The VLDB Journal (2014) 23:1–23
DOI 10.1007/s00778-013-0312-3

REGULAR PAPER

Eliminating unscalable communication in transaction processing

Ryan Johnson · Ippokratis Pandis · Anastasia Ailamaki

Received: 13 February 2012 / Revised: 12 March 2013 / Accepted: 15 March 2013 / Published online: 21 April 2013

Abstract Multicore hardware demands software parallelism. Transaction processing workloads typically exhibit high concurrency, and, thus, provide ample opportunities for parallel execution. Unfortunately, because of the characteristics of the application, transaction processing systems must moderate and coordinate communication between independent agents; since it is notoriously difficult to implement high performing transaction processing systems that incur no communication whatsoever. As a result, transaction processing systems cannot always convert abundant, even embarrassing, request-level parallelism into *execution parallelism* due to *communication* bottlenecks. Transaction processing system designers must therefore find ways to achieve scalability while still allowing communication to occur. To this end, we identify three forms of communication in the system—*unbounded*, *fixed*, and *cooperative*—and argue that only the first type poses a fundamental threat to scalability. The other two types tend not impose obstacles to scalability, though they may reduce single-thread performance. We argue that proper analysis of communication patterns in *any* software system is a powerful tool for improving the system's scalability. Then, we present and evaluate under a common framework techniques that attack significant sources of unbounded communication during transaction processing and sketch a solution for those that remain. The solutions we present affect fundamental services of any transaction processing engine, such as locking, logging, physical page accesses, and buffer pool frame accesses. They either reduce such communication through caching, downgrade it to a less-threatening type, or eliminate it completely through system design. We find that the later technique, revisiting the transaction processing architecture, is the most effective. The final design cuts unbounded communication by roughly an order of magnitude compared with the baseline, while exhibiting better scalability on multicore machines.

Keywords Scalable OLTP · Communication patterns · Shore-MT · SLI · Aether · DORA · PLP · Overlay Bufferpools

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

A. Ailamaki
School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, VD, Switzerland
e-mail: anastasia.ailamaki@epfl.ch

1 Introduction

Multicore hardware requires software to scale with exponentially increasing degrees of parallelism [35]. For the foreseeable future, processors are becoming more numerous rather than faster, leaving software largely responsible for converting Moore's Law into performance. This trend puts data management, in general, and transaction processing, in particular, in an interesting position: on the one hand, transactional workloads usually feature ample concurrency at the request level and even opportunities for parallelism within requests, but the high degree of hardware parallelism also exposes scalability bottlenecks throughout the system which must be addressed to achieve good performance [38].

The main source of scalability bottlenecks stems from the very nature of transaction processing applications, which

dictates concurrent access to a pool of shared application data and data structures. This characteristic typically requires independent agents in the system to coordinate or synchronize. The resulting excessive communication between concurrently active agents significantly reduces usable execution parallelism.

Transaction processing workloads, such as those arising in banking, telecommunications, and commerce, are particularly challenging from a scalability perspective. They are update intensive, highly concurrent (often with conflicting requests), and usually touch too little data to allow significant data-flow parallelism within requests. The database engine must serve these requests with low response times, while maintaining isolation, consistency, and durability of the data (the so-called *ACID* properties [27]). Maintaining ACID properties in the face of arbitrarily overlapping requests is a large contributor to the complexity of transaction processing systems (e.g., by measured instruction counts [29]). Most importantly, ACID underlies most of the communication responsible for scalability bottlenecks.

Some systems drop or *relax* their provided consistency guarantees or the provided functionality, such as the ability to perform efficient joins, to reduce the amount of communication needed for their operation. For example, recent years have seen rising popularity of key-value stores that provide strong atomicity only for individual records [22], as well as systems offering *eventual consistency* guarantees [71] in place of ACID. Relaxing the consistency requirements or reducing the provided functionality, however, is unacceptable for a large class of applications.

Thus, although some workloads exhibiting so-called *embarrassing request-level parallelism* can be architected to eliminate virtually all communication between requests, in the general case the design of transaction processing systems is more challenging. Workloads amenable to distributed *shared-nothing* processing techniques [23,66] avoid such issues, but shared-nothing designs impact the efficiency of certain database functionality, fail to exploit fast communication via shared caches, and impose large resource footprints that tend to interact poorly with shared-memory hierarchies. Shared-everything designs, on the other hand, encourage communication but require some sort of *concurrency control* mechanism to maintain *consistency*. Existing concurrency control techniques approach the problem in different ways (optimistic, pessimistic, etc.), but they all manage contention rather than directly addressing the underlying communication patterns that cause bottlenecks in the first place. This is corroborated in Sect. 2, where we show that a set of modern database engines fail to achieve even close to optimal scalability when serving a perfectly scalable transactional application in a modern highly parallel multicore machine. We therefore focus on changing the underlying communication patterns of transaction processing systems rather than any particular contention management discipline.

Because eliminating all communication within transaction processing systems usually sacrifices important features, system designers must find ways to achieve scalability while still allowing some communication. To guide this search, we break communication patterns into three types: *unbounded*, *fixed*, and *cooperative*. As we argue in Sect. 3 and give examples in Sect. 4, unbounded communication is the main threat to scalability. The other two types both limit the growth of contention, so system designers should focus on eliminating unbounded communication. We have found that addressing harmful communication patterns directly is a powerful tool for improving database engine scalability, and we argue that it can improve the scalability of any software system. To prove our point, we present techniques developed as part of the Shore-MT project [38] and relate them back to the communication categorization and handling of unbounded communication.

There are three ways to handle unbounded communication: The designer can try to *downgrade* unbounded communication to either the fixed or cooperative types, which are less problematic; or *avoid* unbounded communication, for example through caching or approximation. As a drastic measure, *re-architecting* parts of the system can expose communication patterns and completely eliminate the execution of unbounded communication codepaths. In Sect. 5, we present two use cases of improving the scalability of a modern database engine by following the two aforementioned techniques that do not involve re-architecting the system. In particular, in Sect. 5.1, we show how we can convert the serial insert of log entries in the log buffer to use cooperative communication. The log buffer example is very important because it converts contention due to simultaneous log insertions into a *constructive* pattern that actually reduces the amount of work to be performed. The alternative, distributing requests among multiple independent log files, imposes high complexity and non-negligible (or even prohibitive) overheads. Next, in Sect. 5.2, we present Speculative Lock Inheritance, which eliminates unbounded communication in the lock manager by selectively passing information directly between transactions executed by the same agent.

Even when we carefully implement a conventional transaction processing system—employing techniques that avoid or downgrade unbounded communication—the system's code paths are still littered with points of unbounded communication. In Sect. 6, we argue that conventional transaction processing execution design suffers a fundamental scalability limitation due to the prevailing request- or transaction-oriented policy of assigning work to threads in the system. To overcome the limitations of conventional execution, we propose a different execution paradigm, called data-oriented. We demonstrate that a data-oriented approach to transaction

processing combines the best properties of shared-everything and shared-nothing designs and eliminates the majority of unbounded communication in the transaction processing system. The changes, while more disruptive than the others presented in this paper, are nowhere near a complete rewrite, but allow a traditional engine to achieve significantly higher scalability.

