Aether: A Scalable Approach to Logging

Ryan Johnson[†‡] Ippokratis Pandis[†‡] Radu Stoica[‡] Manos Athanassoulis[‡]
Anastasia Ailamaki[†‡]

[†]Carnegie Mellon University
5000 Forbes Ave. Pittsburgh, PA 15213, USA
{ryanjohn, ipandis}@ece.cmu.edu

[‡]École Polytechnique Fédérale de Lausanne
EPFL, CH-1015 Lausanne, Switzerland
{radu.stoica, manos.athanassoulis, natassa}@epfl.ch

ABSTRACT

The shift to multi-core hardware brings new challenges to database systems, as the software parallelism determines performance. Even though database systems traditionally accommodate simultaneous requests, a multitude of synchronization barriers serialize execution. Write-ahead logging is a fundamental, omnipresent component in ARIES-style concurrency and recovery, and one of the most important yet-to-be addressed potential bottlenecks, especially in OLTP workloads making frequent small changes to data.

In this paper, we identify four logging-related impediments to database system scalability. Each issue challenges different level in the software architecture: (a) the high volume of small-sized I/O requests may saturate the disk, (b) transactions hold locks while waiting for the log flush, (c) extensive context switching overwhelms the OS scheduler with threads executing log I/Os, and (d) contention appears as transactions serialize accesses to in-memory log data structures. We demonstrate these problems and address them with techniques that, when combined, comprise a holistic, scalable approach to logging. Our solution achieves a 20%-69% speedup over a modern database system when running log-intensive workloads, such as the TPC-B and TATP benchmarks. Moreover, it achieves log insert throughput over 1.8GB/s for small log records on a single socket server, an order of magnitude higher than the traditional way of accessing the log using a single mutex.

1. INTRODUCTION

Recent changes in computer microarchitecture have led to multicore systems, which in turn have several implications in database management systems (DBMS) software design [6]. DBMS software was designed in an era during which most computers were uniprocessors with high latency I/O subsystems. Database engines therefore excel at exploiting *concurrency* –support for multiple in-progress operations– to interleave the execution of a large number of transactions, most of which are idle at any given moment. However, as the number of cores per chip increases in step with Moore's law, software must exploit *parallelism* –support for concurrent operations to proceed simultaneously– to benefit from new hardware. Although database workloads exhibit high concurrency, internal bottlenecks [11] often mean that database engines cannot extract enough parallelism to meet multicore hardware demands.

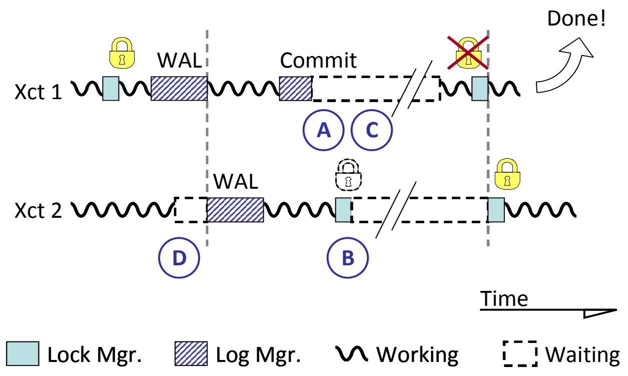

Figure 1. A timeline of two transactions illustrating four kinds of log-imposed delay: (A) I/O-related delays, (B) increased lock contention, (C) scheduler overload, and (D) log buffer contention.

The log manager is a key service of modern DBMSs, especially prone to bottlenecks due to its centralized design and dependence on I/O. Long flush times, log-induced lock contention, and contention for log buffers in main memory all impact scalability, and no single bottleneck is solely responsible for suboptimal performance. Modern systems can achieve transaction rates of 100ktps or higher, exacerbating the log bottleneck.[1] Research to date offers piecewise or partial solutions to the various bottlenecks, which do not lead to a fully scalable log manager for today's multicore hardware.

1.1 Write-ahead Logging and Log Bottlenecks

Nearly all database systems use centralized write-ahead logging (WAL) [14] to protect against data corruption and lost committed work after crashes. WAL allows transactions to execute and commit without requiring that all data pages they update be written to disk first. However, as Figure 1 illustrates, there are four main types of delays which logging can impose on transactions:

I/O-related delays (A). The system must ensure that a transaction's log records reach non-volatile storage before committing. With access times in the order of milliseconds, a log flush to magnetic media can easily become the longest part of a transaction. Further, log flush delays become serial if the log device is overloaded by multiple small requests. Fortunately, log flush I/O times

1. See, e.g. top published TPC-C results or performance figures reported by main-memory databases like H-Store [22].

become less important as fast solid-state drives gain popularity[1][12], and when using techniques such as group commit [8].

Log-induced lock contention (B). Under traditional WAL, each transaction which requests a commit must first flush its log records to disk, retaining all write locks until the operation completes. Holding locks during this final (and often only) I/O significantly increases lock contention in the system and creates an artificial bottleneck in many workloads. For example, the left-most bar in Figure 2 shows CPU utilization as 60 clients run for 95 seconds the TPC-B [24] benchmark in a modern storage manager [11] on a Sun Niagara II chip with 64 hardware contexts (see Section 6.1 for the detailed experimental setup). Due to the increased lock contention the system is idle 75% of the time. Section 3 shows that even though reduced I/O times help, the problem remains even when logging to a ramdisk with minimal latency.

Excessive context switching (C). Log flushes incur additional costs beyond I/O latency because the transaction cannot continue and must be descheduled until the I/O completes. Unlike I/O latency, context switching and scheduling decisions consume CPU time and thus cannot overlap with other work. In addition, the abundance of hardware contexts in multicore hardware can make scheduling a bottleneck in its own right as runnable threads begin to accumulate faster than the OS can dispatch them. The second bar in Figure 2 shows for the same workload the processing time of a system which suffers from the problem of OS scheduler overload. The system remains 20% idle even with transactions ready to run. We analyze excessive context switching problem in Section 4.

Log buffer contention (D). Another log bottleneck arises as the multicore trend continues to demand exponential increases in parallelism; where current hardware trends generally reduce the other bottlenecks (e.g. solid state drives reduce I/O latencies), each successive processor generation aggravates contention with an increase in hardware contexts. The third bar in Figure 2 shows that if we remove the problems of logical lock contention and excessive context switching, the system utilizes fully the available hardware. But, as a large number of threads attempt simultaneous log inserts, the contention for the centralized log buffer contributes a significant (and growing) fraction of total transaction time. We therefore consider this bottleneck as the most dangerous to future scalability, in spite of its modest performance impact on today's hardware. Section 5 focuses on this problem.

In summary, log bottlenecks arise for several reasons, and no single approach addresses them all. A technique known as "asynchronous commit" is perhaps the clearest symptom of the continuing log bottleneck. Available in most DBMSs (including Oracle [16] and PostgreSQL [17]) asynchronous commit allows transactions to complete and return results without waiting for their log entries to become durable. Skipping the log flush step sidesteps problems A-C listed above, but at the cost of unsafe operation: the system can lose committed work after a crash. To date no existing proposal addresses all the bottlenecks associated with log flush, and the looming problem of log buffer contention.

1.2 A Holistic Approach to Scalable Logging

This paper presents Aether, a complete approach towards log scalability, and demonstrates how the proposed solutions address all log bottlenecks on modern hardware, even for the most demanding workloads. Aether combines new and existing solutions to minimize or eliminate the log bottleneck. We highlight new contributions below.

First, we evaluate *Early Lock Release* (ELR), a promising technique for eliminating log-induced lock contention. ELR has been proposed several times in the past but, to our knowledge, has never been evaluated in the literature and is not used today by any

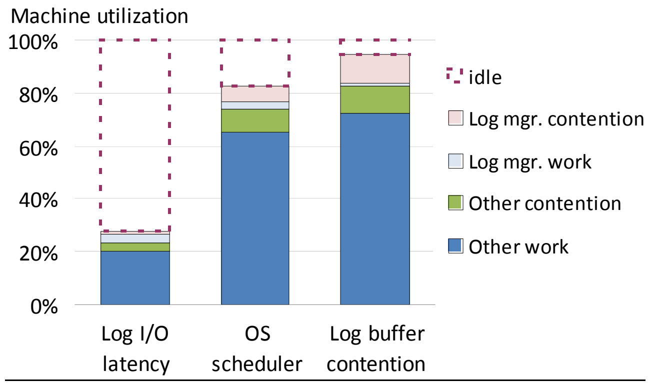

Figure 2. Breakdown of CPU time showing work and contention due to the log vs other parts of the system, when 60 clients run the TPC-B benchmark, as we remove log-related bottlenecks.

mainstream database engine. We show that, particularly for skewed accesses common to real workloads, ELR increases throughput by 15%-164% even when logging to fast flash disks.

