The VLDB Journal
DOI 10.1007/s00778-011-0260-8

SPECIAL ISSUE PAPER

Scalability of write-ahead logging on multicore and multisocket hardware

Ryan Johnson · Ippokratis Pandis · Radu Stoica · Manos Athanassoulis · Anastasia Ailamaki

Received: 23 February 2011 / Revised: 7 August 2011 / Accepted: 25 October 2011

Abstract The shift to multi-core and multi-socket 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, in a single-socket multiprocessor server. Moreover, it achieves log insert throughput over 2.2 GB/s for small log records on the single-socket server, roughly 20 times higher than the traditional way of accessing the log using a single mutex. Furthermore, we investigate techniques on scaling the performance of logging to multi-socket servers. We present a set of optimizations which partly ameliorate the latency penalty that comes with multi-socket hardware, and then we investigate the feasibility of applying a distributed log buffer design at the socket level.

Keywords Log manager · Early lock release · Flush pipelining · Log buffer contention · Consolidation array · Scaling to multisockets

R. Johnson (✉)
Department of Computer Science, University of Toronto, Toronto, ON, Canada
e-mail: ryan.johnson@cs.utoronto.ca

I. Pandis
IBM Almaden Research Center, San Jose, CA, USA
e-mail: ipandis@us.ibm.com

R. Stoica · M. Athanassoulis · A. Ailamaki
School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Vaud, Switzerland
e-mail: radu.stoica@epfl.ch

M. Athanassoulis
e-mail: manos.athanassoulis@epfl.ch

A. Ailamaki
e-mail: anastasia.ailamaki@epfl.ch

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 [10]. 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—in order to benefit from new hardware. Although database workloads

Published online: 03 December 2011

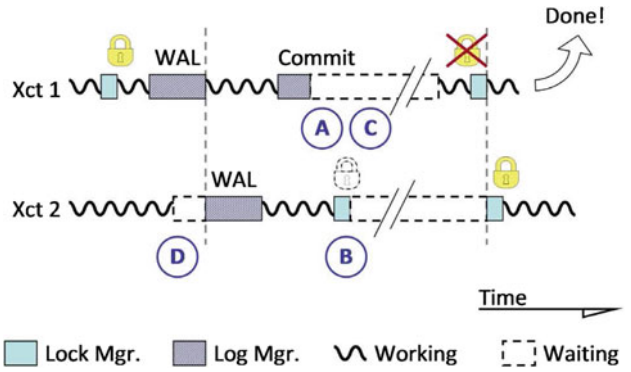

Fig. 1 A time line of two transactions illustrating four kinds of log-imposed delay: I/O-related delays (**a**), increased lock contention (**b**), scheduler overload (**c**), and log buffer contention (**d**)

exhibit high concurrency, internal bottlenecks often mean that database engines cannot extract enough parallelism to meet multicore hardware demands [15].

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 100 ktps 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) [23] to protect against data corruption and lost committed work after crashes. WAL allows transactions to execute and commit without requiring that all the data pages they update to be written to persistent storage first. However, as Fig. 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 become less important as fast solid-state drives gain popularity [1,19], and when using techniques such as group commit [12].

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 may significantly increase lock contention in the system and create an artificial bottleneck in many workloads. For example, the left-most bar in Fig. 2 shows CPU utilization as 60 clients run the TPC-B [37] benchmark in a modern storage manager [15] on a Sun Niagara II chip with 64 hardware contexts (see Sect. 6 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 ram-disk 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 if newly runnable threads accumulate faster than the OS can dispatch them. The second bar in Fig. 2 shows for the same workload the processing time of a system which suffers from the problem of OS scheduler overload. The system remains 40% idle even though transactions are ready and waiting to run. We analyze excessive context switching problem in Sect. 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. The third bar in Fig. 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

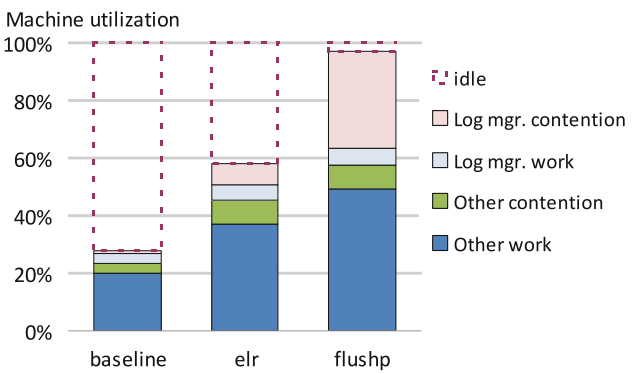

Fig. 2 Breakdown of CPU time showing work and contention due to the log versus other parts of the system, when 60 clients run the TPC-B benchmark, as we remove log-related bottlenecks. Ideally, useful work would make up close to 100% of the total

[1] See, e.g., top published TPC-C results or performance figures reported by main-memory databases like H-Store [35].

simultaneous log inserts, the contention for the centralized log buffer becomes a significant (and growing) fraction of total execution 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 indicator of the continuing log bottleneck. Available in most DBMSs (including Oracle [26] and PostgreSQL [30]) 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 extends *Aether* [16] a complete approach toward 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 its contributions below.

First, we evaluate *Early Lock Release (ELR)*, a technique for eliminating log-induced lock contention. ELR has been proposed in the past but, to our knowledge, has never been evaluated in the literature and is not used today by any mainstream database engine. We show that, particularly for skewed accesses common to real workloads, ELR increases throughput by 15–164% even when log disk latency is minimal.

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 as asynchronous commit without sacrificing durability.

Third, 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.

1.3 Scaling to multisocket systems

Even though the design, as proposed, achieves excellent performance for single-socket multicore systems, multisocket systems pose additional challenges. In particular, in multisocket multicores the communication cost across cores varies from very cheap (few tenths of cycles, if communication takes place between cores within the same chip) to very expensive (up to thousands of cycles, if communication takes place between cores of different chips in a big machine). The non-uniformity in communication complicates further the design of an efficient logging subsystem. We therefore extend our previous work to consider the effects of non-uniform access times imposed by multi-socket hardware.

In Sect. 7 we study the additional performance problems on multisocket systems. First, we propose optimizations which make Aether perform better in such hardware. However, this is not enough to overcome the high communication costs between sockets. We therefore argue that if the system is designed with a single centralized log buffer, a performance hit is unavoidable. We then explore the feasibility of a distributed log buffer design. In microbenchmarks the final design, *AetherSMP*, achieves near-linear performance on multisocket machines, and in macrobenchmarks it eliminates the log bottleneck from database workloads, leaving other bottlenecks as the primary barrier to scalability.

The rest of the document is organized as follows. Section 2 presents related work. Sections 3, 4, and 5 discuss and provide solutions to the three main logging-related bottlenecks, respectively. Combined they provide a scalable logging solution for single-socket multicore machines. Section 7 touches upon the problem of scaling logging on multisocket multicore machines, and Sect. 8 concludes. Finally, for the practitioner interested in a concrete implementation, Sect. A of the appendix gives detailed descriptions and pseudocode for the algorithms discussed in Sect. 5.

2 Related work

As a core database service, logging has been the focus of extensive research. Within a node, the log is typically implemented as a single global structure shared by every transaction, making it a potential bottleneck in highly parallel systems. In Sect. 2.1 we present a representative sampling of the related work on addressing logging-related problems, while Sect. 2.2 discusses work related to distributed logging.

2.1 Handling logging-related problems

Logging is one of the most important components of a database system, but also is one of the most complicated. Even in a single-threaded database engine the overhead of logging is significant. Harizopoulos et al. [11] report that in a single-threaded storage manager, logging accounts for roughly 12% of the total time in a typical OLTP workload.

Virtually all database engines employ some variant of ARIES [23], 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 [22].

DeWitt et al. [7] observe that a transaction can safely release its locks before flushing its log records to disk provided that certain conditions are met. IMS/VS [8] implemented this optimization but its correctness was only proven more recently [34]. We refer to this technique as ELR and evaluate it further in Sect. 3.

Several recent studies, such as [4,19], evaluate solid-state flash drives in the context of logging. Those studies 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 [12,31] reduce the pressure on (magnetic- or flash-based) 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. To significantly reduce the number of context switches due to logging, we present a technique called FlushPipeling and evaluate it further in Sect. 4.

Asynchronous commit [26,30] 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 from becoming durable. Despite being unsafe, asynchronous commit is used widely in commercial and open-source database systems because it provides such a large performance boost. In contrast, Aether gives the same performance boost without sacrificing durability. Furthermore, neither faster I/O devices, such as solid-state flash drives, nor techniques such as log group commit or asynchronous commit handle the problem of contention for inserting records in the log buffer, a main memory resident data structure.

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 [35], replicating each transaction to multiple database instances and relying on hot fail-over to maintain durability. However, replication has its own very large set of challenges [9], and it is a field of active research [36]. Aether is well suited for using in-memory databases because it addresses both log flush delays and contention at the log buffer.

2.2 Distributed logging

In Sect. 7 we investigate the possibility of scaling the performance of log buffer insertions using a distributed logging mechanism. To our knowledge this is the first attempt at improving log buffer insertion performance within a single node (even though multi-socket) using a distributed logging mechanism. To date, the only uses of distributed logging arise in the context of multi-node systems, often for fault tolerance purposes (e.g., [5,20]). Distributed systems must either write back dirty pages when they migrate between logs [17] or else flush the log at every page migration to preclude holes in the event of a crash [21]. Some systems [6] simply maintain a single shared log to avoid the additional complexity of distributed logging.