Re-architecting the system according to the data-oriented model offers significant potential. In Sect. 7, we extend the idea of logical partitioning to tackle other remaining sources of unscalable communication. In particular, we effectively eliminate the communication due to page latching and buffer pool management, with physiological partitioning (PLP) (Sect. 7.1) and overlay buffer pools (Sect. 7.2) correspondingly. PLP eliminates the need for page latching by aligning the physical data layout with the logical partitioning of a data-oriented engine. In order to achieve that, PLP uses a multirooted index structure, with each sub-tree belonging to a single logical partition. The overlay buffer pool is an adaptive cache that transfers the ownership of subsets of frames in the buffer pool to specific agents to improve locality of access, aligned with the PLP. In cases where the working data set of the transactional application fits in main memory (the common case if we consider the ample and ever-growing main memories), the overlay buffer pool mechanism eliminates unbounded communication from the buffer pool manager.

1.1 Contribution and organization

In this paper, we make the case that communication, *per se*, does not automatically diminish scalability in the context of transaction processing. Instead, we argue that eliminating unbounded communication addresses most bottlenecks in the system. Together, the approaches we present in this paper show that "mainstream" shared-everything transaction processing systems can become significantly more scalable than is traditionally assumed possible. The lessons learned and the techniques we present in the following sections can be useful for improving the scalability of a variety of software systems. Due to lesser unbounded communication, the final design of this work executes an order of magnitude fewer unscalable critical sections and achieves up to 6× higher throughput than the multicore-optimized transaction processing system it extends. We make a comparison of the performance of the various techniques under a common framework, using both in-order and out-of-order multicore machines.

The rest of the document is structured as follows. In Sect. 2, we discuss the trade-offs between shared-nothing and shared-everything architectures and show how well modern shared-everything database engines scale. In Sect. 3, we categorize the various types of communication patterns and profile the patterns in a modern database engine. In Sect. 4, we present the negative effects by the unscalable communication of crucial components of a modern database engine. In Sect. 5, we give two concrete examples of downgrading or side-stepping unbounded communication in the log and the lock managers, two essential components of traditional transaction processing. In Sect. 6, we present an alternative transaction processing design based on data-oriented execution and show that systems based on this design break the communication-related limitation of mainstream systems. In Sect. 7, we extend the idea of logical partitioning to tackle other significant sources of unscalable communication; namely, the page latches and the buffer pool communication. In Sect. 8, we compare the presented techniques under the same framework. Finally Sect. 9 presents related work and Sect. 10 concludes.

2 Scalability in transaction processing

When discussing "scalability" in the context of transaction processing application stacks, overall scalability is limited by the worst behaving of (a) the workload, (b) the hardware, and (c) the database engine.

Workload. First, application or workload scalability: do the requests generated by the workload frequently conflict, are they logically independent, or something in between? A transactional workload whose updates frequently conflict exhibits very low concurrency, because those updates must serialize to maintain correctness. For example, the `Delivery` transaction from the `TPC-C` benchmark is fundamentally unscalable: Each transaction attempts to delete the "oldest" record in the database, and the possibility of aborts requires all such transactions to serialize. In practice, most database applications are highly scalable and requests seldom conflict; ordering items from an online retailer or booking seats on a particular flight will not usually lead to conflicts, although the system must still handle those conflicts properly when they do arise.

Hardware. At the bottom of the system stack, scalable hardware refers to a platform supporting a high degree of parallelism. The uni-processors of yesteryear are not scalable in this sense, nor would be a machine lacking memory capacity or disk bandwidth to balance its parallel processing power. Today's multicore machines and data center infrastructure are generally quite scalable, making software scalability more important than ever. We assume in this paper that hardware does not significantly limit system scalability.

Database engine. We are most concerned with a third form of scalability: Given a scalable transactional workload above and scalable hardware below, does the transaction processing system, which sits in between, deliver scalable performance?

2.1 Shared-nothing versus shared-everything

An important consideration when designing a transaction processing system is whether its target workloads can be implemented in a *shared-nothing* fashion [23,66]. Shared-nothing applications assign multiple tasks near-independent operations and dedicated resources; any tasks that do communicate expect high latencies. Many important workloads provide the near-independent tasks needed for this model and exhibit *embarrassing execution parallelism*: They can utilize exponential increases in parallelism with relatively little effort. Examples of shared-nothing designs include distributed databases offering relaxed consistency [71], key-value stores [14,22], Map/Reduce frameworks [21] and —to a lesser degree—transaction processing engines like H-Store/VoltDB [68], and HyPer [42].

In the general case, however, shared-nothing designs are non-ideal: First, they give up a key functionality of the database engine, namely the ability to broker communication between agents that connect to the database. The need for this feature prevents VoltDB's design from being truly shared-nothing, for example, and internode communication reduces its scalability significantly if a suitable partitioning of data cannot be found [18,40]. Furthermore, a large class of enterprise and other applications cannot tolerate compromised consistency, and ACID is thus either required or strongly preferable as long as the system delivers adequate performance. Second, shared-nothing designs do not exploit low communication latencies when the latter are available. Instead, they assume that the underlying inter-process interconnect is some slow medium. Third, the data footprint grows linearly with the number of tasks in the system; each task works on an independent data set, perhaps replicating frequently used read-only data, and thus utilizes poorly the shared memory [16]. In cases where tighter coordination is required, *shared-everything* designs allow tasks to communicate freely and with much lower latency than network connections offer.

In contrast to shared-nothing, shared-everything designs depend on fast communication that allows tasks or threads of execution to access and modify the same data concurrently. However, the risk of conflicting accesses requires some sort of *concurrency control* mechanism to maintain the *consistency* (integrity) of the data and to prevent *races*, or timing bugs that arise when code does not enforce proper consistency. Various concurrency control mechanisms exist to address these difficulties, including mutex locks [51,52], transactional memory [34], and lock-free algorithms [33]. Higher-level design approaches such as message-passing [9] or event-based architectures [72] can also implicitly provide concurrency control. However, in all cases, communication bottlenecks prevent a subset of worker threads from making forward progress. With mutex locks, threads detect contention and serialize before beginning an operation (a pessimistic approach), while transactional memory and lock-free algorithms are usually more optimistic, in the sense that threads always attempt to proceed but may abort and retry their work if contention arises. Bottlenecks in message-passing approaches typically manifest as systematic load imbalances (e.g., tasks whose work queues are always full or empty).

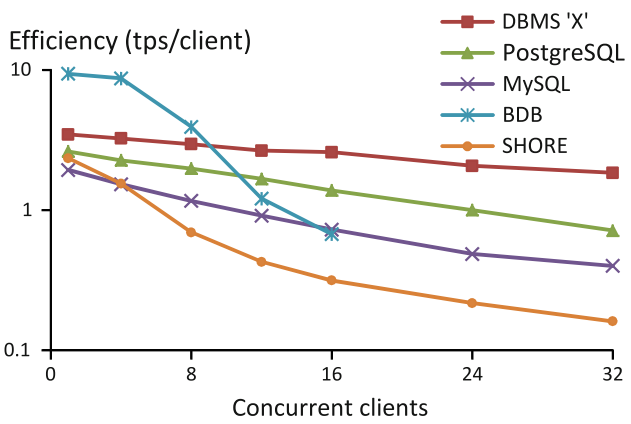

Fig. 1 Efficiency of several database engines executing a scalable transactional workload. Ideally, per-thread throughput remains steady as more threads join the system

While the various mechanisms have different strengths and weaknesses, they mostly provide different ways to manage contention, without addressing the underlying *communication patterns* that force serialized execution and cause bottlenecks. We should note that message passing arguably makes this task easier by forcing the programmer to identify communication patterns explicitly. On the other hand, transactional memory may make it harder by favoring designs where all communication is implicit.