Second, we propose and evaluate *Flush Pipelining,* a technique which allows most transactions to commit without triggering context switches. In synergy with ELR it achieves the same performance with asynchronous commit without sacrificing durability.

Finally, we propose and evaluate three improvements to log buffer design, including a new "consolidation-based backoff" scheme which allows threads to aggregate their requests to the log when they encounter contention. As a result, maximum log contention is decoupled from thread counts and log record sizes. Our evaluation shows that contention is minimized and identifies memory bandwidth as the most likely bottleneck to arise next.

2. RELATED WORK

As a core database service, logging has been the focus of extensive research. Virtually all database engines employ some variant of ARIES [14], a sophisticated write-ahead logging system which integrates concurrency control with transaction rollback and disaster recovery, and allows the system to recover fully even if recovery is interrupted repeatedly by new crashes. To achieve its high robustness with good performance, ARIES couples tightly with the rest of the system, particularly the lock and buffer pool managers, and has a strong influence on the design of access methods such as B+Tree indexes [13]. The log is typically implemented as a single global structure shared by every transaction, making it a potential bottleneck in highly parallel systems. Even in a single-threaded database engine the overhead of logging accounts for roughly 12% of the total time in a typical OLTP workload [7].

Several recent studies [12][3] evaluate solid state flash drives in the context of logging, and demonstrate significant speedups due to both better response times and also better handling of the small I/O sizes common to logging. However, even the fastest flash drives do not eliminate all overhead because synchronous log flush requests still block and therefore cause OS scheduling.

Log group commit strategies [8][18] reduce pressure on magnetic log disks by aggregating multiple requests for log flush into a single I/O operation; fewer and larger disk accesses translate into significantly better disk performance by avoiding unnecessary head seeks. Unfortunately, group commit does not eliminate unwanted context switches because transactions merely block pending notification from the log rather than blocking on I/O requests directly.

Asynchronous commit [16][17] extends group commit by not only aggregating I/O requests together, but also allowing transactions to complete without waiting for those requests to complete. This optimization moves log flush times completely off the critical path but at the expense of durability. That is, committed work can be lost if a crash prevents the corresponding log records to become durable. Despite being unsafe, asynchronous commit is used widely in commercial and open source database systems because it provides a significant performance boost. In contrast, Aether achieves this performance boost without sacrificing durability.

DeWitt et al. [4] observe that a transaction can safely release locks before flushing its log records to disk provided certain conditions are met. IVS [5] implemented this optimization but its correctness was proven more recently [21]. We refer to this technique as early lock release (ELR) and evaluate it further in Section 3.

Main-memory database engines impose a special challenge for log implementations because the log is the only I/O operation of a given transaction. Not only is the I/O time responsible for a large fraction of total response time, but short transactions also lead to high concurrency and contention for the log buffer. Some proposals go so far as to eliminate the log (and its overheads) altogether [22], replicating each transaction to multiple database instances and relying on hot fail-over to maintain durability. Aether is especially well-suited to in-memory databases because it addresses both log flush delays and contention at the log buffer.

3. MOVING LOG I/O LATENCY OFF THE CRITICAL PATH

During its lifecycle a transaction acquires database locks to ensure consistency and logs all actions before performing them. At completion time –after writing a commit record to non-volatile storage– the transaction finally releases the locks it has accumulated. Releasing the locks only after the commit record has reached disk (or been *flushed*) ensures that other transactions do not encounter uncommitted data, but also increases lock hold time significantly, especially for in-memory workloads where the log commit is the longest part of many transactions.

3.1 Early Lock Release

DeWitt et al. [4] observe that a transaction's locks can be released before its commit record is written to disk, as long as it does not return results to the client before becoming durable. Other transactions which read data updated by a *pre-committed transaction* become *dependant* on it and must not be allowed to return results to the user until both their own and their predecessor's log records have reached the disk. Serial log implementations preserve this property naturally, because the dependant transaction's log records must always reach the log later than those of the pre-committed transaction and will therefore become durable later also. Formally, as shown in [21], the system must meet two conditions for early lock release to preserve recoverability:

1. Every dependant transaction's commit log record is written to the disk after the corresponding log record of pre-committed transaction.
2. When a pre-committed transaction is aborted all dependant transactions must also be aborted. Most systems meet this condition trivially; they do no work after inserting the commit record, except to release locks, and therefore can only abort during recovery when all uncommitted transactions roll back.

Early Lock Release (ELR) removes log flush latency from the critical path by ensuring that only the committing transaction must wait for its commit operation to complete; having released all held database locks, others can acquire these locks immediately and

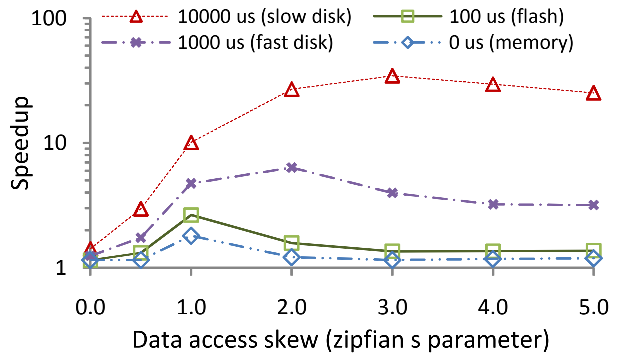

Figure 3. Speedup due to ELR when running the TPC-B benchmark and varying I/O latency and skew in data accesses.

continue executing. In spite of its potential benefits modern database engines do not implement ELR and to our knowledge this is the first paper to analyze empirically ELR's performance. We hypothesize that this is largely due to the effectiveness of asynchronous commit [16][17], which obviates ELR and which nearly all major systems do provide. However, systems which do not sacrifice durability can benefit strongly from ELR under workloads which exhibit lock contention and/or long log flush times.

3.2 Evaluation of ELR

We use the TPC-B benchmark [24] to evaluate ELR. TPC-B was designed as a database stress test and also exhibits significant lock contention. The benchmark executes on a 64-context Niagara II server running the Shore-MT storage manager [11] (further details about the platform and experimental methodology can be found in Section 6.1). Figure 3 shows the benefit of ELR over a baseline system as we vary the two major factors which impact its effectiveness: lock contention and I/O latency. The y-axis shows speedup due to ELR as the skew of zipfian-distributed data accesses increases along the x-axis. Lower skew leads to more uniform accesses and lower lock contention. Different log device latencies are given as data series ranging from 0 to 10ms. The first series (0ms) is measured using a ramdisk which imposes almost no additional delay beyond a round trip through the OS kernel (40-80µs). The remaining series are created by using a combination of asynchronous I/O and high resolution timers to impose additional response times of 100µs (fast flash drive), 1ms (fast magnetic drive), and 10ms (slow magnetic drive).

As shown in the graph, ELR's speedup is maximized (35x) for slower devices, but remains substantial (2x) even with flash drives if contention is present. This effect occurs because transactions are short even compared to 100µs I/O times, and ELR eases contention by removing that delay from the critical path. As write performance of most flash drives remains unpredictable (and usually slower than desired) ELR remains an important optimization even as systems move away from magnetic media.

Varying lock contention impacts performance in three phases. For very low contention, the probability of a transaction to request an already-held lock is low. Thus, holding that lock through the log flush does not stall other transactions and ELR has no opportunity to improve performance. At the other extreme, very high skew leads to such high contention that transactions encounter held locks even with no log flush time. In the middle range, however, ELR significantly improves performance because holding locks through log flush causes stalls which would not have arisen otherwise. The sweet spot becomes wider as longer I/O times stretch out the total transaction length in the baseline case. Finally, by way of

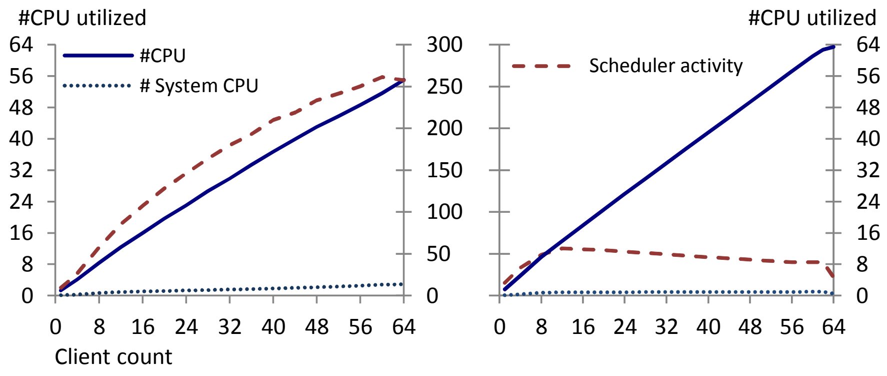

Figure 4. Total and system CPU utilization and number of context switches without (left) and with (right) flush pipelining.