Traditional log implementations enforce an explicit total ordering of log entries. The order is important for recreating a consistent state of the database during recovery. In reality, however, there are more than one correct orders and often a partial order should be enough. Lomet [20] describes a redo logging method in a distributed environment where each node maintains a private log. The author differentiates between crash recovery and media recovery. When a crash occurs, the system can recover using just one log. On the other hand, when a storage medium crashes, the contents of the medium are recovered by merging the existing logs. During the merging, multiple acceptable orderings of log records are possible. Indeed, only the order between log records of the same private log should be maintained making the merging straight-forward. To ensure that each crash can be recovered using one log, dirty pages must be written back to persistent storage before they can migrate to other nodes. We find that migrations occur far too often for this approach to be feasible in a multi-socket multicore system.

In the following several sections we focus on three logging-related problems, while in Sect. 7 we shift our attention to machines with multiple sockets.

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 (i.e., 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. [7] 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 [34], 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.

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 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 [26,30], 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 [37] and TPC-C [38] benchmarks to evaluate ELR. The benchmark executes on a 64-context Niagara II server running the Shore-MT storage manager [15] (further details about the platform and experimental methodology can be found in Sect. 6).

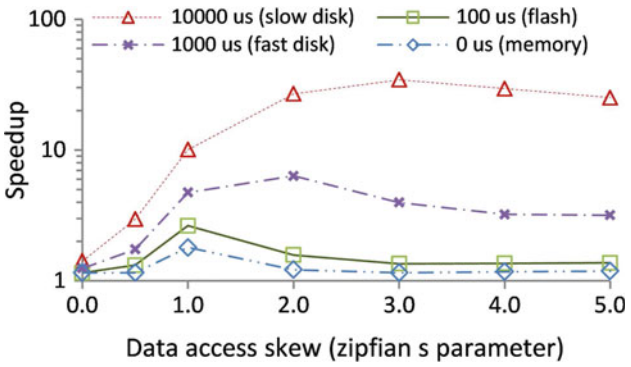

Fig. 3 Speedup due to ELR when running the TPC-B benchmark and varying I/O latency and skew in data accesses. Higher values indicate more improvement due to ELR

Figure 3 shows the benefit of ELR over a baseline system when we run the TPC-B benchmark and 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 10 ms. The first series (0 ms) is measured using a ram-disk which imposes almost no additional delay beyond a round trip through the OS kernel (40–80 μs). The remaining series are created by using a combination of asynchronous I/O and high resolution timers to impose additional response times of 100 μs (fast flash drive), 1 ms (fast magnetic drive), and 10 ms (slow magnetic drive).

TPC-B may exhibit significant lock contention. As shown in the graph, ELR's speedup is maximized (35×) for slower devices, but remains substantial (2×) even with flash drives if contention is present. This effect occurs because transactions are short even compared to 100 μs 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.

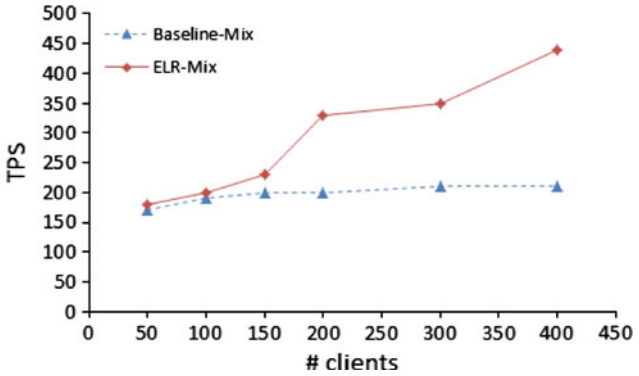

Fig. 4 Impact of ELR in a system running the TPC-C benchmark and the log is stored on a magnetic disk

The sweet spot becomes wider as longer I/O times stretch out the total transaction length in the baseline case. Finally, by way of 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 Fig. 4 we present the throughput of the baseline and an ELR-enabled system (using Shore-MT as well) when running the TPC-C benchmark on a 100WH database. We keep the log device latency equal to 4 ms and we vary the number of the threads executing transactions. We observe that apart from skewness, the number of concurrent clients issuing transactions has important effect in a system with ELR compared against a system without ELR. As the number of concurrent clients increases, the lock contention is higher. In the baseline, the lock contention does not allow the system to achieve higher throughput since clients accessing the same locks have to wait for the log flush to take place. On the contrary, using ELR we can take advantage of the available parallelism and serve a higher number of requests concurrently, increasing the throughput 1.1×, 1.5×, 2.2× for 150, 200, and 400 threads, respectively. We observe that ELR can increase concurrency both when data accesses are skewed and when the number of concurrent clients increases, leading to more accesses per time on the same data locations.

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: (a) the actual I/O wait time and (b) 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 Fig. 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 Fig. 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. Scheduling decisions and thread switching, which usually take 10–20 μs each, thus become 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 5 (left) shows an example of the scheduler overload induced when the Shore-MT storage manager runs the TPC-B benchmark 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.

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

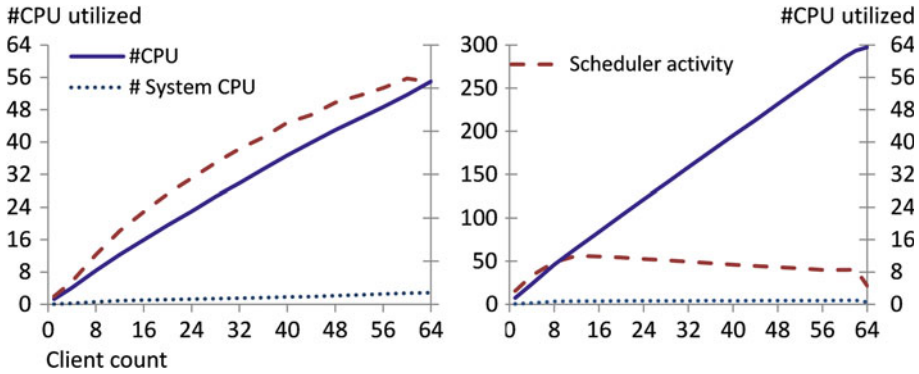

Fig. 5 Comparison of CPU utilization and OS scheduler activity without (*left*) and with (*right*) pipelined log flush optimization active. In a healthy system, utilization is high and system overheads (context switching and system time) are low

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, encore 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 flush daemon.[3]

When combined with ELR (see previous section), flush pipelining provides the same throughput as asynchronous commit without sacrificing any safety.[4] 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 Fig. 5 (left), but this time with flush pipelining active. Figure 5 (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. 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

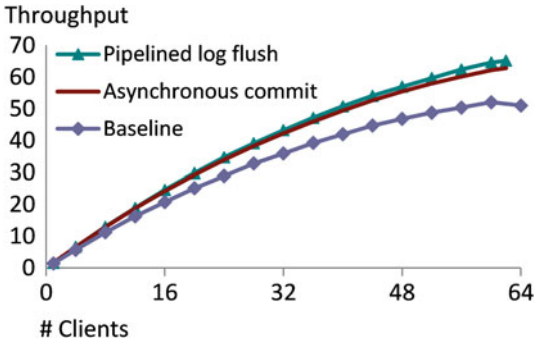

Fig. 6 Performance comparison of the baseline system, flush pipelining, and asynchronous commit. Higher values are better

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 6 compares the performance of baseline Shore-MT to asynchronous commit and flush pipelining when running the TPC-B benchmark. 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 Sect. 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 for multicore

Most database engines use some variant of ARIES [23], which assigns each log record a unique log sequence

[3] Most transaction rollbacks not due to deadlocks arise because of invalid inputs; these usually abort before generating any log and do not have to be considered.

[4] 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.

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. Still, we explore this option in Sect. 7.2.

Because of its serial nature, LSN generation and the accompanying log inserts impose serious limitation on parallelism in the system. In this section we attack the problem at its root, developing techniques which allow LSN generation to proceed in parallel. We achieve parallelism by adapting the concept of "elimination" [33] to allow the system to generate sequence numbers in groups. An especially desirable effect of this grouping is that increased load leads to larger groups rather than causing contention. We also explore the performance trade-offs that come from decoupling the LSN generation process from the actual log buffer insert operation.

We begin by considering the basic log insertion algorithm, which 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
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.

Baseline implementation 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 (see Algorithm 1 in Sect. A.1 of the appendix for a concrete example).

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. Further, even though LSN generation is fully serial, it is also short and predictable (barring exceptional situations such as buffer wraparound or full log buffer, which are comparatively rare).