In this paper, we focus on changing the underlying communication patterns rather than any particular contention management discipline. Many of the examples in this paper are based on mutex-based concurrency control, which is the dominant mechanism employed by today's transaction processing systems. But our solutions often incorporate work in lock-free algorithms (e.g., Sect. 5.1), as well as event-based and message-passing disciplines (e.g., Sect. 6).

2.2 How well modern shared-everything systems scale

In previous work, we observed that database systems designed for the hardware constraints of the past (limited hardware parallelism and highly constrained memory environments) are unable to utilize effectively the abundant processing power and ample main memories of today's hardware [38]. Figure 1 summarizes the situation ca. 2009. We ran a microbenchmark, a simple scalable transactional workload, on several popular open source database engines

(MySQL, BerkeleyDB, PostgreSQL), a research prototype (SHORE [13]), and a commercial system (labeled DBMS "X"). Each client creates a private table and inserts several thousand records into it. In theory, there should be no contention whatsoever because the transactions are completely unrelated, each acting on a disjoint subset of the database. We plot on the y-axis the scalability (given as per-thread performance) as the x-axis varies the number of worker threads available in a SPARC Niagara machine with 32 "lean" cores [28]. A perfectly scalable system would have unvarying per-thread efficiency regardless of the number of running threads, a goal none of the mainstream open source database engines achieves. Even the representative commercial engine, DBMS "X," shows sub-optimal, though superior, scalability.[1]

Somewhat counter intuitively, the observed poor scalability arises because of seemingly "small" bottlenecks. Gene Amdahl codified this paradox with his famous law [5], and Hill and Marty analyzed in detail its implications for multicore systems [35]. Amdahl's law states that, given available hardware parallelism P, the degree of parallelism a system can actually exploit (P_{eff}, or "scale up"), is sharply constrained by the fraction of work, s, which it performs serially:

$$P_{\text{eff}} \leq \frac{1}{s-(1-s)\frac{1}{P}} \approx \frac{\text{TotalWork}}{|\text{CriticalPath}|}, \quad P \to \infty$$

Put another way, even a "small" serial fraction of work (or need for communication between different agents in the system) has a large opportunity cost with respect to scalable parallelism: To maintain utilization, the serial fraction of work must halve for every doubling of available hardware parallelism. Because it determines overall scalability of the system, the serial component of work is known as the *critical path*, and communication patterns largely determine the length of the critical path. In an era of significant hardware parallelism, efforts to improve performance cannot ignore the underlying communication patterns in the system, sometimes with non-intuitive implications.

2.3 Shore-MT: a reliable baseline

As Fig. 1 shows, none of the available open source database engines scales effectively on highly parallel multicore hardware; we set out to create one of our own based on the SHORE storage manager [13]. We selected SHORE for two reasons. First, it supports all the major features of modern database engines: full transaction isolation, hierarchical and row-level locking [27], a CLOCK buffer pool with replacement and prefetch hints [63], KVL-Tree indexes and ARIES-style log-

[1] The situation has improved markedly since 2009. Developers of the various engines have focused on improving their scalability, sometimes reporting to the authors that they used techniques summarized in this paper.

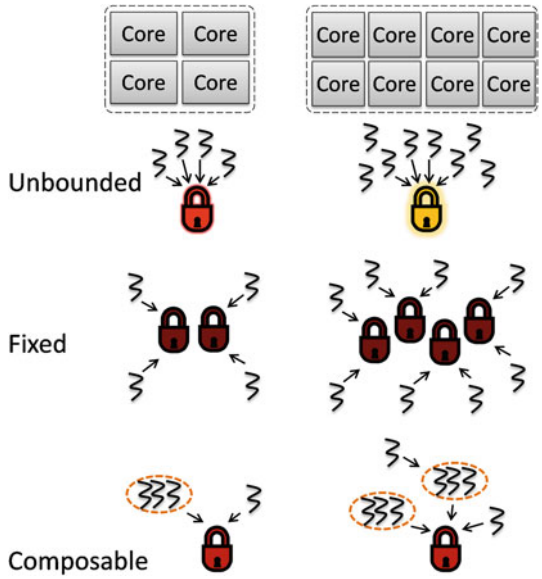

Fig. 2 We classify the communication into three patterns: *unbounded*, *fixed* and *cooperative*. Only the unbounded communication poses scalability problems, the other two mostly cause overheads for single-threaded execution

ging and recovery [53,54]. Additionally, SHORE has previously shown to behave like commercial engines at the instruction level [2], making it a reasonable open source proxy. This exercise resulted in *Shore-MT*, which scales far better than its open source peers while also maintaining excellent single-thread performance [38]. For the rest of this paper, we use Shore-MT both as our baseline for comparison as well as the common codebase where we develop the presented mechanisms.

3 Communication in a database engine

We have just discussed how communication patterns usually require some form of synchronization and serial execution that eventually limits scalability. Shared-nothing approaches sidestep the issue by disavowing nearly all communication and forbidding any form of tight coupling, while shared-everything systems suffer from communication bottlenecks that limit their scalability. Because the transaction processing system cannot always eliminate communication without giving up important features, we must find ways to achieve scalability while still allowing some communication. In order to guide this search, we break communication patterns into three types: *unbounded*, *fixed*, and *cooperative*. We illustrate the three patterns in Fig. 2 and briefly describe them in the following paragraphs.

Unbounded communication. This type of pattern (Fig. 2, left) arises when the number of threads involved with a point of communication is roughly proportional to the degree of parallelism in the system. No matter how efficient or infre-

quent the communication, exponentially increasing parallelism will eventually expose it as a bottleneck. Globally shared data structures, which multiple agents update concurrently, fall directly into this category. Used carelessly, unbounded communication can easily dominate execution.

Fixed communication. At the other extreme of the spectrum, fixed communication patterns (Fig. 2, middle) involve a constant or near-constant number of threads regardless of the degree of parallelism. The pattern itself limits the amount of contention that can arise. Grid-based simulations in scientific computing (including several from the SPLASH-2 benchmark suite [73]) exemplify this type of communication, with each simulated object communicating only with its nearest neighbors in the grid. Peer-to-peer networks (e.g., [65]) also employ near-fixed communication patterns. Producer–consumer patterns, frequently arising in data management applications, also exhibit fixed communication.

Cooperative communication. A third kind of communication pattern, which we call *cooperative*, arises when threads work together to reduce contention while waiting to access a shared resource. A canonical example of cooperative communication arises with a parallel LIFO queue: Pairs of push and pop requests that encounter each other in flight can cooperate, *eliminating* themselves directly without further need to compete for the underlying data structure [55]. As the right part of Fig. 2 shows, such communication is self-moderating: contention increases as more threads communicate, allowing more opportunities for cooperation that in turn reduces contention. Other examples include combining trees and other distributed algorithms.