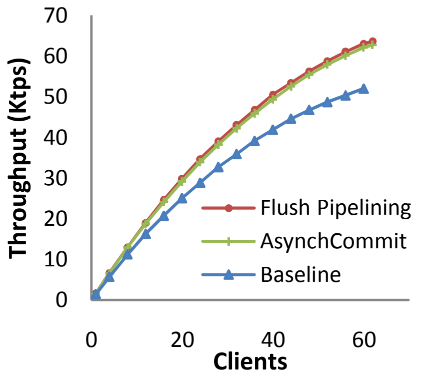

Figure 5. Performance of flush pipelining and asynchronous commit vs. baseline.

comparison, the intuitive rule that 80% of accesses are to 20% of the data corresponds roughly to a skew of 0.85. In other words, workloads are likely to exhibit exactly the contention levels which ELR is well-equipped to reduce.

In conclusion, we find that ELR is a straightforward optimization which can benefit even modern database engines. Further, as the next section demonstrates, it will become an important component in a safe replacement for asynchronous commit.

4. DECOUPLING OS SCHEDULING FROM LOG FLUSH OPERATIONS

The latency of a log flush arises from two sources: the actual I/O wait time and the context switches required to block and unblock the thread at either end of the wait. Existing log flush optimizations, such as group commit, focus on improving I/O wait time without addressing thread scheduling. Similarly, while ELR removes log flush from the critical path of other transactions (shown as (B) in Figure 1) the requesting transaction must still block for its log flush I/O and be rescheduled as the I/O completes (shown as (A) in Figure 1). Unlike I/O wait time, which the OS can overlap with other work, each scheduling decision consumes several microseconds of CPU time which cannot be overlapped.

The cost of scheduling and context switching is increasingly important for several reasons. First, high-performance solid state storage provides access times measured in tens of microseconds, leaving the accompanying scheduling decisions as a significant fraction of the total delay. Second, exponentially growing core counts make scheduler overload an increasing concern as the OS must dispatch threads for every transaction completion. The scheduler must coordinate these scheduling decisions (at least loosely) across all cores. The excessive context switching triggers a scheduling bottleneck which manifests as a combination of high load (e.g. many runnable threads) but low CPU utilization and significant system time.

Figure 4 (left) shows an example of the scheduler overload induced when the Shore-MT storage manager [11] runs the TPC-B benchmark [24] on a 64-context Sun Niagara II machine. As the number of client threads increases along the x-axis, we plot the rate of context switches (in thousands/s), as well as the CPU utilization achieved and the number of CPUs running inside the OS kernel (system time). The number of context switches increases steadily with the number of client threads.[2] The CPU utilization curve illustrates that the OS is unable to handle this load, as 12 of the 64 hardware contexts are idle. Further, as load increases an increasing fraction of total load is due to system time rather than the application, further reducing the effective utilization.

Excessive context switching explains why group commit alone is not fully scalable and why asynchronous commit is popular despite being unsafe. The latter eliminates context switching associated with transaction commit while the former does not.

4.1 Flush Pipelining

To eliminate the scheduling bottleneck (and thereby increase CPU utilization and throughput), the database engine must decouple the transaction commit from thread scheduling. We propose *Flush Pipelining*, a technique which allows agent threads to detach from transactions during log flush in order to execute other work, resuming the transaction once the flush is completed.

Flush pipelining operates as follows. First, agent threads commit transactions asynchronously (without waiting for the log flush to complete). However, unlike asynchronous commit they do not return immediately to the client but instead detach from the transaction, enqueue its state at the log and continue executing other transactions. A daemon thread triggers log flushes using policies similar to those used in group commit (e.g. "flush every X transactions, L bytes logged, or T time elapsed, whichever comes first"). After each I/O completion, the daemon notifies the agent threads of newly-hardened transactions, which eventually reattach to each transaction, finish the commit process and return results to the client. Transactions which abort after generating log records must also be hardened before rolling back. The agent threads handle this case as relatively rare under traditional (non-optimistic) concurrency control and do not pass the transaction to the daemon.

When combined with ELR (see previous section), flush pipelining provides the same throughput[3] as asynchronous commit without sacrificing any safety. Only the log's daemon thread suffers wait time and scheduling due to log flush requests, with agent threads pipelining multiple requests to hide even long delays.

4.2 Evaluation of Flush Pipelining

To evaluate flush pipelining we run the same experiment as in Figure 4 (left), but this time with flush pipelining active. Figure 4 (right) shows the result. As before we vary the number of client threads and measure the number of context switches (in millions), utilization achieved, and the OS system time contribution.

2. Daemon threads contribute a secondary effect. As load increases these threads wake more and more frequently at first, then sleep less and less, and finally revert to polling as the system becomes saturated.

3. Asynchronous commit does deliver superior response times for individual transactions (they do not wait for the log flush to complete), but the delays overlap perfectly and overall throughput is not impacted.

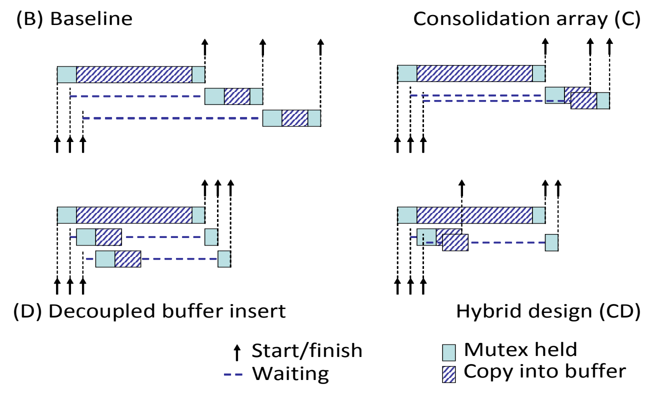

Figure 6. Illustrations of several log buffer designs. The baseline system can be optimized for shorter critical path (D), fewer threads attempting log inserts (C), or both (CD)

In contrast to the baseline case, the number of context switches after an initial increase, remains almost steady for the entire load spectrum. The utilization reaches the maximum possible (64) indicating that the scheduling bottleneck has been resolved. Further confirmation comes from the system time contribution, which remains nearly constant regardless of how many threads enter the system. This behavior is expected because only one thread issues I/O requests regardless of thread counts, and the group commit policy ensures that requests become larger rather than more frequent.

Figure 5 compares the performance of baseline Shore-MT to asynchronous commit and flush pipelining when running the TPC-B. The x-axis varies the number of clients as we plot throughput on the y-axis. Even with a fast log disk, the baseline system begins to lag almost immediately as scheduling overheads increase reducing its scalability. In contrast, the other two scale better achieving up to 22% higher performance. As Section 6.4 will show, for even more log-intensive workloads the benefits of flush pipelining are larger.

In summary, flush pipelining successfully and safely removes the log from the system's critical path of execution by breaking the correlation between transaction commits and scheduling.

5. SCALABLE LOG BUFFER DESIGN

Most database engines use some variant of ARIES, which assigns each log record a unique log sequence number (LSN). The LSN encodes a record’s disk address, acts as a timestamp for data pages written to disk, and serves as a pointer to log records both in memory and on disk. It is also convenient for LSN to serve as addresses in the log buffer, so that generating an LSN also reserves buffer space. In order to keep the database consistent in spite of repeated failures, ARIES imposes strict ordering constraints on LSN generation. While a total ordering is not technically required for correctness, valid partial orders tend to be too complex and interdependent to be worth pursuing as a performance optimization (see Section A.5 of the Appendix for further discussion). Inserting a record into the log buffer consists of three distinct phases:

1. **LSN generation and log buffer acquire**. The thread must first claim the space it will eventually fill with the intended log record. Though serial, LSN generation is short and predictable barring exceptional situations such as buffer wraparound or full log buffer
2. **Log record insertion**. The thread copies the log record in the buffer space it has claimed.
3. **Log buffer release**. The transaction releases the buffer space, which allows the log manager to write the record to disk.

A straightforward log insert implementation acquires a central mutex before performing all three phases and the mutex is released at the same time as the buffer (pseudocode in Algorithm 1, Appendix). This approach is attractive for its simplicity: log inserts are relatively inexpensive, and in the monolithic case buffer release is simplified to a mutex release.