The monolithic log insert suffers a major weakness because it serializes buffer fill operations, *even though buffer regions never overlap*, adding their cost directly to the critical path. In addition, log record sizes vary significantly, making copying costs unpredictable. Figure 7(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 40 and 264 B (a 6× 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 [13,24], a hybrid approach combining elimination trees [33] 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

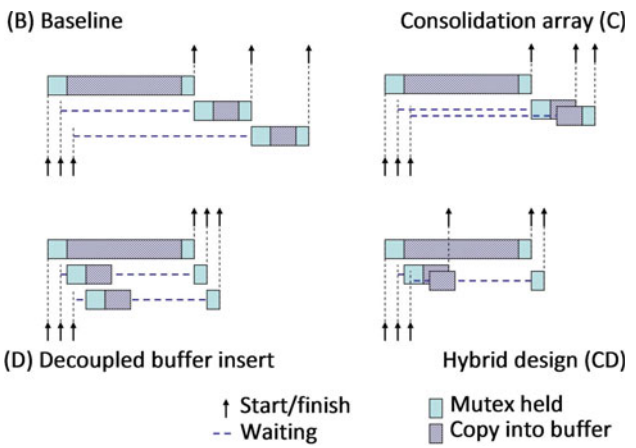

Fig. 7 Illustrations of several log buffer designs. The baseline system can be optimized for shorter critical path (D), fewer threads attempting log inserts (C), or both (CD)

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 (Sect. A.2 of the 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 7(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.

5.2 Decoupling buffer fill and delegating release

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, Sect. A.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 7(D) illustrates the improved concurrency that results with significantly reduced wait times at the buffer acquire phase. Under most circumstances, log record sizes do not vary enough that threads wait for previous ones to release the buffer, but high skew in the record size distribution will limit scalability because a very large record will force small ones which follow to wait for it to complete. A further optimization (not shown in the figure) allows threads to *delegate* their buffer release to a predecessor which has still not completed.

To summarize the delegated buffer release protocol, threads which would normally have to wait for a predecessor instead attempt to mark their buffer as *abandoned* using an atomic compare-and-swap operation. Threads which succeed in abandoning their buffer before the predecessor notifies them are free to leave, forcing the predecessor to release all buffers that would have waited for it. In addition to making the system much less sensitive to large log inserts, it also improves performance because a single thread releases groups of buffers in a tight loop rather than communicating the releases with other threads.

5.3 Putting it all together: hybrid log buffer

In the previous two sections we outlined (a) a consolidation array which reduces the number of threads entering the log insert 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, but the two are orthogonal 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 allow 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 Fig. 7(CD), achieves bounded contention for threads in the buffer acquire stage and maximum pipelining of all operations. As we will see later, this hybrid version consistently outperforms the other configurations by combining their best features.

6 Evaluation of log buffer optimizations

We implement the techniques described in Sects. 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 Sects. 3 and 4, respectively. Unless otherwise stated, in this subsection 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 64 GB 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 [15], an open-source multi-threaded storage manager. We developed Shore-MT by modifying the SHORE storage manager [3] to achieve high 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 [14]. We run the following benchmarks:

TATP TATP [25]. 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 100 K Subscribers.

TPC-B This benchmark [37]. 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 data set.

TPC-C TPC-C [38]. models an online transaction processing database for a retailer. It consists of five transactions which follow customer orders from initial creation to final delivery and payment. We use a database of 100 Warehouses.

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 [35], we use memory-resident data sets, 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, because problematic functions can be seen to shift their position in the timing breakdowns.

We note that the hardware limits scalability somewhat 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 log buffer contention after the ELR and the flush pipelining have been applied. Figure 8 shows the time breakdown for Shore-MT with 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 Sects. 5.1–5.3 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 the paragraphs which follow explore the impact of skew in the record size distribution and of changing the number of slots in the slot array.

Scalability with respect to thread count The most important metric of a log buffer is how many insertions it can sustain

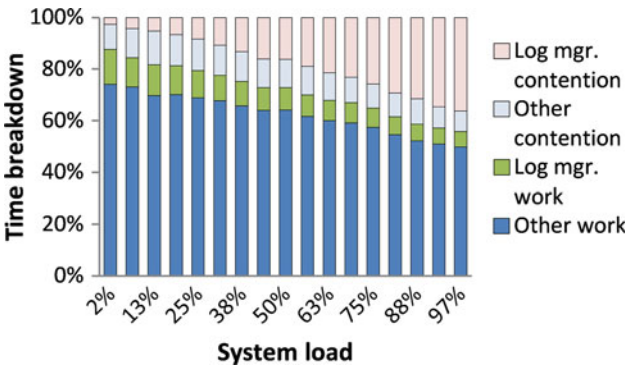

Fig. 8 Breakdown of the execution time of Shore-MT with ELR and flush pipelining, running TATP-UpdateLocation transactions, as load increases. The log buffer becomes the bottleneck

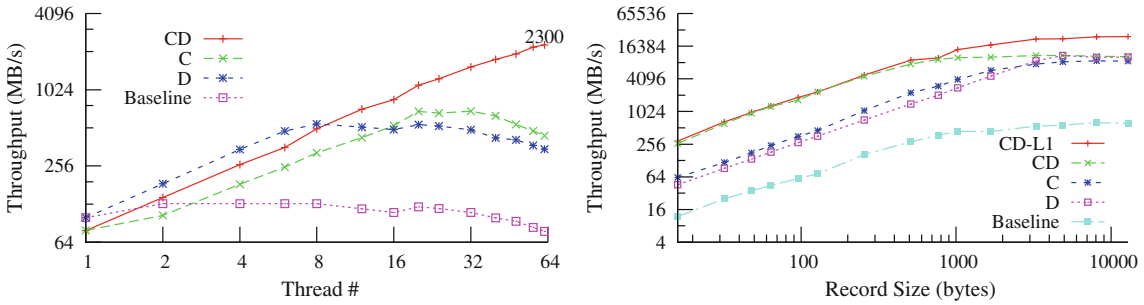

Fig. 9 Sensitivity analysis of the C-Array with respect to thread counts and log record size

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 200 MBps of log, or between 800 K and 1.6 M log insertions per second.

Figure 9 (left) shows the performance of the log insertion microbenchmark for records of an average size of 120 B 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 140 MB/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 Sect. 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.

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 9 (right) shows the impact of these two factors, plotting sustained bandwidth achieved by 64 threads as they insert log records ranging between 48 B and 12 kB (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 1 kB, 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 21 GBps (nearly 20× higher throughput than modern systems can reach).

Robustness to skewed record sizes. In Fig. 10 we test the stability of the consolidated buffer acquire with delegated buffer release (CDE, for short) and compare it with the hybrid variant (CD) from Sect. 5.3. We use the same microbenchmark setup from Sect. 6 but modify it to present the worst-case

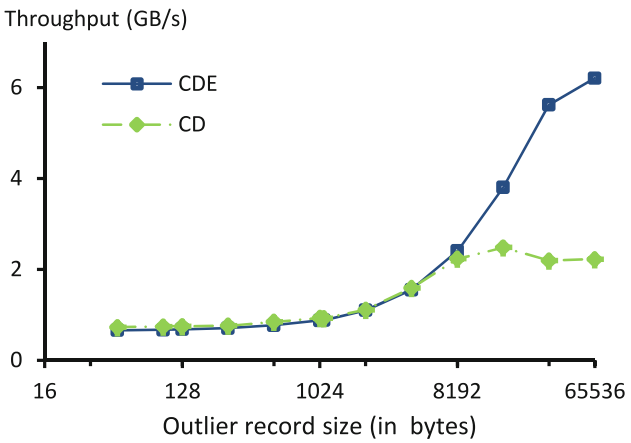

Fig. 10 Performance impact of log record size skew

scenario for the CD algorithm: a strongly bi-modal distribution of log record sizes. We fix one peak at 48 bytes (the smallest and the most common log record size 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 CDE perform similarly until an outlier size of around 8 kB, when CD stops scaling and its performance levels off. CDE, which is immune to record size variability, achieves up to double the performance of the CD for outlier records larger than 65 kB.

The CDE 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 12 kB with a frequency of 0.01% of the total log inserts. In addition, CDE achieves around 10% lower throughput than the CD variant under normal circumstance, making it unattractive. Nevertheless, for other configurations which encounter significant skew, the CDE algorithm might be attractive given its stability guarantee.

Sensitivity to slot array size. Our last microbenchmark analyzes whether (and by how much) the consolidation arrays 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 11 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 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.

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 Fig. 1 and can identify optimizations which are likely to have highest impact now and in the future.

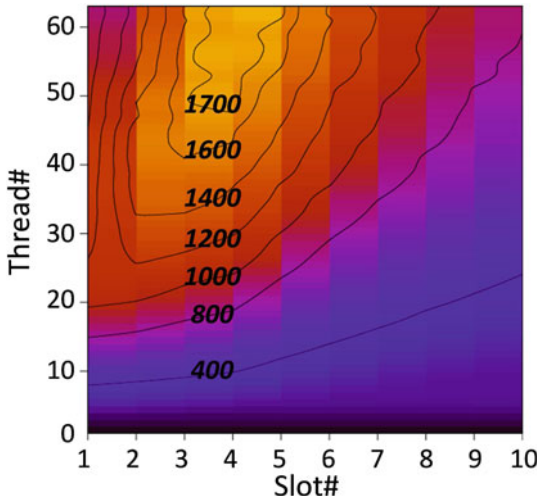

Fig. 11 Sensitivity to the number of slots in the consolidation array

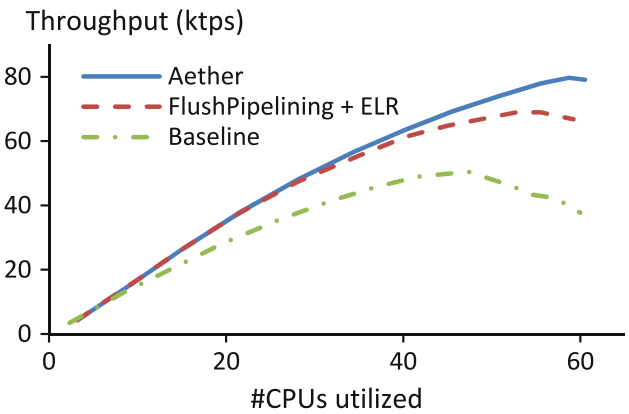

Fig. 12 Overall performance improvement provided by each component of Aether when running the TPC-B benchmark

Figure 12 captures the scalability of Shore-MT running the `TPC-B` benchmark. 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, 37% higher peak performance than the baseline. The scalable log buffer adds a 15% further speedup by eliminating log contention. Overall the Aether logging subsystem achieves 58% higher performance than the baseline implementation.