Examining these three types of communication suggests that *unbounded* communication is the main threat to scalability. The other two types avoid unbounded contention and increased parallelism usually compensates for any loss of single-thread performance.

On the other hand, the *impact* of communication on the performance and scalability of a system depends on the communication pattern (i.e., how many threads participate) and also on the communication's duration. Critical sections implement most communication in shared-memory systems, so the length of each critical section impacts performance strongly. Many types of unbounded communication (or critical sections) are short in comparison with others, and they do not appear as bottlenecks when profiling the performance under low parallelism. They are difficult to detect, let alone to prevent, but the contention they inevitably trigger poses a serious threat to scalability. As code bases mature and measure millions of lines of code (like all the modern database engines) it is increasingly difficult to go back and fix such lurking problems. It is tempting to improve the performance of the system by attacking the longer critical sections, regardless of the types of communication they correspond to; though this approach has short-term benefits, bottlenecks usually recur. We argue that longer-term scalability requires a focus on eliminating unbounded communication.

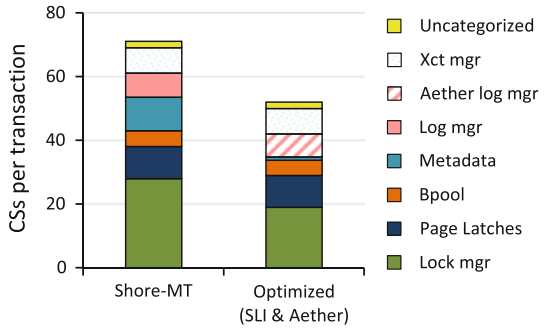

Fig. 3 Number and type of critical sections executed by a simple transaction, `TATP UpdateLocation`, for baseline and enhanced versions of Shore-MT. *Bars* with *solid fill mark* un-scalable critical sections. Speculative lock inheritance (SLI) and cooperative log buffer inserts (Aether) eliminate a large number of un-scalable critical sections

3.1 Critical sections in Shore-MT

We use communication points (critical sections), particularly unbounded ones, to identify lurking scalability bottlenecks in the system. Figure 3 shows the average number of critical sections entered by two variants of Shore-MT for a very simple transaction: the `UpdLocation` transaction of the `TATP` benchmark increments only one field of one record from a single table. We categorize critical sections based on the component or service of the transaction processing system responsible (e.g., lock manager, log manager, etc...). The dark-colored solid bars indicate unbounded communication, the light-colored solid bars identify fixed patterns, and the striped bars show cooperative communication. In order to get the breakdown, we execute the transaction 1,000 times and instrument each critical section code.

The left-most bar of Fig. 3 shows the critical sections entered by the public version of Shore-MT, our "conventional" baseline. Even for this minimal transaction, the baseline system enters approximately 70 critical sections, with roughly 60 of those being un-scalable. The lock manager is responsible for almost half of the unbounded critical sections, followed by the log manager, the metadata manager, the page latches, and the buffer pool manager. These bottlenecks are immediately exposed when we move to more parallel or powerful hardware: The second generation Niagara chip contains 64 hardware contexts, double the number of cores in Fig. 1, and ×86_64 cores, while less numerous, are highly sophisticated and run at higher frequency.

4 Impact of unbounded communication

So far, we have tallied communication points or critical sections, and Fig. 3 suggests that logging, locking, buffer pool

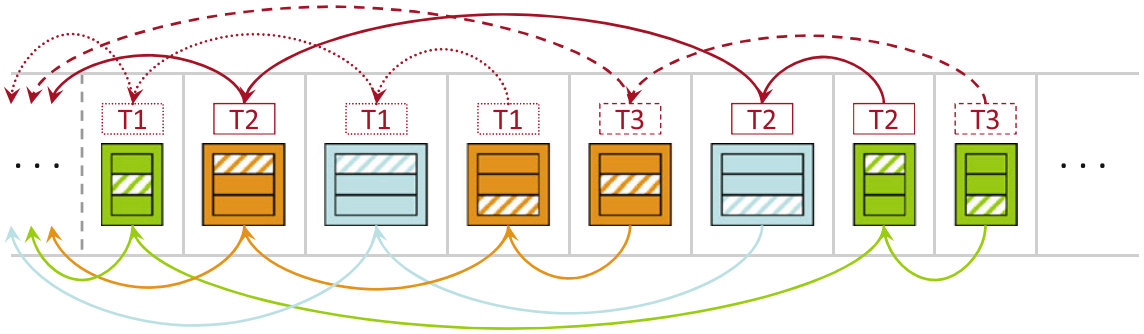

Fig. 4 LIFO-ordered linked lists connect each log record with successively older modifications made by the same transaction or to the same page. The resulting partial order must be honored or log recovery fails

management, and page latching all pose threats. Each provides crucial functionality, but relies on central communication. In this section, we examine the performance impact of those bottlenecks.

4.1 Database logging: serial by design?

Logging is a bastion of centralized communication in the transaction processing system. Most implementations use some variant of ARIES [53,54], a write-ahead logging (WAL) scheme which guarantees recoverability in spite of repeated failures, as well as moving a significant fraction of disk I/O off the critical path through asynchronous page cleaning. Unfortunately, WAL requires every update transaction in the system to communicate with the log manager at least twice: one or more times to write-ahead changes, and a final time to request a commit.

ARIES centers around a globally consistent "history of the world" recorded in the database log. The log record associated with each update receives a unique log sequence number (LSN) that serves as a timestamp for the update within this global history. The LSN also encodes a record's location on disk, identifies data pages as dirty or clean, and serves as a checksum of sorts for 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. Once a transaction logs some change it intends to make, it can make the actual change in memory, without having to write it back to disk immediately: In the event of a crash, log replay will regenerate any missing updates and allow the system to recover.

In order to keep the database consistent in spite of repeated failures, ARIES imposes strict constraints on log record ordering. As indicated in Fig. 4, log records form two sets of linked lists such that each record identifies the most recent previous modification to the same page as well as the most recent update from the same transaction. 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. The biggest difficulty lies in avoiding "holes" in the log: Any transaction must ensure that all log entries for pages it touched are durable before it can complete or risk making the database unrecoverable. A naive implementation would be forced to flush all logs at every transaction commit; tracking dependencies between log records more carefully has quadratic complexity in the number of active transactions [64]. These characteristics thus discourage the use of distributed logging mechanisms to improve scalability.[2]

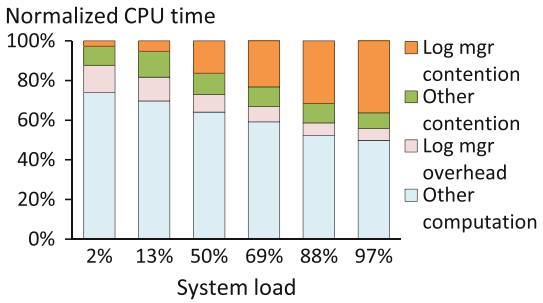

Fig. 5 Log manager overhead and contention for the update transactions of the TATP benchmark as load increases. Contention consumes CPU time without increasing performance

Though itself serial, ARIES enables higher concurrency elsewhere in the system by allowing transactions to release latches immediately after logging the updates, rather than holding them until the corresponding dirty pages are written out to disk. Unfortunately, as parallelism continues to increase, this centralized communication quickly becomes a scalability bottleneck.