The weakness of a monolithic log insert is that it serializes buffer fill operations –even though buffer regions never overlap– which adds their cost directly to the critical path. In addition, log record sizes vary significantly, making copying costs unpredictable. Figure 6(B) illustrates how a single large log record can impose long delays on later threads; this situation arises frequently in our system because the distribution of log records has two strong peaks at 40B and 264B (a 6x difference) and the largest log records can occupy several kB each.

To permanently eliminate contention for the log buffer, we seek to make the cost of accessing the log independent of both the sizes of the log records being inserted and the number of threads inserting them. The following subsections explore both approaches and propose a hybrid solution which combines them.

5.1 Consolidating Buffer Allocation

A log record consists of a standard header followed by an arbitrary payload. Log buffer allocation is composable in the sense that two successive requests also begin with a log header and end with an arbitrary payload. We exploit this composability by allowing threads to combine their requests into groups, carve up and fill the group’s buffer space off the critical path, and finally release it back to the log as a unit. To this end we extend the idea of elimination-based backoff [9][15], a hybrid approach combining elimination trees [19] with backoff. Threads which encounter contention back off, but instead of sleeping or counting cycles they congregate at an *elimination array*, a set of auxiliary locations where they attempt to combine their requests with those of others.

When elimination is successful threads satisfy their requests without returning to the shared resource at all, making the backoff very effective. For example, stacks are amenable to elimination because push and pop requests which encounter each other while backing off can cancel each other directly via the elimination array and leave. Similarly, threads which encounter contention for log inserts back off to a *consolidation array* and combine their requests before reattempting the log buffer. We use the term “consolidation” instead of “elimination” because, unlike with a stack or counter, threads must still cooperate after combining their requests so that the last to finish can release the group’s buffer space. Like an elimination array, any number of threads can consolidate into a single request, effectively bounding contention at the log buffer to the number of array entries protecting the log buffer, rather than the number of threads in the system. Algorithm 2 (Appendix) provides a sketch of the consolidation array-based buffer allocation.

The net effect of consolidation is that only the first thread from each group competes to acquire buffer space from the log, and only the last thread to leave must wait to release it. Figure 6(C) depicts the effect of consolidation; the first thread to arrive is joined by two others while it waits on the log mutex and all three proceed in parallel once the mutex acquire succeeds. However, as the figure also shows, consolidation leaves significant wait times because only buffer fill operations within a group proceed in parallel; operations between groups are still serialized. Given enough threads in the system, at least one thread of each group is likely to insert a large log record, delaying later groups.

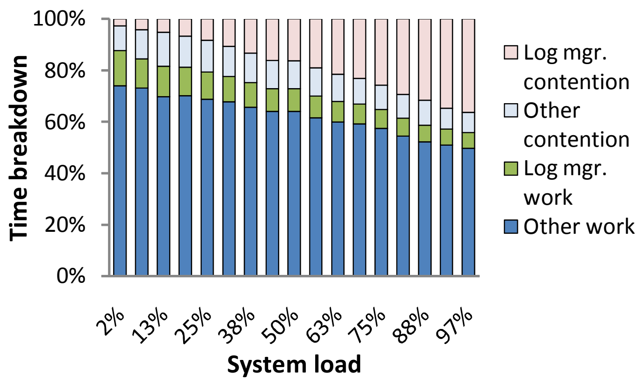

Figure 7. Breakdown of the execution time of Shore-MT with ELR and flush pipelining, running TATP-UpdateLocation transactions, as load increases. The log buffer becomes the bottleneck.

5.2 Decoupling Buffer Fill

Because buffer fill operations are not inherently serial (records never overlap) and have variable costs, they are highly attractive targets to move off the critical path. All threads which have acquired buffer regions can safely fill those regions in any order as long as they release their regions in LSN order. We therefore modify the original algorithm so that threads release the mutex immediately after acquiring buffer space. Buffer fill operations thus become pipelined, with a new buffer fill starting as soon as the next thread can acquire its own buffer region.

Decoupling log inserts from holding locks results in a non-trivial buffer release operation which becomes a second critical section. Like LSN generation, buffer release must be serialized to avoid creating gaps in the log. Log records must be written to disk in LSN order because recovery must stop at the first gap it encounters; in the event of a crash any committed transactions beyond a gap would be lost. No mutex is required, but before releasing its own buffer region, each thread must wait until the previous buffer has been released (Algorithm 3 of the appendix gives pseudocode).

With pipelining in place, arriving threads can overlap their buffer fills with that of a large log record, without waiting for it to finish first. Figure 6(D) illustrates the improved concurrency that results, with significantly reduced wait times at the buffer acquire phase. Though skew in the record size distribution could limit scalability because of the requirement to release buffers in order, we find that this is not a problem in practice because realistic log record sizes do not vary enough to justify the additional complexity. We consider this matter further in Section A.3 of the appendix and propose a solution which provides robustness in the face of skewed log record sizes with a 10% performance penalty.

5.3 Putting it all Together: Hybrid Log Buffer

In the previous two sections we outlined (a) a consolidation array based approach to reduce the number of threads entering the critical section, and (b) a decoupled buffer fill which allows threads to pipeline buffer fills outside the critical section. Neither approach eliminates all contention by itself. The two are orthogonal, however, and can be combined easily. Consolidating groups of threads limits log contention to a constant that does not depend on the number threads in the system, while providing a degree of buffer insert pipelining (within groups but not between them). Decoupling buffer fill operations allows pipelining between groups and reduces the log critical section length by moving buffer outside, thus making performance relatively insensitive to log record sizes. The resulting design, shown in Figure 6(CD), achieves bounded contention for threads in the buffer acquire stage and maximum pipelining of all operations.

6. PERFORMANCE EVALUATION

We implement the techniques described in sections 3, 4, and 5 into a logging subsystem called Aether. To enhance readability, most of the performance evaluation of ELR and flush pipelining is shown in sections 3 and 4, respectively. Unless otherwise stated in this section we assume those optimizations are already in place. This section details the sensitivity of the consolidation array based techniques to various parameters, and finally evaluates performance of Aether in a prototype database system.

6.1 Experimental Setup

All experiments were performed on a Sun T5220 (Niagara II) server with 64GB of main memory running Solaris 10. The Niagara II chip contains sixteen processing pipelines, each capable of supporting four hardware contexts, for a total of 64 OS-visible "CPUs." The high degree of hardware parallelism makes it a good indicator of the challenges all platforms will face as on-chip core counts continue to double. We use Shore-MT [11], an open-source multi-threaded storage manager. We developed Shore-MT using as basis the SHORE storage manager [2], to achieve scalability on multicore platforms. To eliminate contention in the lock manager and focus on logging, we use a version of Shore-MT with Speculative Lock Inheritance [10]. We run the following benchmarks:

TATP. TATP (aka TM1) [23] models a cell phone provider database. It consists of seven very small transactions, both update and read-only. The application exhibits little logical contention, but the small transaction sizes stress database services, especially logging and locking. We use a database of 100K Subscribers.

TPC-B. This benchmark [24] models a banking workload and it is intended as a database stress test. It consists of a single small update transaction and exhibits moderate lock contention. Our experiments utilize a 100-teller dataset.

Log insert microbenchmark. We extract a subset of Shore-MT's log manager as an executable which supports only log insertions without flushes to disk or performing other work, thereby isolating the log buffer performance. We then vary the number of threads, the log record size and distribution, and the timing of inserts.

For each component of Aether, we first quantify existing bottlenecks, then implement our solution in Shore-MT and evaluate the resulting impact on performance. Because our focus is on the logging subsystem, and because modern transaction processing workloads are largely memory resident [22], we use memory-resident datasets, while disk still provides durability.

All results report the average of 10 30-second runs unless stated otherwise; we do not report variance because all measurements were within 2% of the mean. Measurements come from timers in the benchmark driver as well as Sun's profiling tools. Profiling is highly effective at identifying software bottlenecks even in the early stages before they begin to impact performance. The hardware limits scalability artificially by multiplexing many hardware contexts over each processor pipeline; we verify that this is the case by running independent copies of Shore-MT in parallel (where the effect remains in spite of a total lack of software contention), and on multi-socket machines (where the effect is shifted to the right by a factor proportional to the number of sockets).