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 the most serious bottleneck unless techniques such as the hybrid implementation presented in this section are used. Even today, contention at the log buffer impacts scalability to considerable degree. In addition, the profiling results from Fig. 8 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 modern 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 Scalable logging on multiple sockets

Server designs today increasingly utilize non-uniform memory architectures (NUMA) where groups of cores or CPUs closely located (islands of cores) can access their own local memory faster than memory belonging to other islands. For example, in a multi-socket machine it is common for each CPU to have one or more dedicated memory controllers which provide much faster access than the memories of other sockets. Such configurations are attractive because memory bandwidth is proportional to the number of processing units. Inter-island communication, however, is much slower and has lower available bandwidth compared to the local memory; additionally, the programming model becomes more complicated because programmers must now pay attention to the island where each chunk of memory is allocated.

In this section we investigate the scalability of Aether to multi-socket architectures, which put several multicore processors together for even higher parallelism than a traditional chip multiprocessor machine. To test the scalability of the Aether logging system, we use a Sun Niagara II machine very similar to the one in Sect. 6.1 except that this time the server boasts 4 sockets, each with a Niagara II chip, for a total of 256 hardware contexts.

In Fig. 13 we run the microbenchmark of Sect. 6.3 on the multisocket Niagara II machine to test the scalability of the combination array (labeled as *CD-Vanilla*) by varying along the *x*-axis the number of threads that access the log, while measuring the throughput along the *y*-axis. For comparison purposes, we also show the throughput achieved by Aether in the single-socket case (labeled as *CD-SocketBound*). As we can see, for a single thread the combination array achieves the same performance on both the single-socket and multi-socket machine. Between two and four threads, however, the throughput of the system drops by a factor of 2 with every new thread added due to the increasing inter-socket communication. Next, for thread counts up to 32, Aether scales linearly with the number of added threads but is unable to bridge the performance gap imposed by inter-chip communication. After 32 threads, the contention caused by the growing number of threads explodes and brings performance almost to the single-thread level. This is caused by expensive atomic operations used by lock hand-offs and by the management of the combination array; off-chip accesses increase latency inside important critical sections, exacerbating contention and making the optimal number of slots vary widely between 3 and 10.

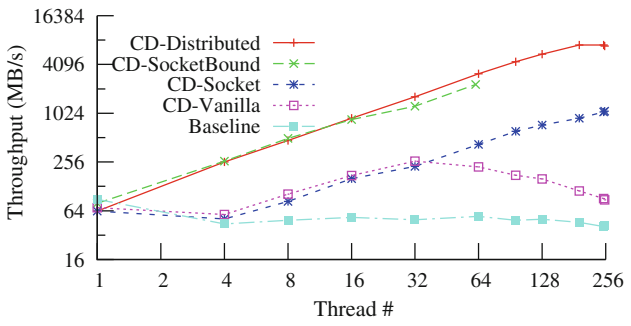

Fig. 13 Throughput of the three Aether variations on a four-socket multicore system

7.1 Optimizing the log buffer design for NUMA

From the Fig. 13, we identify two main performance problems. First, high contention due to increased latency means the consolidation array no longer scales effectively as thread counts increase beyond 64 threads. Second, latency reduces single-thread performance to the point that, even with perfect scalability beyond 64 threads, the NUMA machine would barely match the performance of its smaller sibling with 1/4 the processing power!

In the following subsections we discuss two improvements to the combination array implementation which make it more suitable for NUMA accesses by minimizing inter-socket communication. The first is illustrated by Fig. 14b. We observe that allowing threads from different sockets to combine into the same group is counter-productive because the latency of crossing socket boundaries is higher than the wait without any consolidation at all. We therefore introduce separate combination arrays for every socket. Threads running on a given socket can combine their requests with others from the same socket, utilizing socket-local data structures. Only the group leaders are required to make the expensive requests for memory as they acquire the log insert mutex on behalf of the entire socket. This leads to orderly, round robin-style hand-offs as log state migrates between sockets. These optimizations have multiple advantages: (i) they eliminate inter-socket communication overhead for the management of the logic of the combination array (as all combinations are done on socket-local data structures), (ii) they reduce cache coherency communication when writing to the single log buffer, because large aggregated log records will not be as prone to false sharing of cache lines, and (iii) a single lock hand-off services all requests of a socket.

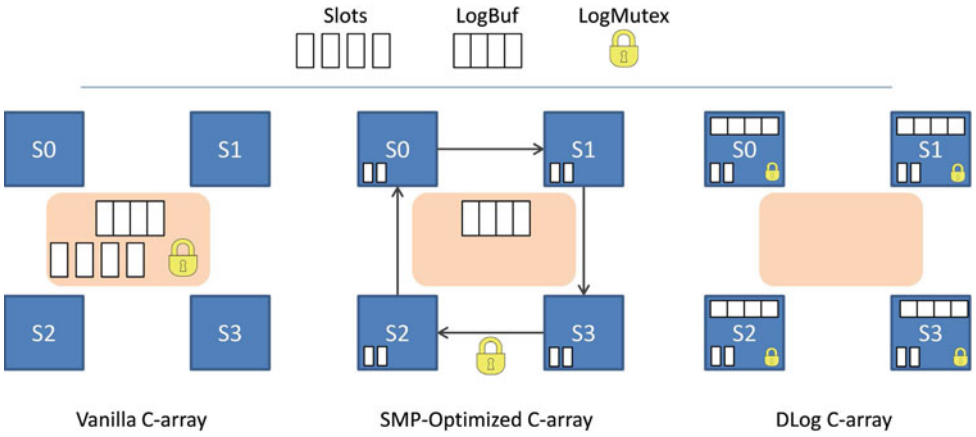

Fig. 14 Modifications of the Aether combination array for scaling to NUMA architectures

We present in Fig. 13 the results for the improved multi-socket combination array (called *CD-Socket*). As we can see, the drop in performance due to contention for slots for higher thread counts is fully addressed. Unfortunately, the performance hit between 2 and 4 threads remains, albeit somewhat smaller than it was for the vanilla combination array implementation. Thus, the combination array is successful in handling contention but it cannot compensate for the long latencies NUMA imposes on lock hand-off and log buffer writing times. This result suggests that the only way to avoid the penalty is to also avoid most inter-socket communication. This leads us to our second improvement, a distributed log implementation designed to avoid latency penalties within a single process.

7.2 Distributed logging

A distributed log has the potential to ease bottlenecks by spreading load over N logs instead of just one; if we bind each log to a socket, we can further apply our single-socket optimizations such as the consolidation array. This design is shown on the far right of Fig. 14. Intuitively, it should be possible to parallelize the log, given that most transactions execute in parallel without conflicts. In particular, the reason we suffer log contention at all is because many transactions attempt to make independent updates which the system did not need to order with respect to each other. However, several issues must be addressed by a potential distributed log:

- Because it is prohibitively expensive to write back pages every time they migrate, or to track dependencies due to migrations, transactions must flush all logs when they commit, increasing latency drastically.
- Log sequence numbers (LSN), the de-facto unit of time in the database, are incomparable unless they refer to the same log. We need some other method for establishing a system-wide notion of time.

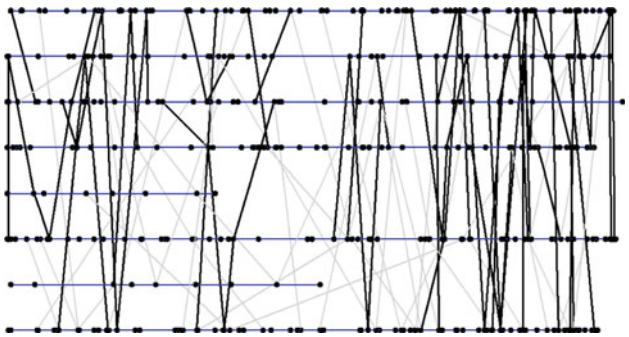

Fig. 15 Inter-log dependencies for 1 ms of TPC-C (8 logs, 100 kB, 30 commits)

- Log files may not all flush at the same rate, leading to "holes" in the log which can lead to unrecoverable situations, such as redoing the deletion of a record which was never added. ELR makes this worse, because transactions can commit without flushing and later transactions now depend on data which may be lost in a hole.

The following subsections highlight in turn each challenge and our proposed solution; more detailed descriptions follow in the next subsection.