Contention within the log manager can become a significant and growing fraction of total execution time, especially in update-intensive workloads. Figure 5 shows the break-

[2] Distributed transaction processing systems use distributed logging to avoid unnecessary partition crossings and for fault tolerance purposes, e.g., [19,49], but scale poorly when transactions span multiple nodes. Many distributed systems, including Rdb/VMS [50], actually maintain a single shared log, usually at a dedicated node.

down of a modern database engine when we increase the load of the system as it serves the update transactions from the TATP benchmark in a single-socket UltraSPARC Niagara II server with 64 hardware contexts. To isolate the impact of log-related bottlenecks, the evaluated system is baseline Shore-MT equipped with a mechanism for avoiding lock contention described shortly (Sect. 5.2). In Fig. 5, the x-axis varies load on the system from very light (left) to very heavy (right), while the y-axis shows the breakdown of CPU time each transaction spends in the log manager (not counting time spent idling, for example blocked on I/O or due to lock conflicts). This figure, and those that follow, defines overhead and contention as the useful and useless work, respectively, performed by the system while processing transactions. In the workload with TATP's update transactions, we see that useful log-related work remains a small fraction of execution time as the system load increases, but contention in the log manager increases with load, accounting for more than 35 % of the execution time on a saturated machine.

Some transaction processing systems dispense with logging altogether, relying on large main memories to avoid I/O and using replication strategies to prevent, rather than recover from, system outages [42,68]. This strategy works well when a shared-nothing is feasible, but does not help in a shared-everything environment. We also note that replication poses its own, very large, set of challenges [26,69]. We return to the issue of logging in Sect. 5.1.

4.2 Bottlenecks in the database lock manager

Logical locking is a particularly significant source of overhead and complexity in a transaction processing system, especially in transactional workloads consisting of short transactions common in telecommunication, banking, and sales/retail, that cause heavy use of core database services. Even on a single-core server with no contention for lock data structures, locking accounts for 16–25 % of work in TPC-C transactions [29].

Virtually all transaction processing systems use some form of hierarchical locking [27] to allow applications to trade off concurrency and overhead. For example, requests accessing large amounts of data can acquire coarse-grained locks to reduce overhead at the risk of reduced concurrency. At the same time, small requests can use precise locks to maximize concurrency with respect to other independent requests. The lock hierarchy is crucial for application scalability because it makes fine-grained logical concurrency control efficient.

Ironically, however, hierarchical database locking causes a new scalability problem while addressing the first one: Transactions must acquire intention locks in the upper levels of the hierarchy before accessing individual objects at lower levels. All agent threads communicate with the centralized lock manager repeatedly to access the same intention locks.

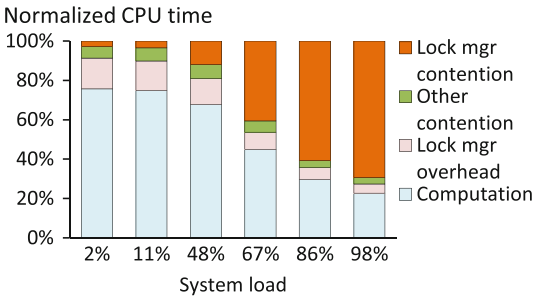

Fig. 6 Lock manager overhead as system load varies. The unbounded communication in the lock manager quickly becomes the bottleneck. The corresponding contention consumes CPU time without increasing performance

Unfortunately, lock management overhead linear in the number of holders, so intention locks suffer both increased request load and longer processing times per request. This is a form of unbounded communication according to our categorization, since on a bigger machine with a larger number of active agents, a larger number of agents would need to communicate with the lock manger for the same locks. This excessive communication for accessing the same data structures (the database locks) causes physical contention in the system that penalizes throughput.

Until recently, single-node database engines typically time-shared all requests on a small number of processors. Thus, the negative impact of unbounded communication was not noticeable. However, as core counts continue to double each processor generation, increased hardware concurrency leads to bottlenecks in the centralized lock manager, especially when hierarchical locking forces many threads to update repeatedly the state of a few hot locks.

The unbounded communication causes locking-related bottlenecks even for scalable transactional workloads having few logical conflicts. Because this is inherent behavior of hierarchical locking, we expect that every system will eventually encounter this kind of problem within the lock manager, if it has not done so already. Figure 6 highlights how contention for database locks impacts performance as we increase load in the baseline Shore-MT system running the TATP benchmark in the single-socket UltraSPARC Niagara II machine with 64 hardware contexts. Similarly with Fig. 5, the x-axis in Fig. 6 varies load on the system from light to heavy, while the y-axis shows the fraction of CPU time each transaction spends in the lock manager.

We make two observations from Fig. 6. First, the useful work due to the lock manager is a small part of the total. Second, nearly all of the rapidly growing contention in the system arises within the lock manager, eventually accounting for nearly 75 % of the transaction's CPU time. We return to this issue in Sect. 5.2 and Sect. 6, two different approaches to eliminate contention in the lock manager. As a side effect,

these approaches eliminate contention in the metadata manager (see Sect. 5.2.1). As mentioned in Sect. 4.1, the lock contention we are concerned with here manifests well before log manager contention—recorded in the "other" category—becomes noticeable.

4.3 Page latching and buffer pool management

According to the communication breakdown shown in Fig. 1, two other components that are significant contributors of unbounded communication are the page latching and the buffer pool manager.

Page latching. Latches are used heavily inside transaction processing systems. Invisible at the transactional level, they are held only briefly to assure physical consistency of data and resources. Whenever an agent reads or writes the contents of a database page it must latch and unlatch the page and frequently accessed pages may become points of contention. Upper levels of B+Tree indexes, popular records, and false sharing (unrelated data stored on the same page) can all become hotspots; solutions typically introduce padding that forces problematic records to separate pages. However, it is not always easy to detect such pathological situations, and they increase database tuning and administration costs. Finally, we note that the sheer number of page latches acquired imposes unwanted overhead and contention, even though page latching is inexpensive compared to acquiring a database lock.

Buffer pool management. The buffer pool provides the rest of the system the illusion that the entire database resides in main memory, similar to an operating system's virtual memory manager. When a transaction requests a database page not currently in memory, it must wait while the buffer pool manager fetches it from disk. The buffer pool contains a fixed number of "frames," each of which can hold one page of data from disk. If no frame is available for the newly fetched page, the buffer pool evicts another page following a replacement policy, such as Least-Recently Used or CLOCK [63], attempting to minimize impact on other requests (e.g., by evicting a hot page). Transactions "pin" in-use pages in the buffer pool to prevent them from being evicted and "unpin" them when finished. The buffer pool manager and log manager are also responsible to ensure that modified pages are flushed to disk (preferably in the background) so that changes to in-memory data become durable.

The updates to the buffer pool frames result in unbounded communication. To quickly find any database page requested, buffer pools are typically implemented as large hash tables. Operations within the hash table must be protected from concurrent structural changes caused by evictions, usually with per-bucket latches. Hash collisions and hot pages can cause contention among threads for the hash buckets.