6.2 Log Buffer Contention

First, to set the stage, we measure the contention on the log buffer once the Early Lock Release and the flush pipelining have been applied. Figure 7 shows the time breakdown for Shore-MT with

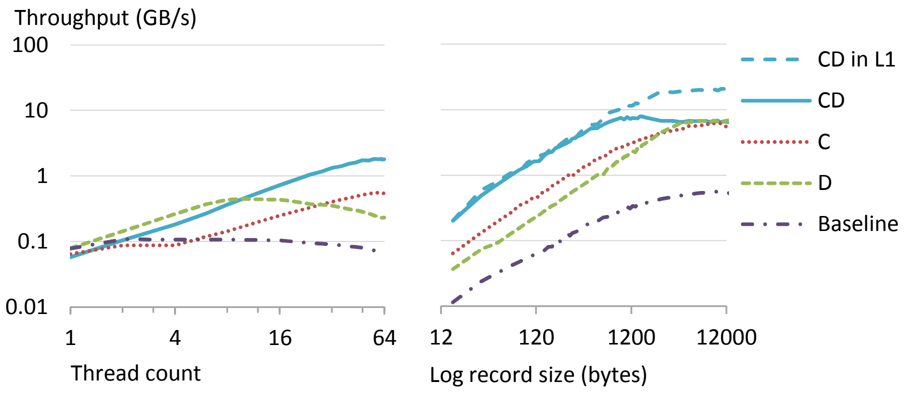

Figure 8. Log buffer scalability with respect to thread counts (left, 120B log records) and log record size (right, 64 threads)

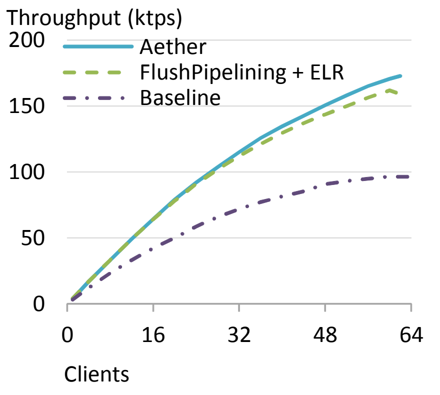

Figure 9. Overall performance improvement provided by each component of Aether

ELR and flush pipelining active using its baseline log buffer implementation as an increasing number of clients submit the UpdateLocation transaction from TATP. As the load in the system increases, the time each transaction spends contenting for the log buffer increases, at a point which the log buffer contention becomes the bottleneck taking more than 35% of the execution time. This problem will only grow as processor vendors release more parallel multi-core hardware.

6.3 Impact of log buffer optimizations (microbenchmarks)

A database log manager should be able to sustain any number of threads regardless of the size of the log records they insert, limited only by memory and compute bandwidth. Next, through a series of microbenchmarks we determine how well the log buffer designs proposed in Section 5 meet these goals. In each experiment we compare the baseline implementation with the consolidation array (C), decoupled buffer insert (D), and the hybrid solution combining the two optimizations (CD). We examine scalability with respect to both thread counts and log record sizes and we analyze how the consolidation array's size impacts its performance. Further experiments in Sections A.3 and A.4 (appendix) explore the impact of skew in the record size distribution and of changing the number of slots in the slot array.

6.3.1 Scalability With Respect to Thread Count

The most important metric of a log buffer is how many insertions it can sustain per unit time, or the bandwidth which the log can sustain at a given average log insert size. It is important because core counts grow exponentially while log record sizes are application- and DBMS-dependent and are fixed. The average record size in our workloads is about 120 bytes and a high-performance application generates between 100 and 200MBps of log, or between 800K and 1.6M log insertions per second.

Figure 8(left) shows the performance of the log insertion microbenchmark for records of an average size of 120B as the number of threads varies along the x-axis. Each data series shows one of the log variants. We can see that the baseline implementation quickly becomes saturated, peaking at roughly 140MB/s and falling slowly as contention increases further. Due to its complexity, the consolidation array starts out with lower throughput than the baseline. But once contention increases, the threads combine their requests and performance scales linearly. In contrast, decoupled insertions avoid the initial performance penalty and perform better, but eventually the growing contention degrades performance and perform worst than the consolidation array.

Finally, the hybrid approach combines the best properties of both optimizations, eliminating most of the startup cost from (C) while limiting the contention which (D) suffers. The drop in scalability near the end is a hardware limitation, as described in Section 6.1. Overall, we see that while both consolidation and decoupling are effective at reducing contention, both have limitations which we overcome by combining the two, achieving near-linear scalability.

6.3.2 Scalability With Respect to Log Record Size

In addition to thread counts, log record sizes also have a strong influence on the performance of the log buffer. In the case of the baseline and consolidated variants, larger record sizes increase the critical section length; in all cases, however, larger record sizes decrease the number of log inserts one thread can perform because it must copy an increasing amount of data per insertion.

Figure 8(right) shows the impact of these two factors, plotting sustained bandwidth achieved by 64 threads as they insert log records ranging between 48B and 12kB (the largest record size in Shore-MT). As log records grow the baseline performs better, but there is always enough contention that makes all other approaches more attractive. The consolidated variant (C) performs better at small records sizes as it can handle contention much better than the decoupled record insert (D). But once the records size is over 1kB contention becomes low and the decoupled insert variant fares better as more log inserts can be pipelined at the same time. The hybrid variant again significantly outperforms its base components across the whole range, but in the end all three become bandwidth-limited as they saturate the machine's memory system.

Finally, we modify the microbenchmark so that threads insert their log records repeatedly into the same thread-local storage, which is L1 cache resident. With the memory bandwidth limitation removed, the hybrid variant continues to scale linearly with record sizes until it becomes CPU-limited at roughly 21GBps (nearly 20x higher throughput than today's systems can reach).

6.4 Overall Impact of Aether

To complete the experimental analysis, we successively add each of the components of Aether to the baseline log system and measure the impact. With all components active we avoid the bottlenecks summarized in Figure 1 and can identify optimizations which are likely to have highest impact now and in the future.

Figure 9 captures the scalability of Shore-MT running the UpdateLocation transaction from TATP. We plot throughput as the number of client threads varies along the x-axis. For systems today, flush pipelining provides the largest single performance boost, 68% higher than the baseline. The scalable log buffer adds a modest 7% further speedup by eliminating log contention.

Based on these results we conclude that the most pressing bottleneck is scheduler overload induced by high transaction throughput and the associated context switching. However, flush pipelining depends on ELR to prevent log-induced lock contention which would otherwise limit scalability.

As core counts continue to increase, we also predict that in the future log buffer contention will become a serious bottleneck unless techniques such as the hybrid implementation presented in Section 5 are used. Even today, contention at the log buffer impacts scalability to a small degree. In addition, the profiling results from Figure 7 indicate that this bottleneck is growing rapidly with core counts and will soon dominate. This indication is further strengthened by the fact that Shore-MT running on today's hardware achieves almost exactly the peak log throughput we measure in the microbenchmark for the baseline log. In other words, even a slight increase in throughput (with corresponding log insertions) will likely push the log bottleneck to the forefront. Fortunately, the hybrid log buffer displays no such lurking bottleneck and our microbenchmarks suggest that it has significant headroom to accept additional log traffic as systems scale in the future.

7. CONCLUSIONS

Log manager performance becomes increasingly important as database engines continue to increase performance by exploiting hardware parallelism. However, the serial nature of the log, as well as long I/O times, threatens to turn the log into a growing bottleneck. As available hardware parallelism grows exponentially, contention for the central log buffer threatens to halt scalability. A new algorithm, consolidation array-based backoff, incorporates concepts from distributed systems to convert the previously serial log insert operation into a parallel one which scales well even under much higher contention than current systems can generate. We address more immediate concerns of excessive log-induced context switching using a combination of early lock release and log flush pipelining which allow transactions to commit without triggering scheduler activity, and without sacrificing safety or durability. Taken together, these techniques allow the database log manager to stay off the critical path of the system for maximum performance even as available parallelism continues to increase.

ACKNOWLEDGMENTS

This work was partially supported by a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

REFERENCES

[1] L. Bouganim, B. Jonsson, and P. Bonnet. "uFlip: Understanding Flash I/O Patterns." In *Proc. CIDR, 2009.*

[2] M. Carey, et al. "Shoring up persistent applications." In *Proc. SIGMOD, 1994.*

[3] S. Chen. "FlashLogging: exploiting flash devices for synchronous logging performance." In *Proc. SIGMOD, 2009.*

[4] D. DeWitt, et al. "Implementation Techniques for Main Memory Database Systems." *ACM TODS, 14(2), 1984.*

[5] D. Gawlick and D. Kinkade. "Varieties of Concurrency Control in IMS/VS Fast Path." *IEEE Database Eng. Bull. 1985*.