7.2.1 Dependency tracking in a distributed log

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,[5] as a transaction's commit will never reach disk before its dependencies. A distributed log removes that latter

[5] 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.

condition, and must therefore track transaction dependencies and ensure that logs become durable in a coherent order, as discussed by DeWitt et al. [7]. In addition, it must do so without requiring multiple log flushes per transaction, or the serial implementation will actually be faster.

Unfortunately, this challenge is difficult to address by simple partitioning schemes because physical dependencies can be very tight, especially with hot database pages. For example, Fig. 15 shows the dependencies which would arise in an 8-way distributed log for a system running the TPC-C benchmark [38]. 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 100 kB of log records, which corresponds to less than 1 ms wall time and about a dozen transaction commits.

Our experiments show that, on a four-socket system pages have a 30–40% chance of migrating sockets after any given use. This migration ratio holds for both TATP (with very small transactions) and for TPC-C (with significantly larger transactions). Similarly, there is no obvious way to partition transactions in such a way that they access only a certain subset of the data, as required for partitioned/shared-nothing systems to avoid two-phase commit.

Rather than track dependencies, we assume a relatively small number of sockets in the system and flush all logs at every transaction commit. Each transaction is pinned and inserts records only to the log of the socket it started on, which means that log records for the same data page are allowed to be recorded freely to any log. We rely on early lock release and flush pipelining to hide the associated latencies and allow the system to maintain high throughput. The design of our distributed flush protocol only requires one log record, in the transaction's local log.

7.2.2 Replacing the LSN with a global clock

Because log sequence numbers from different logs cannot be used to order log records globally, we must find some other mechanism to track the flow of time so that during recovery, logs can be merged properly. To this end, we propose a global sequence numbering (GSN) scheme based on Lamport timestamps [18], which allows us to establish a total order for all updates to a given page or transaction while still permitting parallelism between pages and transactions. We then replace the LSN everywhere that global timekeeping is required. In particular, we convert the page recovery LSN to a GSN by giving each page a Lamport timestamp, and log recovery merges logs based on record GSNs. We discuss this scheme in more detail in the next section.

7.2.3 Avoiding holes in the log

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. Also, log files may not all flush at the same rate, leading to holes in the log. Judging by Fig. 15 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. The two primary techniques for preserving log integrity are to write back dirty pages whenever they migrate (e.g., Oracle CacheFusion [17]), or to flush the log at each migration (Rdb/VMS [21]). Because page migrations are so frequent in our system, occurring on 30–40% of accesses, both approaches impose unacceptable I/O penalties.

To avoid holes without an I/O for every migration, we require every externally visible event, such as a transaction commit or dirty page writeback, to synchronize with all logs. It does so by recording the tail of each log as part of a log sync record, which is inserted into the transaction's local log. At recovery time, the system can examine these sync records in order to identify holes in the log; the system then truncates the log to remove holes. Ensuring that the logs flush successfully before a transaction returns results to the user ensures that it is safe to truncate in this fashion—by definition, any changes dependent on a hole in the log were never exposed to the world and can be ignored.

7.3 Lamport clocks

The Lamport clock [18] is a distributed time-tracking device which defines a loose partial ordering of events in the system. Conceptually, Lamport clocks are based on message passing: "processes" (threads, transactions, workers, etc.) perform "actions" (computation) which are ordered locally but not globally. Processes send "messages" to each other when they need to communicate, and these messages provide a partial ordering for events in the system. Every process maintains a timestamp under the following rules:

1. Every message is accompanied by its sender's current timestamp
2. A receiving process updates its own timestamp to be at least as large as the one in the message.
3. Every meaningful action or message received is an "event" and increments the process timestamp.

The above rules order events relative to inter-process communication: if two processes never communicate, their timestamps will be completely unrelated to each other and can be

said to have occurred "simultaneously." On the other hand, two processes which communicate heavily will have much stronger ordering imposed as they repeatedly synchronize their timestamps, but still the ordering is too weak to establish causality for events originating from different processes.

Formally, if we let $C(x)$ be the timestamp of event x, and define $A \rightarrow B$ to mean "A happened before B", then $A \rightarrow B \Rightarrow C(A) < C(B)$, but the opposite does not hold. That is, we cannot say with certainty that $A \rightarrow B$ based only on Lamport timestamps $C(A)$ and $C(B)$ (unless A and B are from the same process). Recall that all events from two processes which never communicate occur "at the same time" even though the corresponding timestamps may very well be larger or smaller timestamps relative to each other. However, given events A with timestamp $C(A)$, we can say with certainty that $C(A) \not< C(B) \Rightarrow A \not\rightarrow B$ (A can only have occurred "before" B if its timestamp is smaller).

In the database system, we are interested in three types of "processes": the records inserted into a given log, the reads and writes performed by a transaction, and the updates to objects such as database pages. We therefore maintain a Lamport timestamp for each of these, allowing all events involving the same page, etc., to be totally ordered. Our inability to infer happened-before relationships for other events is unimportant, because pessimistic concurrency control ensures that any two log entries which can occur simultaneously are independent. Therefore, establishing that B *did not* happen before A is enough to safely replay A before B, because if the events were not independent the dependency would have been captured when synchronizing timestamps.

Figure 16 illustrates the three main types of communication which arise in the database system: page reads, updates, and log synchronization. The first involves a transaction and a page only (no logging), the second involves a transaction and the log (and usually a page as well), while the third involves multiple logs only.

Before continuing, we should point out that logging in a database system confuses matters slightly because some events (those which generate log records) are more important than others (those which merely update timestamps), because only the former are available at recovery time. We therefore use solid circles to denote logged events, while hollow circles represent timestamp updates (which serve only to impose order on future log records). In each example, the starting timestamps are denoted by hollow circles filled with a hatched pattern, and a timestamp synchronization is denoted with an arrow whose length shows how much the timestamp changed. Time flows from left to right.

7.3.1 Page read

When a transaction reads a page (Fig. 16a, b) it synchronizes its timestamp with that of the page while the page's timestamp remains unchanged. For example, part (a) of Fig. 16 shows the case where transaction T reads a "newer" page P and must update its timestamp to reflect this fact. Part (b), on the other hand, shows a transaction which reads an "older" page and does not respond to the page's smaller timestamp.

Conceptually, we can say the page broadcasts a message (its contents) whenever it is updated, and transactions which eventually read that version of the page can be said to have "received" the message. Synchronizing the transaction with the page is important to capture the flow of information between pages. For example, if a transaction building an index reads record R from page P_1 without synchronizing its timestamp, a later insertion to page P_2 could have $C(P_2) < C(P_1)$. Should a crash occur, recovery would wrongly conclude that page P_2 should be recovered first (because it could not have occurred after the change to P_1), leaving a window of vulnerability where index probes find the entry in P_2 but fail to find the corresponding value in page P_1. By synchronizing T's timestamp when it reads P_1, we ensure that any later actions which depend on that read (such as the insert to P_2) will be recovered in order.

7.3.2 Log writes

Log writes are the only source of timing information available after a crash, and must therefore preserve all ordering needed to maintain consistency of the recovered data. When a transaction T inserts a record into log L, both must synchronize their timestamps to the larger of the two; if a page P

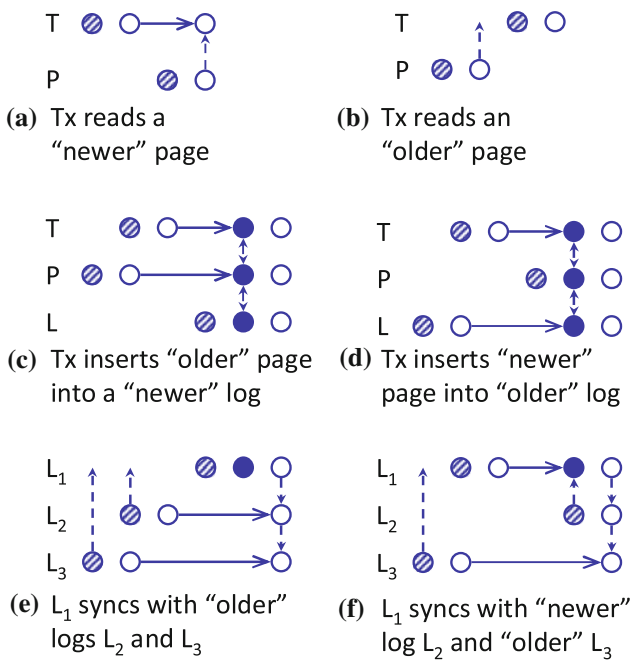

Fig. 16 Examples of Lamport timestamp synchronization in the distributed log

is involved it must also be synchronized. This synchronization ensures proper ordering of updates to a page, as well as history in the transaction and log. While in a truly distributed system such three-way synchronization would be impossible, in our single-node system the transaction, log, and page have already been brought together by the computation and synchronizing them is inexpensive. Part (c) of Fig. 16 illustrates the case where the page and transaction must be brought forward to match the log's timestamp, while part (d) shows the case where the transaction and log are behind in time before the insert. Either way, all three entities have the same timestamp once the log insertion completes.

Although log records are already totally ordered by their log sequence number (LSN, or address on disk), log timestamps are important because pages frequently migrate between logs. When page P is updated in log L_1 and then L_2, timestamp synchronization ensures that recovery processes L_1's record first. Log sequence numbers, which are only comparable for records in the same log, do not provide this necessary information.