The impact of the unbounded communication due to page latching and buffer pool operations is not as severe as those due to logging and locking; however, this unbounded communication remains a lurking threat to scalability. In Sect. 7, we return to these issues.

5 Attacking unbounded communication

If a workload is not amenable to a shared-nothing arrangement, the system designer needs to minimize the impact of communication. In particular, we focus on three ways to reduce the frequency of unbounded critical sections in the system and improve its scalability:

- **Downgrade**. Design changes that convert unbounded communication to either fixed or cooperative forms can move communication off the critical path.
- **Avoid**. Targeted changes to the system can sometimes avoid unbounded communication, for example by judicious use of caching.
- **Re-architect**. When other measures fail, it may be necessary to re-architect (parts of) the system to facilitate replacing contention-prone algorithms code paths with scalable ones. The goal of re-architect it so make the communication patterns explicit, so that they are exposed and easier to deal with.

The rest of this section illustrates the first two points in the context of log and lock management, respectively; the next section touches on re-architecting the system.

5.1 Downgrade unbounded communication with cooperative log buffer inserts

In this section, we highlight our recent work [39] addressing contention within the log manager by converting the underlying unscalable communication using a cooperative technique we call "consolidation." The technique, inspired by work from the parallel algorithms community [55], allows competing threads to collaborate and reduce overall demand on shared resources.

5.1.1 Consolidation

The log record associated with each update receives a unique log sequence number (LSN) that identifies the update's location in this global history and serves several important purposes throughout the system. A typical log insert requires latching the log buffer, generating an LSN, copying the associated data into the buffer (along with links to relevant previous log records), and releasing the latch. Because of its serial nature, LSN generation and the accompanying log inserts

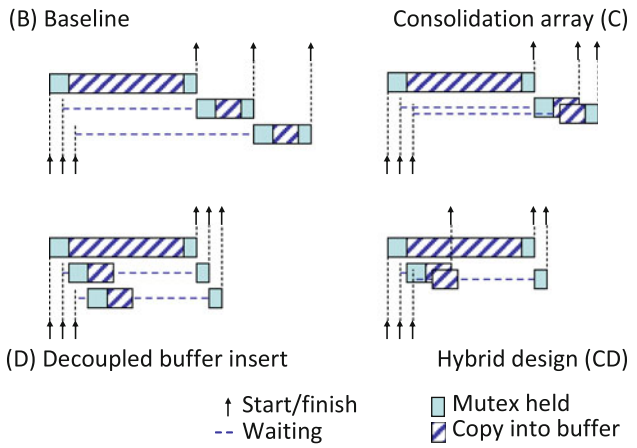

Fig. 7 Comparison of four log buffer design approaches

limit parallelism in the system. While it is possible to shorten the critical path by breaking LSN generation and data movement into two stages, the benefits are limited because LSN generation is very fast and the two operations must start and complete in the same order to prevent "holes" in the log.

We attack the problem log buffer contention at its root, developing techniques which allow LSN generation to proceed in parallel while still yielding a globally consistent history. We achieve parallelism by adapting the concept of "elimination" [55] so that the system can generate sequence numbers, and manage data copying, for multiple log requests simultaneously. We call this behavior "consolidation" for reasons which will become clear in the paragraphs that follow. A desirable effect of consolidation is that high loads actually reduce demand on the log buffer latch, rather than increasing it.

The LIFO stack provides perhaps the easiest way to visualize the concept of elimination which underlies consolidation: Consider a large number of requests to both `push` and `pop` items from the stack. A naive implementation would require every request to acquire a latch and insert or remove an item from the end of the queue. However, if we can identify a pair of requests, one to `push` and one to `pop`, the two can complete by exchanging information directly, rather than contending for the centralized data structure. Requests that *eliminate* each other in this fashion not only avoid waiting for a contended latch themselves, but also reduce the wait times for other requests that remain.

In the case of log records, pairs of requests cannot eliminate each other because the data from both must end up in the log. However, groups of threads can create a single, larger, request at the log, carving up the resulting LSN, and buffer space among themselves without consulting other threads in the system. Figure 7C illustrates the benefits of this approach: Threads waiting for a slow log insertion can combine their requests, presenting the log buffer with a single one.

An orthogonal optimization—decoupled or pipelined insertion—is also shown in the figure (labeled "D"). Decoupled insertion allows threads to overlap the data copy operations. Though requests must complete in order, a further optimization (not shown) allows threads to delegate completion to a (slow) predecessor that would cause them to wait—the latter then becomes responsible to complete all requests which accumulated during the delay. The experiment in Sect. 8.5 shows that the best performance is achieved by a combination of the two approaches ("CD" in the figure), which moves both large requests and bursts of requests off the critical path for maximum robustness.

5.2 Sidestepping unbounded communication with speculative lock inheritance

We have seen already (Sect. 4.2) how the centralized lock manager generates significant unbounded communication, and the scalability bottleneck that results—ironically—when threads repeatedly request compatible modes for the same locks.

The key to reducing contention within the lock manager comes by observing that virtually all transactions request high-level locks in compatible modes; even updates to rows or pages in the database generate compatible intent locks higher up, and transactions requiring coarse-grained exclusive access are extremely rare in scalable workloads. Further, in the absence of intervening updates, it makes no semantic difference whether a shared-mode lock is released and reacquired or simply held continuously. Either way, a transaction will see the same (unchanged) object, and other transactions are free to interleave their reads to the object as well.

Speculative Lock Inheritance exploits these characteristics of shared-mode locks to transparently and safely allow committing transactions to pass some of their locks to those which follow, instead of releasing and reacquiring them. The technique is inspired by early distributed databases that used local lock caches to avoid expensive network traffic [50]. Where the older systems cached only exclusive locks,[3] however, we are interested only in shared ones. Focusing on the hottest shared locks in the system alleviates contention because most transactions acquire those locks directly from their predecessor instead of making requests to the centralized lock manager. For our test system with 64 hardware contexts running a variety of short transactions, we find that SLI holds contention in the lock manager to small, near-constant levels even as hardware concurrency increases.

Figure 8 highlights SLI in action: an agent thread executing transactions detects a high degree of physical con-

[3] In a well-partitioned system, exclusive locks seldom move between nodes, and caching them is extremely effective.

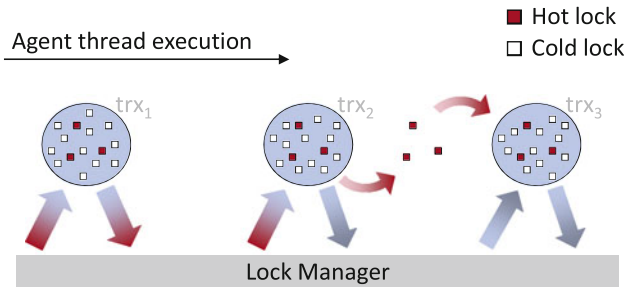

Fig. 8 Agent threads which detect contention at the lock manager retain hot locks beyond transaction commit, passing them directly to the transactions which follow

tention with certain hot locks while executing trx_1 and trx_2, so it retains the offending locks when trx_2 commits, rather than attempting to release them. When the agent begins execution of trx_3, the locks are already available without further involvement from the lock manager. In other words, SLI bypasses lock manager's *unbounded* communication by selective use of caching. To avoid starving transactions that make incompatible requests to the lock, waiting requests cause agents release the lock at transaction end.