[6] N. Hardavellas, et al. "Database servers on chip multiprocessors: limitations and opportunities." In *Proc. CIDR, 2007.*

[7] S. Harizopoulos, D. J. Abadi,. S. Madden, and M. Stonebraker. "OLTP through the looking glass, and what we found there." In *Proc. SIGMOD, 2008.*

[8] P. Helland, H. Sammer, J. Lyon, R. Carr, and P. Garrett. "Group Commit Timers and High-Volume Transaction Systems." In *Proc. HPTS, 1987.*

[9] D. Hendler, N. Shavit, and L. Yerushalmi. "A Scalable Lock-free Stack Algorithm." In *Proc. SPAA, 2004.*

[10] R. Johnson, I. Pandis, and A. Ailamaki. "Improving OLTP Scalability using Speculative Lock Inheritance." In *Proc. VLDB*, 2009.

[11] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. "Shore-MT: a scalable storage manager for the multicore era." *In Proc. EDBT, 2009.*

[12] S.-W. Lee, B. Moon, J.-M. Kim, and S.-W. Kim. "A Case for Flash Memory SSD in Enterprise Database Applications." In *Proc. SIGMOD, 2008.*

[13] C. Mohan. "ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes." In *Proc. VLDB, 1990.*

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." *ACM TODS, 17(1), 1992.*

[15] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. "Using Elimination to Implement Scalable FIFO Queues." In *Proc. SPAA, 2005*.

[16] Oracle. *Oracle Asynchronous Commit*. Oracle Database Advanced Application Developer's Guide. Available at: http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14251/adfns_sqlproc.htm.

[17] PostgreSQL Asynchronous Commit. PostgreSQL 8.4.2 Documentation. Available at: http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4.2-A4.pdf.

[18] A. Rafii, and D. DuBois. "Performance Tradeoffs of Group Commit Logging." In *Proc. CMG Conference, 1989.*

[19] N. Shavit and D. Touitou. "Elimination Trees and the Construction of Pools and Stacks." In *Theory of Computing Systems*, 30(6), pp 645-670, 1997.

[20] M. L. Scott. "Non-Blocking Timeout in Scalable Queue-Based Spin Locks." In *Proc. PODC*, 2002.

[21] E. Soisalon-Soininen, and T. Ylonen. "Partial Strictness in Two-Phase Locking." In *Proc. ICDT, 1995.*

[22] M. Stonebraker, et al. "The end of an Architectural Era (It's Time for a Complete Rewrite)." In *Proc. VLDB, 2007.*

[23] Telecom Application Transaction Processing Benchmark (TATP). *TATP Benchmark Description.* Available at: http://tatpbenchmark.sourceforge.net/TATP_Description.pdf.

[24] Transaction Processing Performance Council (TPC). *TPC Benchmark B: Standard Specification*. Available at http://www.tpc.org/tpcb/spec/tpcb_current.pdf.

A APPENDIX

This appendix consists of four subsections. First, we present in detail the log buffer designs, presented in Section 5, using code sketches for the various algorithms (Section A.1). Second, we describe in detail the consolidation array used by Algorithm 2 (Section A.2). Third, we discuss a further modification of the log buffer design to address a potential source of delays coming from the requirement that all threads need to release their buffer in-order (Section A.3). Fourth, we discuss about distributing the log and why it is very difficult to have an efficient and scalable distributed log implementation (Section A.5).

A.1 Details of the Log Buffer Algorithms

In this subsection we explain the implementation of the algorithms, presented in Section 5, with pseudocode sketches.

Algorithm 1 – Baseline log insertion algorithm

```
1   log_insert(size, data):
2       lock_acquire(L)
3       lsn = buffer_acquire(size)
4       buffer_fill(lsn, size, data)
5       buffer_release(lsn, size)
6   end
7
8   buffer_acquire(size):
9       /* ensure buffer space available */
10      lsn = /* update lsn and buffer state */
11      return lsn
12  end
13
14  buffer_fill(lsn, size, data):
15      /* set record's LSN */
16      /* copy data to buffer (may wrap) */
17  end
18
19  buffer_release(lsn, size):
20      /* release buffer up to lsn+size */
21      lock_release(L)
22  end
```

Algorithm 2 – Log insertion with consolidated buffer acquire

```
1   log_insert(size, data):
2       if (lock_attempt(L)== SUCCESS)
3           lsn = buffer_acquire(size)
4           buffer_fill(lsn, size, data)
5           buffer_release(lsn, size)
6           return      /* no contention */
7       end
8       {s, offset} = slot_join(size)
9       if (0 == offset)      /* slot owner */
10          lock_acquire(L)
11          group_size = slot_close(s)
12          replace_slot(s)
13          lsn = buffer_acquire(group_size)
14          slot_notify(s, lsn, group_size)
15      else                  /* wait for owner */
16          {lsn, group_size} = slot_wait(s)
17      end
18      buffer_fill(lsn+offset, size, data)
19      if (slot_release(s) == SLOT_DONE)
20          buffer_release(lsn, group_size)
21      end
22  end
```

Algorithm 3 – Log insertion with decoupled buffer fill

```
1   buffer_acquire(size, data):
2       /* wait for buffer space */
3       lsn = /* update lsn and buffer state */
4       lock_release(L)
5       return lsn
6   end
7
8   buffer_release(lsn, size):
9       while (lsn != next_release_lsn)
10          /* wait my turn */
11      end
12      /* release buffer up to lsn+size */
13      next_release_lsn = lsn+size
14  end
```

Baseline. In a straightforward implementation, a single mutex protects the log's buffer, LSN generator, and other structures. Algorithm 1 presents such an approach, which the later designs build on. In the baseline case a log insert always begins with acquiring the global mutex (L2) and finishes with its release (L21). Inside the critical section there are three operations: (i) A thread first allocates log buffer space (L8-12); (ii) It then performs the record insert (L14-17); (iii) Finally, it releases the buffer space making the record insert visible to the flush daemon by incrementing a dedicated pointer (L20). As discussed, the baseline algorithm suffers two weaknesses. First, contention is proportional to the number of threads in the system; second, the critical section length is proportional to the amount of work performed by each thread.

Consolidation array. Consolidation-based backoff aims to reduce contention and, more importantly, make it independent of the number of threads accessing the log. A sketch of the code is presented in Algorithm 2. The primary data structure consists of an array with a fixed number of slots where threads can aggregate their requests. Rather than acquiring the lock unconditionally, threads begin with a non-blocking lock attempt. If the attempt succeeds, they perform the log insert directly, as before (L2-6). Threads which encounter contention back off to the consolidation array and attempt to join one of its slots at random (L8). The first thread to claim a slot becomes the slot's owner and is responsible to acquire buffer space on behalf of the group which forms while it waits for the mutex. Once inside the critical section, the group leader reads the current group size and marks the group as closed using an atomic swap instruction (L11); once a slot closes threads can no longer join the group. The group leader then acquires buffer space and notifies the other group members before beginning its own buffer fill (L13-14). Meanwhile, threads which join the group infer their relative position in the meta-request from the group size; once the group leader reports the LSN and buffer location each thread can compute the exact LSN and buffer location which belongs to it (L16 and L18). As each thread leaves (leader included), it decrements the slot's reference count and the last thread to leave releases the buffer (L19-20).

Once a consolidation array slot closes, it remains inaccessible while the threads in the group perform the consolidated log insert, with time proportional to the log record insert size plus the overhead of releasing the buffer space. To prevent newly-arrived threads from finding all slots closed and being forced to wait, each slot owner removes the consolidation structure from the consolidation array when it closes, replacing it with a fresh slot that can accommodate new threads (L12). The effect is that the array slot reopens even though the threads that consolidated their request are still working on the previous (now-private) version of that slot. We avoid memory management overheads by allocating a large number of consolidation structures at startup, which we treat as a circular buffer when allocating new slots. At any given moment of time arriving threads access only the combination structures present in the slots of the consolidation array, and those slots are returned to the free pool after the buffer release stage. In the common case the next slot to be allocated was freed long ago and each "allocation" requires only an index increment. Section A.3 describes the implementation details of the consolidation array.