7.3.3 Log synchronization

Whenever a thread is about to make updated state externally visible, it must first synchronize all logs in the system. This is a three-step process:

1. The thread records the current timestamp of every log and updates each log's timestamp to be no smaller than its own.
2. The thread inserts a synchronization record—containing the recorded timestamps—into its own log
3. The thread requests the system to harden all logs up to its own timestamp, waiting for the I/O to complete before continuing.

Figure 16e and f illustrate the first two steps. In (e), the calling log is the most advanced in time and therefore updates only the other logs' timestamps; in (f), the calling log is neither the most nor the least advanced, and must update both its own and others' timestamps.

Log synchronization is required in order to permit recovery to detect "holes" in the log which would otherwise render the system unrecoverable. We discuss log holes in the next section.

7.4 Recovery and holes in the distributed log

Perhaps the biggest challenge of distributing the log is the possibility of encountering "holes"—log records which did not make it to stable storage before a crash but whose successors did. Holes are not a problem for a single log because all records go to the same place; if any earlier record is missing, then none of the later ones were written either. While it is technically possible for holes to arise in a single log, recovery avoids any problems by truncating the log at the first invalid or missing record it finds. In a distributed log, however, recovery has no way to detect whether dependent records from a different log are missing because the available Lamport timestamps do not establish causality. If log L_1 and L_2 have $C(L_2) < C(L_1)$, we may have lost log records in the crash and it is tempting to truncate L_1 at $C(L_2)$. Unfortunately, it's just as likely that L_2 was fully up to date and simply contained no later records, in which case truncating L_1 could lose committed data.

One workaround would be to force each externally visible action (transaction commit or dirty page write back) to insert into every log, thus synchronizing their timestamps and ensuring that recovery knows "when" in each log the action occurred. However, this approach suffers from two major weaknesses. First, forcing every transaction commit to insert into every log increases drastically the number of log records inserted and also obviates much of the benefits we might gain by pinning transactions to socket-local log managers. Second, transactions frequently update different records stored in the same page, and a small hole in the log can render the page unrecoverable even if the transactions involved never committed.[6] Forcing transactions to record every page update in every log would completely defeat the point of distributing the log in the first place.

Our solution, which we described earlier, is to have transactions record log dependencies in a single log record before making changes visible to the outside world. As long as they ensure each log is hardened up to the recorded timestamp before continuing, the system can recover the log. This approach has the advantage that it only requires the transaction to ever write to its own log, and we can minimize the additional overhead by embedding the dependency information in the commit record which has to be written anyway.

At recovery time, the system analyzes each log individually, recording the last synchronization record whose dependencies are all met. After analysis completes, the system selects the maximum timestamp for each log out of all completed sync records and truncates the log there. We can safely truncate because by definition no later events (including attempted a second attempted sync) were exposed to the outside world and can therefore be ignored. Because every synchronization advances the other logs' Lamport timestamps, it is impossible for one sync record to have unmet dependencies while a later synchronization originating from the same

[6] For example, suppose that T_1 inserts R_i into slotted page P, and T_2 later inserts R_j. Suppose that T_1 and T_2 are bound to logs L_1 and L_2, and only L_2 hardens before a crash. Recovery will attempt to insert R_j past-end in the slot array and flag the page as corrupted.

log is fully satisfied. Log redo then processes log records, smallest timestamp first, until it reaches the last sync record of each log. Every log record redone is thus guaranteed to have all its dependencies from other logs met. Finally, log undo proceeds as usual, rolling back all in-progress transactions discovered in all logs, largest timestamp first.

7.5 Performance of the distributed log

Our preliminary evaluation of distributed logging takes two forms. First, we modify the consolidation array microbenchmark to incorporate per-socket logging, and second, we implement our distributed design in Shore-MT. We test both on a four-socket Niagara II machine with 256 hardware contexts.

7.5.1 Experimental setup

The primary difference with our main experimental setup is the need for running threads to identify their current socket. This turns out to be straightforward in x86 machines, where the `CPUID` instruction is an efficient source of this information. On our SPARC hardware, however, we rely on Sun's DTrace [2] to cooperate with our program to identify the logical CPU (and hence socket) which each thread runs on. We achieve this by instrumenting DTrace to capture the address of a thread-local variable at thread startup, into which it writes the CPU ID every time a thread is scheduled on-CPU. While it is technically possible with both DTrace and the `CPUID` instruction to have an untimely preemption which migrates the thread to a new socket, this would only cause a slight performance drop due to additional cache misses. Correct execution does not rely on the thread being pinned to a given socket for any amount of time.

7.5.2 Consolidation array microbenchmark

In order to make a case for distributed logging, we extend the consolidation array microbenchmark to create one full log buffer replica for each socket in the system. Threads then identify which socket's log they should use. The performance of this technique, when combined with the normal consolidation array within each socket, is shown as the *CD-Distributed* series of Fig. 13. The latency penalty suffered by previous schemes is completely eliminated, and performance scales linearly until the machine is nearly saturated. The slight dip at the end occurs because some memory is allocated on the wrong sockets and leads to unwanted coherence traffic; for very high thread counts this traffic overwhelms the interconnect between sockets and slows down the system.

As a further experiment (not shown), we simulated the case where threads must synchronize the logs after every few log inserts. In the very worst case, an update transaction would insert exactly two records: one for a single change to the database, and a commit record. Our profiling shows that TATP transactions insert 3–5 log records per transaction, while TPC-C transactions generate 75 or more. However, we found that adding log synchronization reduced performance by less than 5%. This low cost makes sense because log synchronization does not generate extra log records in the other logs, only a memory read and single atomic swap in the case where the other log is out of date compared with the current one.

The high performance of the distributed c-array, combined with the low synchronization penalty we observe, lead us to conclude that the distributed log mechanism is a promising way to reduce log contention in the system.

7.6 Distributed log in Shore-MT

Following the design outlined in Sect. 7.3, we implemented a distributed log in Shore-MT. We then evaluate the performance of this version of Shore-MT using the TATP benchmark. While TATP provides the high update rates which lead to contention, it also poses a worst case for the distributed log because transactions are so short that they cannot amortize the cost of a log sync over very many other log record insertions. Table 1 summarizes the results of these experiments when run on our four-socket Niagara II machine, as well as a four-socket Intel Xeon with six cores per socket (24 contexts total with hyperthreading disabled).

For each machine, we examine two cases: the mix of transactions of TATP (15% of the transactions are updates), and isolated the read-only `GetSubscriberData` transaction. Further, we examine three cases: the baseline Shore-MT as well as the distributed log when accessed by socket (Dlog-S) or randomly (Dlog-R). We would expect that the socket-bound distributed log will be more expensive than the baseline, but scale better than either the baseline or the case where threads access logs randomly rather than by socket.

In practice, we see that the distributed log costs anywhere from 5–15% performance due to its increased complexity and the need to log both GSN and LSN information for each log record. Unfortunately, we find that the hoped-for scalability does not materialize because applying the distributed log only

Table 1 Comparison of the performance of baseline and distributed logging on two multisocket machines

	Niagara		Xeon	
	TATP (ktps)	GSD	TATP (ktps)	GSD
Baseline	256	551 ktps	326	726 ktps
DLog-S	260	515 ktps	275	720 ktps
DLog-R	247	N/A	267	N/A

shifts the bottleneck to other parts of the database engine. While we eliminated several of these bottlenecks, we came to the conclusion that there are simply too many other sources of cross-socket communication in the system for the distributed log to make much difference over the multisocket-optimized consolidation array. This hypothesis is supported by the small or even non-existent penalty we observe by allowing threads to access logs randomly—there are enough other sources of coherence traffic that the distributed log's contributions are lost in the noise. We note that the distributed log indeed eliminates contention, unlike the multisocket-optimized c-array. Profiling the systems indicates that the log generates no contention to speak of and that other areas such as the lock manager (particularly deadlock detection) are the bottlenecks.

Based on these results, we conclude that, in order to truly benefit from a distributed log, the other memory accesses in the system must also be distributed, perhaps following a design philosophy such as DORA [28] and PLP [29]. These approaches either logically or physically partition the accesses of the worker threads, eliminating most forms of communication in the system while still allowing it when necessary. We leave this integration as future work.

8 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. Finally, we explore potential methods for extending this performance to non-uniform architecture (NUMA) machines, which raise a different set of challenges than single-socket multicore machines. 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.

Appendix A: Detailed log insertion algorithms

This section provides detailed descriptions and pseudocode for the optimizations we discussed in the previous section. It also describes in detail the "consolidation array" algorithm which makes the log insertion optimizations possible. We expect this section to be especially useful for achieving a deeper understanding of these optimizations, and also to allow practitioners to implement them in their own systems.

A.1 Baseline

In a straightforward implementation, a single mutex protects the logs buffer, LSN generation state, 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 (L18). Inside the critical section there are three operations: (i) A thread first allocates log buffer space (L7–11); (ii) It then performs the record insert (L12–15); (iii) Finally, it releases the buffer space making the record insert visible to the flush daemon by incrementing a dedicated pointer (L17). 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.