5.2.1 Passing information between agent threads

Although it primarily serves to reduce contention in the lock manager, SLI has the beneficial side effect of reducing contention and overhead associated with the database catalog. The catalog maintains metadata such as the root pages for each index and heap file, as well as tracking the assignment of pages to those files. In order to avoid recovery-related corner cases, the transaction must verify the membership of every page it accesses in case a crash interrupted a page allocation operation.[4] Although each transaction caches metadata to avoid repeated accesses and latching overheads, short transactions tend to access each metadata item only once before committing and releasing the locks that prevented the underlying metadata from changing.

In a naive design, metadata page accesses cause significant unbounded communication. Transactions that perform few operations per object accessed are especially vulnerable, because each new transaction begins with an empty cache. However, with SLI active, the high-level locks that prevent updates to the cached metadata are not released between transactions. As long as it retains these locks, an agent thread can also retain the contents of its metadata cache, eliminating both contention and overhead that would come from refilling it. There is essentially no downside because metadata change infrequently; when they do change, the calling transaction first acquires the corresponding locks; the incompatible requests cancel lock inheritance, and the affected cache entries are dropped.

As the evaluation section shows, SLI improves both system utilization and the fraction of utilization which corresponds to useful work. This translates to significant performance improvements, especially for workloads that stress the lock manager heavily. We note, however, that SLI does not reduce the single-thread overheads imposed by database locking. These overheads still account for roughly 15 % of total execution time, consistent with findings reported in the literature [29]. As we will see in Sect. 6, a careful redesign can eliminate both overhead and contention due to the lock manager, as well as opening the door to further optimizations in other parts of the system.

6 Logical partitioning

So far, all the techniques presented in this paper improve the performance of conventional transaction processing designs, without requiring significant changes to the system architecture. For example, SLI (Sect. 5.2) does not change how agents acquire database locks, but rather arranges for each new transaction to begin with the hottest locks already held and in the transaction's local cache. In the case of cooperative log buffer inserts (Sect. 5.1), changes are localized to the log buffer code, with each agent continuing to submit the same log insertion requests as before. Although effective, these approaches do not fundamentally alter the system's logical behavior: Locks must still be acquired and log entries must still be recorded. In this section, we highlight a more disruptive approach: re-architect the contention-prone components so communication patterns become explicit, to expose and deal with them.

To motivate the proposed changes, we identify the conventional engine's thread-to-transaction policy for assigning work as the primary cause of unbounded communication in the system. Then, we present data-oriented transaction execution or DORA [56], an event-based transaction processing system design (similar to SEDA [72] and QPipe [30]) that employs a thread-to-data assignment policy designed to minimize unbounded communication, especially in the lock manager. Where traditional shared-everything systems allow each agent to access whatever data the transaction specifies, DORA assigns each agent in the system a logical partition of the data, and no other agent accesses that data. DORA then decomposes each incoming transaction into a data-flow graph whose nodes each access data controlled by a single agent. This logical partitioning avoids most conflicts and allows database locks that remain to be managed privately by their owning agent thread, without resorting to centralized coordination.

[4] Because delete operations do not update pages on disk, a page which was deleted and then re-allocated just before a crash may still announce its old ownership during recovery.

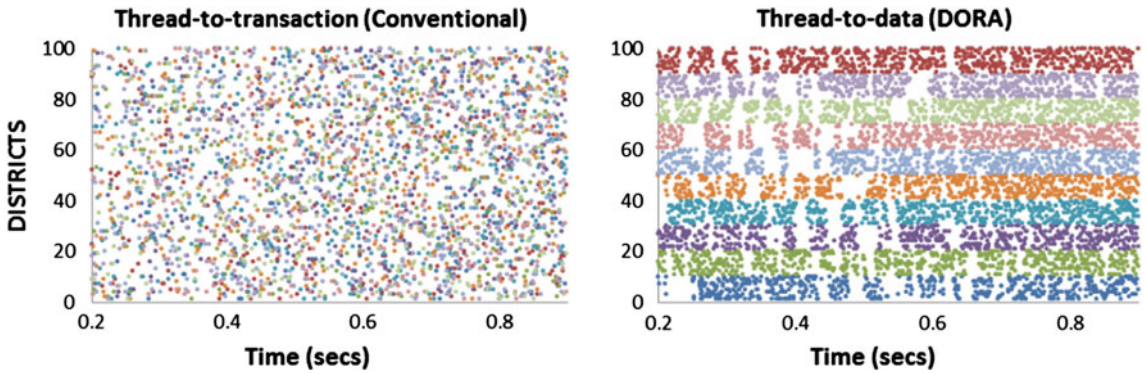

Fig. 9 Trace of the record accesses by the agents of a conventional system (*left*) and a DORA system (*right*). The data accesses for the conventional are highly unpredictable, while for DORA they are coordinated and regular

6.1 Thread-to-transaction versus thread-to-data

Traditional systems assign each transaction to an agent, a mechanism we refer to as *thread-to-transaction* assignment. Because workers access arbitrary data on behalf of assigned transactions, concurrent transactions generate communication to both the logical and physical levels and cause contention that limits scalability.

The left part of Fig. 9 depicts the accesses patterns arising in a conventional transaction processing system as ten agents repeatedly serve `TPC-C Payment` transactions submitted to a database of ten `Warehouses`. This corresponds to a `TPC-C` database of scaling factor 10, intentionally small to enhance readability of the graph. We track the database's 100 `District` records on the y-axis over 0.7 seconds of execution along the x-axis; each access by a worker produces a dot whose color identifies the worker involved.

The chaotic, uncoordinated access pattern of the thread-to-transaction (i.e., conventional) assignment policy is apparent by visual inspection. In order to maintain data integrity, each agent enters a large number of critical sections during the short lifetime of each transaction it executes—70 or more for even the simplest transactions—and the overhead of the unscalable critical sections increases with the number of parallel agents. In contrast, the plot on the right shows the same benchmark under DORA. DORA produces highly regular access patterns that can be exploited to reduce synchronization overheads.

Shared-nothing designs with physical partitioning naturally produce a thread-to-data assignment of work and eliminate all centralized mechanisms. Shared-nothing thus achieves the desirable access patterns shown in Fig. 9 (right), but the physical separation produces two undesirable side effects: Transactions that access data in more than one partition must coordinate using some distributed protocol such as two phase commit (a notoriously unscalable solution [31,58]), and non-uniform data accesses lead to load imbalances where some partitions see high load and others see very little [18]. Unfortunately, frequent repartitioning is prohibitively expensive under the shared-nothing discipline because the data involved must be physically migrated to a different location. Ideally, we would like a system with the best properties of both shared-everything and shared-nothing designs: A centralized data store that sidesteps the challenges of distributed transactions and of data movement during (re)partitioning, but also an access scheme whose regular access patterns avoid unbounded communication.