Decoupled buffer fill. Decoupling the log insert from holding the mutex reduces the critical section length and in addition contention cannot increase with the size of the log record size. Algorithm 3 shows the changes over the baseline implementation (Algorithm 1) needed to decouple buffer fills from the serial LSN generation phase. First, a thread acquires the log mutex, generates the LSN, and allocates buffer space. Then, it releases the central mutex

Algorithm 4 – Log insertion with delegated buffer release

```
1    buffer_acquire(size, data):
2        /* wait for buffer space */
3        lsn = /* update lsn and buffer state */
4        qnode = queue_join(Q, lsn, size)
5        lock_release(L)
6        return qnode
7    end
8
9    buffer_release(qnode):
10       if (queue_delegate(Q, qnode) == DELEGATED)
11           return /* someone else will release*/
12       end
13
14     do_release:
15       /* release qnode’s buffer region */
16       next = queue_handoff(Q, qnode)
17       if (next && is_delegated(next))
18           qnode = next
19           goto do_release
20       end
21   end
```

immediately (L4) and performs its buffer fill concurrently with other threads. Once the buffer fill is completed, the thread waits for all other threads before it to finish their inserts (L9) and the last to finish releases the log buffer space (L13). The release stage uses the implicit queuing of the release_lsn to avoid expensive atomic operations or mutex acquisitions.

A.2 Consolidation-based Backoff

The consolidated log buffer acquire described in Algorithm 2 uses a new algorithm, the consolidation array to divert contention away from the log buffer. We base our design on the elimination-based backoff algorithm [9], extending it to allow the extra cooperation needed to free the buffer after threads consolidate their requests.

Elimination backoff turns “opposing” operations (e.g. stack push and pop) into a particularly effective form of backoff: threads which encounter contention at the main data structure probe randomly an array of N “slots” while they wait. Threads which arrive at a slot together serve each others’ requests and thereby cancel each other out. When such eliminations occur, the participating threads return to their caller without ever entering the main data structure, slashing contention. With an appropriately-sized elimination array, an unbounded number of threads can use the shared data structure without causing undue contention.

Consolidation backoff operates on a similar principle to elimination, but with the complication that log inserts do not cancel each other out entirely: At least one thread from each group (the “leader”) must still acquire space from the log buffer on behalf of the group. In this sense consolidation is more similar to a shared counter than a stack, but with the further requirement that the last thread of each group to complete its buffer fill operation must release the group’s buffer back to the log. These additional communication points require two major differences between the consolidation array and an elimination array. First, the slot protocol which threads use to combine requests is significantly more complex. Second, slots spend a significant fraction of their lifecycle unavailable for consolidation and it becomes important to replace busy slots with fresh ones for consolidation to remain effective under load. Algorithm 5 gives pseudocode for the consolidation array implementation, which the following paragraphs describe in further detail, while Figure 10 supplements the pseudocode with a summary of a slot’s life cycle and state space.

Slot join operation (lines 1-19). The consolidation array consists of ARRAY_SIZE pointers to active slots. Threads which enter the slot array start probing for slots in the OPEN state, starting at a random location. Probing repeats as necessary, but should be relatively rare because slots are swapped out of the array immediately whenever they become PENDING. Threads attempt to claim OPEN slots using atomic compare-and-swap to increment the state by the insert size. In the common case the CAS fails only if another thread also incremented the slot’s size. However, the slot may also close, forcing the thread to start probing again. Eventually the thread succeeds in joining a slot and returns a (slot, offset) pair. The offset serves two purposes: the thread at position zero becomes the “group leader” and must acquire space in the log buffer on behalf of the group, and follower threads use their offset to partition the resulting allocation with no further communication.

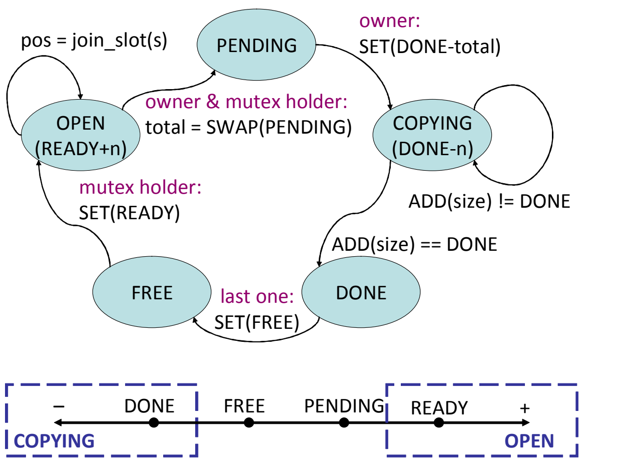

Figure 10. Life cycle and state space of a c-array slot, to accompany Algorithm 5.The OPEN (COPYING) state covers all values at least as large (small) as READY (DONE).

Slot close operation (lines 21-33). After the group leader acquires the log buffer mutex, it closes the group in order to determine the amount of log space to request (and to prevent new threads from arriving after allocation has occurred). It does so using an atomic swap, which returns the current state and assigns a state of PENDING. The state change forces all further slot_join operations to fail (line 7), but most threads will never see this because the calling thread first swaps a fresh slot into the array. To do so, it probes through the pool of available slots, searching for a FREE one. The pool is sized large enough to ensure the first probe nearly always succeeds. The pool allocation need not be atomic because the caller already holds the log mutex. Once the slot is closed the function returns the group size to the caller so it can request the appropriate quantity of log buffer space.

Slot notify and wait operations (lines 35-46). After the slot owner acquires buffer space, it stores the base LSN and buffer address into the slot, then sets the slot’s state to DONE-group_size as a signal to the rest of the group. Meanwhile, waiting threads spin until the state changes, then retrieve the starting LSN and size of the group (the latter is necessary because any thread could be the one to release the group’s buffer space).

Slot release and free operations (lines 48-55). As each thread completes its buffer insert, it decrements the slot’s count by its contribution. The last thread to release will detect that the slot became DONE must free the slot; all others may leave immediately. The slot does not immediately become free, however, because the calling thread may still use it. This is particularly important for the delegated buffer release optimization described in Section A.3, because the to-be-freed slot becomes part of a queue to be processed by some other thread. Once the slot is truly

Algorithm 5 – Consolidation array implementation

```
slot_join(size):
  probe_slot:
    idx = randn(ARRAY_SIZE)
    s = slot_array[idx];
    old_state = s->state
  join_slot:
    if(old_state < SLOT_READY)
        /* new threads not welcome */
        goto probe_slot;
    end
    new_state = old_state + size
    cur_state = cas_state(s, old_state, new_state)
    if(cur_state != old_state)
        old_state = cur_state
        goto join_slot
    end
    /* return my position within the group */
    return {s, old_state-SLOT_READY}
end

slot_close(s):
  retry:
    s2 = slot_pool[pool_idx % POOL_SiZE];
    pool_idx = pool_idx+1
    if(s2->state != SLOT_FREE)
        goto retry;
    end
    /* new arrivals will no longer see s */
    s2->state = SLOT_OPEN
    slot_array[s->idx] = s2
    old_state = swap_state(s, SLOT_PENDING)
    return old_state-SLOT_READY
end

slot_notify(s, lsn, group_size):
    s->lsn = lsn
    s->group_size = group_size
    set_state(s, SLOT_DONE-group_size)
end

slot_wait(s):
    while(info->state > SLOT_DONE)
        /* wait for notify */
    end
    return {s->lsn, s->group_size}
end

slot_release(s, size):
    new_state = state_atomic_add(s, size)
    return new_state
end

slot_free(s):
    set_state(s, SLOT_FREE)
end
```

finished, the owning thread sets its state to FREE; the operation need not be atomic because other threads ignore closed slots.

In conclusion, the consolidation array provides a way for threads to communicate in a much more distributed fashion than the original (serial) log buffer operation which it protects. The overhead is small, in the common case two or three atomic operations per participating thread, and occurs entirely off the critical path (other threads continue to access the log unimpeded).

A.3 Delegated Log Buffer Release and Skew

The requirement that all threads release their buffers in order remains a potential source of delays. Many smaller insertions might execute entirely in the shadow of a large one but must still wait for the large insert to complete before releasing their buffer space. Buffer and log file wraparounds complicate matters further, because they prevent threads from consolidating buffer releases. Such wrapping buffer releases must be identified and processed separately from normal ones because they impose extra work at log flush time, such as closing and opening log files.