A.2 Consolidation array

Consolidation-based backoff aims to reduce contention and, more importantly, make it independent of the number of

Algorithm 1 Baseline log insertion

```
1  def log_insert(size, data):
       lock_acquire(L)
       lsn = buffer_acquire(size)
4      buffer_fill(lsn, size, data)
       buffer_release(lsn, size)
   end
7  def buffer_acquire(size):
       /* ensure buffer space available */
       lsn = /* update lsn and buffer state */
10     return lsn
   end
   def buffer_fill(lsn, size, data):
13     /* set record's LSN */
       /* copy data to buffer (may wrap) */
   end
16 def buffer_release(lsn, size):
       /* release buffer up to lsn+size */
       lock_release(L)
19 end
```

Algorithm 2 Log insertion with consolidation

```
   def log_insert(size, data):
 2     if (lock_attempt(L)== SUCCESS)
           /* no contention */
           lsn = buffer_acquire(size)
 5         buffer_fill(lsn, size, data)
           buffer_release(lsn, size)
           return
 8     end
       {s, offset} = slot_join(size)
       if (0 == offset)
11         /* slot owner */
           lock_acquire(L)
           group_size = slot_close(s)
14         lsn = buffer_acquire(group_size)
           slot_notify(s, lsn, group_size)
       else
17         /* wait for owner */
           {lsn, group_size} = slot_wait(s)
       end
20     buffer_fill(lsn+offset, size, data)
       if (slot_release(s, size) == SLOT_DONE)
           /* Last to leave, must release buffer */
23         buffer_release(lsn, group_size)
           slot_free(s)
       end
26 end
```

Algorithm 3 Log insertion with decoupled buffer fill

```
 1 def buffer_acquire(size, data):
       /* wait for buffer space */
       lsn = /* update lsn and buffer state */
 4     lock_release(L)
       return lsn
   end
 7 def buffer_release(lsn, size):
       while (lsn != next_release_lsn)
           /* wait my turn */
10     end
       /* release buffer up to lsn+size */
       next_release_lsn = lsn+size
13 end
```

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 (L2–8). Threads which encounter contention back off to the consolidation array and attempt to join one of its slots at random (L9). The first thread to claim a slot becomes that slot's owner; other threads can join the slot's group while the slot owner (or "group leader") waits for the mutex. Once inside the critical section, the group leader is responsible to acquire buffer space for all waiting threads in its group. It atomically reads the current group size and marks the group as closed (L13); once a slot closes, threads can no longer join the group. The group leader then computes the amount of buffer space needed, acquires it, and notifies the other group members (L14–15) before beginning its own buffer fill. Meanwhile, threads which join the group infer their relative position in the group's meta-request based the group size at the time they joined; once the group leader reports the LSN and buffer location each thread can compute the exact LSN and buffer region which belongs to it (L18 and L20). As each thread leaves (leader included), it decrements the slots reference count; the last thread to leave releases the buffer and frees the slot for reuse (L21–25).

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 (L13). The effect is that the array slot position reopens quickly, even though the threads which consolidated their request are still working on the previous (now-private) resident of that slot. We avoid memory management overheads by allocating a large number of consolidation structures at start-up time, 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* consists of incrementing an array offset. Section A.4 describes the implementation details of the consolidation array.

A.3 Decoupled buffer fill

Decoupling the log insert from holding the mutex reduces the critical section length and also prevents large log records from causing contention. 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 immediately (L4) and performs its buffer fill concurrently with other threads. Once the buffer fill is completed, the thread waits for all other threads to finish their inserts (L8) before it releases its log buffer space (L12). The release stage uses the implicit queuing of the `next_release_lsn` variable to avoid expensive atomic operations or mutex acquisitions.

A.4 Consolidation-based backoff

The consolidated log buffer acquire described in Algorithm 2 uses a new algorithm which we develop here, the consolidation array, to divert contention away from the log buffer. We base our design on the elimination-based backoff algorithm [13], 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 *group 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 groups 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 4 gives pseudocode for the consolidation array implementation, which the following paragraphs describe in further detail, while the next section supplements the pseudocode with a detailed description of a slot's life cycle and state space.

Slot join operation (L14–32). 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 `compare-and-swap` instruction 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

Algorithm 4 Consolidation array implementation

```
1  def atomic_swap_state(s, val):
       /* atomically read /s->state/ and assign
          /val/ to it; return the value read */
4  end
   def atomic_cas_state(s, old_val, new_val):
       /* atomically read /s->state/ and assign
7         /new_val/ to it iff the value read equals
          /old_val/; always return the value read */
   end
10 def atomic_add_state(s, amt):
       /* atomically increment /s->state/ by /amt/;
          return the value stored */
13 end
   def slot_join(size):
     probe_slot:
16     idx = randn(ARRAY_SIZE)
       s = slot_array[idx];
       old_s = s->state
19   join_slot:
       if(old_s < SLOT_READY)
           /* new threads not welcome */
22         goto probe_slot;
       end
       snew = old_s + size
25     cur_s = atomic_cas_state(s, old_s, new_s)
       if(cur_s != old_s)
           old_s = cur_s
28         goto join_slot
       end
       /* return my position within the group */
31     return {s, old_s-SLOT_READY}
   end
   def slot_close(s):
34   retry:
       s2 = slot_pool[pool_idx % POOL_SiZE];
       pool_idx = pool_idx+1
37     if(s2->state != SLOT_FREE)
           goto retry;
       end
40     /* new arrivals will no longer see s */
       s2->state = SLOT_OPEN
       slot_array[s->idx] = s2
43     old_s = atomic_swap_state(s, SLOT_PENDING)
       return old_s-SLOT_READY
   end
46 def slot_notify(s, lsn, group_size):
       s->lsn = lsn
       s->group_size = group_size
49     s->state = SLOT_DONE-group_size
   end
   def slot_wait(s):
52     while(s->state > SLOT_DONE)
           /* wait for notify */
       end
55     return {s->lsn, s->group_size}
   end
   def slot_release(s, size):
58     new_s = atomic_add_state(s, size)
       return new_s
   end
61 def slot_free(s):
       s->state = SLOT_FREE
   end
```

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.

Slot close operation (L33–45). 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 (L20), but incoming threads will almost never see this, instead using the fresh slot which the calling thread already swapped 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 (L46–56). After the slot owner acquires buffer space, it stores the base LSN and buffer address into the slot, then sets the slots 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 groups buffer space).

Slot release and free operations (L57–63). As each thread completes its buffer insert, it decrements the slots count by its contribution. The last thread to release will detect that the slot count became `DONE` and must free the slot; all other threads may leave immediately. The slot does not immediately become free, however, because the calling thread may still need to use it. This is particularly important for the delegated buffer release optimization described in Sect. A.6, because the to-be-freed slot becomes part of a queue to be processed by some other thread. Once the slot is truly 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.5 Lifecycle of a c-array slot

The pseudocode presented in the previous section makes heavy use of the *state* of a c-array slot, using it to encode several bits of information at different points in its lifetime. These include the status of the slot, the size of the group being formed, and whether the slot can be released or other threads still use it.

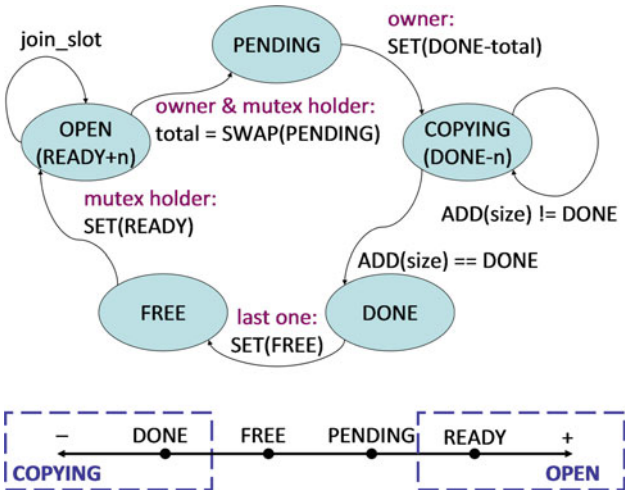

Fig. 17 Life cycle and state space of a c-array slot, to accompany Algorithm 4. The `OPEN` (`COPYING`) state covers all values at least as large (small) as `READY` (`DONE`)

Figure 17 summarizes the state space for an array slot. There are four major phases in the slot's lifetime: *open*, *pending*, *copying*, and *done*. The figure is comprised of two parts: the upper part shows the state machine and highlights the conditions that cause transitions to new states. The lower part depicts the space of integer values used to encode states compactly. The latter is important because the lock-free algorithms we use only allow us to manipulate a single machine word atomically.

Open phase. During the first phase, the slot is *open* and arriving threads build up a group. The state starts with the constant value `READY` (zero works well in practice), which indicates to incoming threads that no group has formed yet. As each thread joins the group, it atomically increments the state by the amount of buffer space it needs. The size of the group at any time is therefore `state-READY`. The open phase ends when the group leader (the first thread to arrive) succeeds in acquiring the log buffer mutex.

Pending phase. Once the group leader acquires the log mutex, it (non-atomically) allocates and assigns a new slot to the c-array, then atomically swaps the state's value with the constant value `PENDING` (any integer less than `READY` suffices). Incoming threads which still retain a reference to the slot treat `PENDING` as a signal to retrieve the new slot from the slot array and use it instead; all later incoming threads see the new slot first and use it directly. The group leader then extracts the group size by subtracting `READY` from the state value it swapped out, requests the needed log buffer space,

and stores the results (non-atomically) in other members of the slot object.

