughput than page-level partitioning. As hardware become increasingly NUMA, the gap is likely to grow.

Recovery. When restarting from a failure, the database will redo log records and undo transactions that did not successfully commit. A page-partitioned log can redo in parallel easily: each thread can work with pages from a single log. Undo will require building up per-transaction logs, which might be expensive because usually most transactions have committed with only a few losers left to undo. Parallel undo is straightforward for a transaction-partitioned log as all the records to undo a transaction reside in one log. Redo is more complex because dependencies among pages can reside in any log. A naïve parallel replay of all logs could apply log records for the same page in arbitrary order, potentially skipping certain records. A two-step redo pipeline can solve this problem efficiently: the first stage uses N threads to scan the logs and partition log records into M buckets by page ID. M threads then sort each bucket and apply the changes in parallel, without risk of skipped records. Full parallelism can be achieved with well-tuned N and M values.

3. NVRAM based logging: now and the future

Modern CPUs rely on caching for good performance. When using NVRAM as log buffers, log records are first cached by the CPU and then written to NVRAM upon cache flush. Records that are not yet persistent in NVRAM could be lost upon failures. Therefore, we need to ensure that all the log records belonging to a transaction, in all logs, written on any CPU, are persistent upon commit. Naïve solutions might disable the CPU cache or flush it after each write. Neither is efficient, given that most NVRAMs have slower writes.

We propose **passive group commit**, a lightweight group commit protocol to solve this problem efficiently. It ensures that all log records written by the committing transactions are persistent in NVRAM before notifying the client that commit succeeded. We maintain a thread-local variable $dgsn$ to record the GSN of the log record the thread can guarantee to be persistent: upon commit, the

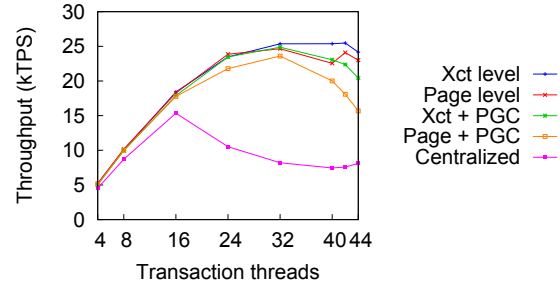


Figure 3. Throughput for TPC-C. Distributed logging scales well until 24 threads, as the server has 24 physical cores and it is well-known that hyper-threading is not effective for database workloads.

transaction flushes the CPU cache (or empties WC buffers by issuing a memory fence if write-combining is used) and stores the last log record’s GSN as $dgsn$. The transaction is then added to a commit queue. The group commit daemon periodically examines the $dgsn$ of each thread and records the smallest $dgsn$ as $mgsn$. Log records written on any CPU with a $GSN \leq mgsn$ are guaranteed to be persistent. The daemon dequeues transactions once their commit records precedes $mgsn$ and notifies the client. If any thread takes too long to update its $dgsn$, the daemon will send a signal to the straggler, which then persists log records and updates $dgsn$.

Figures 2 and 3 show the throughput numbers of running Update Location and TPC-C, respectively. In these experiments, we set the caching mode for log buffers to write-combining for easier straggler handling (sending a signal implicitly drains the WC buffers). For both workloads, passive group commit (“PGC”) incurs only a small amount of overhead. For comparison, the “Page + PGC” variant allocates NVRAM from a single socket, thus seeing dropping performance after 32 threads due to the NUMA effect.

Durable processor cache. Although it solves the durability problem, passive group commit is still a stop-gap solution and we argue that the ultimate solution is durable CPU cache built by NVRAMs such as FeRAM or even capacitor-backed SRAM which will drain the cached data to NVRAM upon failures. NVRAM can then be treated like normal memory and software need only issue memory fences to force ordering when necessary. For a distributed log, passive group commit will no longer be necessary as all writes are immediately durable.

Hardware/OS Support. To use NVRAM in existing systems, the memory controller should distinguish NVRAM and DRAM for the OS to manage them. Transaction-level log space partitioning requires allocating NVRAM from specified NUMA nodes. It is therefore desirable to have some NVRAM on each NUMA node and a software interface similar to `numa_allloc_onnode` for NVRAM.

References

- [1] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. *EDBT*, pages 24–35, 2009.
- [2] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, pages 681–692, 2010.
- [3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, pages 558–565, 1978.
- [4] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial roll backs using write-ahead logging. *TODS*, 17(1):94–162, 1992.
- [5] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.