We remove this extra dependency between transactions by turning the implied LSN queue into a physical data structure, as shown in Algorithm 4. Before releasing the mutex, during the buffer acquire, each thread joins a release queue (L4), storing in the queue node all information needed to release its buffer region. The decoupled buffer fill proceeds as before. At buffer release time, the thread first attempts to abandon its queue node, delegating the corresponding buffer release to a (presumably slow) predecessor which has not yet completed its own buffer fill. The delegation protocol is lock-free and non-blocking, and is based on the abortable MCS queue lock by Scott [20] and the critical section-combining approach suggested by Oyama et al. [A5]

To summarize the protocol, a thread with at least one predecessor attempts to change its queue node status atomically from *waiting* to *delegated* (L10, corresponding to the *aborted* state in Scott's work). On success, a predecessor will be responsible for the buffer release and the thread returns immediately (L11). Otherwise, or if no predecessor exists, the thread releases its own buffer region and attempts to leave before its successor can delegate more work (L16). A successful CAS from *waiting* to *released* prevents the successor from abandoning its node; on failure, the thread continues to release delegated nodes until it reaches the end of the queue or successfully hands off (L17-20). Threads randomly choose not to abandon their nodes with probability 1/32 to prevent a "treadmill" effect where one thread becomes stuck performing endless delegated buffer releases. This breaks long delegation chains (which are relatively rare) without impeding pipelining in the common case. As with Oyama's proposal [A5], the grouping actually improves performance because a single thread does all the work without incurring coherence misses.

In Figure 11 we test the stability of the new algorithm (named CDME) and compare it with the hybrid variant from Section 5.3 (CD). We use the same microbenchmark setup from Section 6 but modify it to present the worst-case scenario for the CD algorithm: a strongly bi-modal distribution of log record sizes. We fix one peak at 48 bytes (the smallest log record in Shore-MT) and we vary the second peak (called the outlier). For every 60 small records a large record is inserted in the log. CD performs poorly with such a workload because the rare, large, record can block many smaller ones and disrupt the pipelining effect. We present along the y-axis the throughput as we increase the outlier record size along the x-axis. CD and CDME perform similarly until an outlier size of around 8kiB, when CD stops scaling and its performance levels off. CDME, which is immune to record size variability, achieves up to double the performance of the CD for outlier records larger than 65kiB.

The CDME algorithm is more robust than the CD variant but, for the database workloads we examined, it is unnecessary in practice because nearly all records are small and the frequency of larger outliers is orders magnitude smaller than examined here. For example, in Shore-MT the largest log record is 12kiB with a frequency of 0.01% of the total log inserts. In addition, CDME achieves around 10% lower throughput than the CD variant under normal circumstance, making it unattractive. Nevertheless, for other configurations which encounter significant skew, the CDME algorithm might be attractive given its stability guarantee.

A.4 Sensitivity to consolidation array size

Our last microbenchmark analyzes whether (and by how much) the consolidation array's performance is affected by the number of available slots. Ideally the performance should depend only on the hardware and be stable as thread counts vary. Figure 12 shows a contour map of the space of slot sizes and thread counts, where the height of each data point is its sustained bandwidth. Lighter colors indicate higher bandwidth, with contour lines marking specific throughput levels. We achieve peak performance with 3-4 slots, with lower thread counts peaking with fewer and high thread

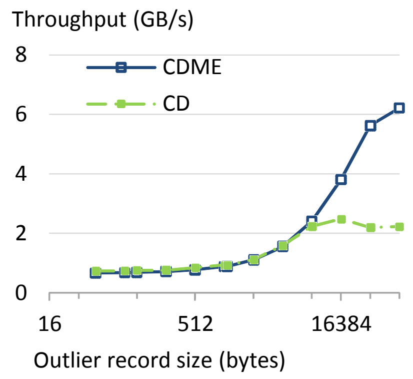

Figure 11. Performance impact of log record size skew.

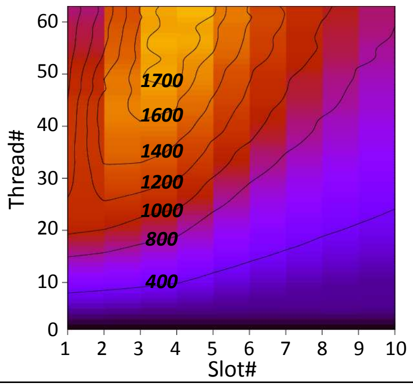

Figure 12. Sensitivity to the number of slots in the consolidation array

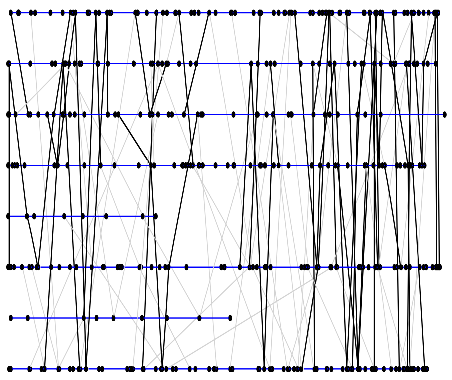

Figure 13. Inter-log dependencies for 1ms of TPC-C (8 logs, ~100kB, ~30 commits).

counts requiring a somewhat larger array. The optimal slot number corresponds closely with the number of threads required to saturate the baseline log which the consolidation array protects. Based on these results we fix the consolidation array size at four slots to favor high thread counts; at low thread counts the log is not on the critical path of the system and its peak performance therefore matters much less than at high thread counts.

A.5 A Case Against Distributed Logging

This subsection presents qualitative and quantitative analysis showing that our improved log buffer design is likely to outperform distributed logging as a contention-reducing approach, both from a performance and implementation perspective.

A distributed log has the potential to ease bottlenecks by spreading load over N logs instead of just one. ARIES-style recovery only requires a partial order between the transactions accessing the same data. Intuitively, it should be possible to parallelize the log, given that most transactions execute in parallel without conflicts. However, a distributed log must track transaction dependencies and make sure that logs become durable in a coherent order, as discussed by DeWitt et al. [4].

Write-ahead logging allows transactions to release page latches immediately after use, minimizing data contention and allowing database pages to accumulate many changes in the buffer pool before being written back to disk. Further, serial logging allows transactions to not track physical dependencies, especially those that arise with physiological logging,[4] as a transaction's commit never reaches disk before its dependencies. A distributed implementation must instead track or eliminate these physical dependencies without requiring multiple log flushes per transaction. Otherwise, the serial implementation will actually be faster.

Unfortunately, this challenge is difficult to address efficiently because physical dependencies can be very tight, especially due to hot database pages. For example, Figure 13 shows the dependencies which would arise in an 8-way distributed log for a system running the TPC-C benchmark [A6]. Each node in the graph represents a log record, with horizontal edges connecting records from the same log. Diagonal edges mark physical dependencies which arise when a page moves between logs. Dark edges mark tight dependencies where the older record is one of the five most recently inserted records for its log at the time. The entire graph covers roughly 100kB of log records, which corresponds to less than 1ms wall time and dozens of transaction commits.

Because dependencies are so widespread and frequent, it is almost infeasible to track them, and even if tracked efficiently the dependencies would still require most transactions to flush multiple logs at commit time. In Figure 13 there is no obvious way of assigning log records to different partitions so that the dependency lines between partitions would be significantly reduced. The authors are unaware of any DBMS which distributes the log within a single node, and even distributed DBMS often opt for a shared log (including Rdb/VMS [A4]). Distributed DBMS which utilize distributed logging either force transactions to span multiple nodes (with well-known consequences for performance and scalability [A1]) or else migrate dirty pages between nodes through a shared storage or network interconnect rather than accepting the high overhead of having a distributed transaction that needs to flush multiple logs in a specific sequence [A2].

Using physical-only logging and having an almost-perfectly partitionable workload makes the implementation of a distributed log feasible [A3]. However, if physiological logging is used and as Figure 13 shows, distributed logs are both highly complex and potentially very slow under many workloads. We conclude that adding a distributed log manager within a database instance is neither attractive nor feasible for reducing log buffer contention.

4. For example, if transaction A inserts a record in slot 13 of a page, and then B inserts a record in slot 14, A's log record must become durable first or recovery could encounter an inconsistent page and fail.

APPENDIX REFERENCES

[A1] P. Helland. "Life Beyond Distributed Transactions: an Apostate's Opinion." In Proc. *CIDR, 2007.*

[A2] T. Lahiri, V. Srihari, W. Chan, and N. MacNaughton. "Cache Fusion: Extending shared-disk clusters with shared caches." In Proc. *VLDB, 2001.*

[A3] D. Lomet. "Recovery for Shared Disk Systems Using Multiple Redo Logs." *CRL 90/4, 1990.*

[A4] D. Lomet, R. Anderson, T. K. Rengarajan, and P. Spiro. "How the Rdb/VMS Data Sharing System Became Fast." *CRL 92/4, 1992.*

[A5] Y. Oyama, K. Taura, and A. Yonezawa. "Executing Parallel Programs with Synchronization Bottlenecks Efficiently." In Proc. *PDSIA, 1999*, pp. 182--204.

[A6] Transaction Processing Performance Council. "TPC - C v5.5: On-Line Transaction Processing (OLTP) Benchmark."