Copying phase. Having stored the requisite information in the slot object, the group leader notifies waiting threads by assigning `DONE-size` to the slot's state (where `DONE` is smaller than `PENDING`). Once the slot's state is less than or equal to `DONE`, other threads in the group can safely proceed to claim and fill their buffer space. As each thread completes the fill operation (group leader included), it again increments atomically the slot's state by the amount of buffer space it used; the thread which causes the slot's state to equal `DONE` is the last to finish and must release the slot.

Done phase. Once the last thread releases the slot and all other uses for that slot are known to be finished (e.g., serving as a qnode for delegated buffer release), the slot can be freed by assigning (non-atomically) the value `FREE` to it, where `DONE < FREE < PENDING`. At this point the slot can be reallocated by a group leader that transitions some other *open* slot to the *pending* phase, and the cycle repeats.

A.6 Delegated log buffer release and skew

Both the consolidation and decoupling optimizations discussed so far still require that all threads release their buffers in LSN order, which remains a potential source of delays. Many smaller insertions might execute entirely in the shadow of a large one and would then have to wait for the large insert to complete before releasing their buffer space. Further, buffer and log file wraparounds (which require special treatment and extra work at log flush time) prevent consolidation of buffer releases: consolidation would occur before acquiring the log mutex which allows the corner cases to be identified.

We remove this extra dependency between transactions by extending the decoupled buffer fill optimization so that the implied LSN queue becomes a physical data structure, as shown in Algorithm 5. 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.[7] 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 slower) 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 [32] and the critical section-combining approach by Oyama et al. [27].

To summarize the protocol, a thread with at least one predecessor attempts to change its queue node status atomically from waiting to delegated (L9, corresponding to the aborted state in Scotts work). On success, a predecessor will be responsible for the buffer release and the thread returns immediately. Otherwise, or if no predecessor exists, the thread releases its own buffer region and attempts to leave before its successor can delegate more work to it (L14). A successful `compare-and-swap` 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 (L15–18). 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 for its peers. This breaks long delegation chains (which are relatively rare) without impeding pipelining in the common case. As with Oyama's proposal [27], the grouping actually improves performance because a single thread does all the work without incurring the cache coherence misses that arise when multiple threads hand off to each other.

As an implementation detail, we note that the state variable `s` can double as a queue node when delegated buffer release is combined with consolidation array. In this case, `queue_handoff` is responsible for calling `slot_free` once the passed-in qnode is no longer visible to other threads.

Algorithm 5 Log insertion with delegated buffer release

```
   def buffer_acquire(size, data):
       /* wait for buffer space */
 3     lsn = /* update lsn and buffer state */
       qnode = queue_join(Q, lsn, size)
       lock_release(L)
 6     return qnode
   end
   def buffer_release(qnode):
 9     if (queue_delegate(Q, qnode) == DELEGATED)
           return /* someone else will release*/
       end
12   do_release:
       /* release qnode's buffer region */
       next = queue_handoff(Q, qnode)
15     if (next && is_delegated(next))
           qnode = next
           goto do_release
18     end
   end
```

[7] The signature of `buffer_release` function changes to reflect this fact.

References

1. Bouganim, L., Jónsson, B.T., Bonnet, P.: uFLIP: understanding flash IO patterns. In: CIDR'09: Fourth Biennial Conference on Innovative Data Systems Research, pp. 48–54. Asilomar, USA (2009)

2. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: USENIX Annual Technical Conference (2004)
3. Carey, M.J., DeWitt, D.J., Franklin, M.J., Hall, N.E., McAuliffe, M.L., Naughton, J.F., Schuh, D.T., Solomon, M.H., Tan, C.K., Tsatalos, O.G., White, S.J., Zwilling, M.J.: Shoring up persistent applications. In: Proceedings of the 1994 ACM SIGMOD international conference on management of data, Minneapolis, USA, pp. 383–394. ACM, New York (1994)
4. Chen, S.: Flashlogging: exploiting flash devices for synchronous logging performance. In: Proceedings of the 35th SIGMOD international conference on management of data, pp. 73–86. ACM, New York (2009)
5. Daniels, D.S., Spector, A.Z., Thompson, D.S.: Distributed logging for transaction processing. In: Proceedings of the 1987 ACM SIGMOD international conference on management of data, San Francisco, CA, USA, pp. 82–96. ACM, New York (1987)
6. Dewitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.I., Rasmussen, R.: The Gamma database machine project. IEEE Trans. Knowl. Data Eng. **2**(1), pp. 44–62. IEEE, Piscataway, NJ, USA (1990)
7. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation techniques for main memory database systems. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, Boston, MA, USA, pp. 1–8. ACM, New York (1984)
8. Gawlick, D., Kinkade, D.: Varieties of concurrency control in IMS/VS fast path. IEEE Database Eng. Bull. **8**(2), pp. 3–10. Washington, DC, USA (1985)
9. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proceedings of the 1996 ACM SIGMOD international conference on management of data, Boston, Montreal, Quebec, Canada, pp. 173–182. ACM, New York (1996)
10. Hardavellas, N., Pandis, I., Johnson, R.F., Mancheril, N., Ailamaki, A., Falsafi, B.: Database servers on chip multiprocessors: limitations and opportunities. In: CIDR'07: Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, pp. 79–87 (2007)
11. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, Vancouver, Canada, pp. 981–992. ACM, New York (2008)
12. Helland, P., Sammer, H., Lyon, J., Carr, R., Garrett, P., Reuter, A.: Group commit timers and high volume transaction systems. In: HPTS'87: 2nd International Workshop on High Performance Transaction Systems, Pacific Grove, CA, USA, pp. 301–329
13. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, Barcelona, Spain, pp. 206–215. ACM, New York (2004)
14. Johnson, R., Pandis, I., Ailamaki, A.: Improving OLTP scalability using speculative lock inheritance. PVLDB **2**(1), 479–489 (2009)
15. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, Saint Petersburg, Russia, pp. 24–35. ACM, New York (2009)
16. Johnson, R.F., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. PVLDB **3**(1–2), 681–692 (2010)
17. Lahiri, T., Srihari, V., Chan, W., MacNaughton, N., Chandrasekaran, S.: Cache fusion: extending shared-disk clusters with shared caches. In: Proceedings of the 27th International Conference on Very Large Data Bases, pp. 683–686. Morgan Kaufmann Publishers Inc., San Francisco (2001)
18. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
19. Lee, S.W., Moon, B., Park, C., Kim, J.M., Kim, S.W.: A case for flash memory SSD in enterprise database applications. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, Boston, Vancouver, Canada, pp. 1075–1086. ACM, New York (2008)
20. Lomet, D.: Recovery for shared disk systems using multiple redo logs. Technical report CRL-90-4, Digital Equipment Corporation, Cambridge Research Lab (1990)
21. Lomet, D., Anderson, R., Rengarajan, T.K., Spiro, P.: How the Rdb/VMS data sharing system became fast. Technical report CRL-92-4, Digital Equipment Corporation, Cambridge Research Lab (1992)
22. Mohan, C.: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In: Proceedings of the 16th International conference on very large data bases, pp. 392–405. Morgan Kaufmann Publishers Inc., San Francisco (1990)
23. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS **17**(1), 94–162 (1992)
24. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, Las Vegas, Nevada, USA, pp. 253–262. ACM, New York (2005)
25. Neuvonen, S., Wolski, A., Manner, M., Raatikka, V.: Telecom application transaction processing benchmark (TATP). See http://tatpbenchmark.sourceforge.net/
26. 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
27. Oyama, Y., Taura, K., Yonezawa, A.: Executing parallel programs with synchronization bottlenecks efficiently. In: PDSIA'99: International Workshop on parallel and distributed computing for symbolic and irregular applications, Sendai, Japan, pp. 182–204 (1999)
28. Pandis, I., Johnson, R.F., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB **3**(1–2), pp. 928–939 (2010)
29. Pandis, I., Tözün, P., Johnson, R., Ailamaki, A.: PLP: page latch-free shared-everything OLTP. Technical report, EPFL (2011)
30. 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
31. Rafii, A., DuBois, D.: Performance tradeoffs of group commit logging. In: CMG Conference (1989)
32. Scott, M.L.: Non-blocking timeout in scalable queue-based spin locks. In: Proceedings of the twenty-first annual symposium on principles of distributed computing, Monterey, California, pp. 31–40. ACM, New York (2002)
33. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks: preliminary version. In: Proceedings of the seventh annual ACM symposium on parallel algorithms and architectures, SPAA'95, Santa Barbara, CA, USA, pp. 54–63. ACM, New York (1995)
34. Soisalon-Soininen, E., Ylönen, T.: Partial strictness in two-phase locking. In: Proceedings of the 5th International Conference on Database Theory, pp. 139–147. Springer, London (1995)
35. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proceedings of the 33rd international conference on very large data bases, Vienna, Austria, pp. 1150–1160 (2007)

36. Thomson, A., Abadi, D.J.: The case for determinism in database systems. PVLDB **3**(1–2), 70–80 (2010)
37. TPC benchmark B standard specification, revision 2.0 (1994). Available at http://www.tpc.org/tpcb
38. TPC benchmark C (OLTP) standard specification, revision 5.9 (2007). Available at http://www.tpc.org/tpcc