6.2 Data-oriented transaction execution

Even with SLI in place, we find that locking is the biggest source of unbounded critical sections in the system and our main target with the thread-to-data approach. To eliminate contention in the lock manager, DORA logically partitions the data among agents, breaking transactions into components, which access only one logical partition (similar to how shared-nothing systems need to distribute data accesses among physical partitions). These actions are then directed to the appropriate agents by a new routing layer of the database engine. Each logical partition is accessed by a single agent, and agents use a private locking mechanism to coordinates accesses within the partition, limiting interactions with the centralized lock manager. Because the data are only logically partitioned, existing database engine services continue to provide full ACID compliance without distributed transactions, and load balancing requires only to update assignments maintained by the routing layer (any transient contention that results is handled by the underlying engine, whose concurrency control mechanisms remain fully functional).

In detail, DORA is a layer on top of a traditional storage manager and provides three basic services.

Binding agents to data. DORA couples agents with data using *routing rules* that map disjoint subsets of records to specific agents. Importantly, these routing rules do not change the physical layout of tables at all. All data resides in the same buffer pool and the routing rules imply no physical

separation or data movement. A resource manager adjusts the routing rules and the number of agents to tune the workload to the available hardware resources and for load balancing purposes.

Distributing transactions over data. To distribute the work of each transaction to the appropriate agents, DORA decomposes each transaction into a *transaction flow graph*. Each node of the graph captures accesses to a single data set, with edges capturing control- and data-flow of the overall transaction. Although most actions can be routed statically, the graph allows correct routing of actions having dynamic data dependencies, once their inputs become available. In addition, the system effortlessly exploits any intra-transaction parallelism exposed by the partial order of dependencies, improving transaction response times significantly (especially in cases of low concurrency) and reducing the amount of time locks are held.

Executing requests. DORA routes all actions that access the same data set to one agent that maintains isolation and imposes ordering among conflicting actions. Actions are processed in the order they enter an *inbound* queue, and the agent uses a *private lock table* to detect conflicting actions, based on the data they indicate they will access. Because actions may access overlapping subsets of data, the locking scheme employed is similar to that of key-prefix locks [25]. Once an action acquires the local lock, it proceeds without further concurrency control. Local locks are held until the overall transaction commits or aborts, following a strict 2PL protocol. After a transaction commits globally (i.e., after the log flush completes), the transaction's actions are sent back to a *committed* queue of their respective agents. The agents pick the actions of the committed transactions, update the local lock table, and allow blocked actions (of other transactions) to proceed.

In the spirit of SLI, each agent retains an intent exclusive (IX) lock for the whole table. The agent interfaces with the centralized lock manager only if the speculation is invalidated due to some table-level operation, such as a rebalancing operation in the routing layer. Transactions that merely modify large data ranges (e.g., a table scan) enqueue an action to every agent operating on that particular table rather than acquiring coarse-grained locks. We note that the number of agents per table is usually less than ten, so the additional cost of "fine-grained" locking in the case of table scans is quite tolerable. DORA agents also benefit from the same metadata caching as SLI.

In summary, DORA exploits the efficient inter-core communication of multicore hardware. Transactions flow from one agent to the other with minimal overhead, as each agent accesses different parts of the database. DORA eliminates virtually all the unbounded communication in the lock manager, substituting the centralized lock management with much lighter-weight thread-local locking mechanism. The remaining biggest sources of unbounded communication are page latching and buffer pool management, which we address next.

7 Beyond logical-only partitioning

The previous section outlines scalability problems faced by conventional shared-everything transaction processing systems: Assigning transactions to agents means that any agent might access any data item at any time (even though it would not usually collide with another), requiring communication between agents and careful concurrency control at both the logical and physical levels of the system. As a remedy, the previous section proposes to assign agents to data instead to transactions (a form of logical partitioning), but the DORA design only addresses communication related to logical locking.

Although logical-only partitioning eliminates contention-prone communication in the lock manager, it does little to reduce contention-prone communication arising from physical data accesses and the corresponding latching of data pages and buffer pool frames. Physically partitioned systems eliminate communication from physical data accesses at the cost of introducing distributed transactions. To achieve the benefits of physical partitioning without the costs that usually accompany it, we observe that partitioning is effective because it ensures single-threaded access to each item in the database; physical partitioning is not necessary. Any scheme that arranges for a thread-to-data policy should achieve the same regularity and reduced reliance on centralized mechanisms.

Thus, we extend the DORA design in order to privatize the majority of the physical accesses as well, with two designs we call *PLP* and Overlay Bufferpools. Just as DORA eliminates most communication due to record locking, PLP eliminates nearly all communication due to page latching, by ensuring single-threaded access to most physical data structures in the system. PLP achieves this goal with a new access method that we call MRB+Tree. This index structure allows partitioning of data within database files and indexes, capturing logical accesses and most types of physical data accesses as well. Any page affected by PLP becomes fully latch-free; only a few types of metadata pages, having no relation to any logical data item (e.g., free space management) use latches.

Additionally, we show that a simple change in the page cleaning protocol converts page cleaning (the only remaining source of latching in the buffer pool) to message passing. Orthogonal to the benefits in scalability coming from removing the last remaining source of unscalable communication, the average latency of the buffer pool frame accesses can be further improved in multisocket systems by migrating the

frame to the socket where the frame's owning agent resides. The modified page cleaning protocol and the active buffer pool which migrates the frames to the socket hosting the page's owner agent, we call Overlay Bufferpools.

Combined, DORA and the two techniques presented in this section eliminate the vast majority of unscalable critical sections in the system. Such a system in some cases executes two orders of magnitude fewer unscalable critical sections than a conventional system and achieves up to 50 % higher performance than an optimized conventional system employing all the techniques presented earlier in this paper. In addition, the code path simplifications that accompany the elision of locking and latching are significant, leading to a simpler and more maintainable system.

7.1 Latch-free execution in a shared-everything system

Both conventional systems and DORA employ *page latching* in order to ensure that physical modifications to the data leave the system in a consistent state and to prevent readers from encountering partially applied updates that might appear inconsistent. Although page latches can be released as soon as the corresponding page access completes (unlike locks, which are held to transaction completion), they are acquired far more frequently and can easily lead to contention that limits scalability [38]. Even in the absence of contention, page latching imposes a significant penalty to single-thread performance [29] and leads to complex and subtle access methods [54]. In addition, the performance of the system is sensitive to *page false sharing*, where hot—but unrelated—records happen to reside on the same page, serializing logically unrelated agents that access those records. Careful tuning is often needed to detect and resolve such issues, for example by padding problematic records to spread them out.

To eliminate physical contention, we must extend the DORA design (which keeps the physical layer unchanged) deep into the storage manager. That is, we need to alter access paths and physical table layouts to promote the same kind of regular access that DORA provides at the logical level. To this end, we propose the concept of *PLP*, logical partitioning of the physical data accesses, enabled by the MRB+Tree, a B+Tree variant which allows logical partitioning of data pages.

7.1.1 Multirooted B+Tree

PLP adapts the traditional B+Tree [10] by splitting it into multiple sub-trees, each covering a contiguous range of the key space. Crucially, the top-level partitioning information, and the various sub-trees are all part of the same database object. Figure 10 gives an example of two MRB+Trees having two and three partitions. Each entry in DORA's routing

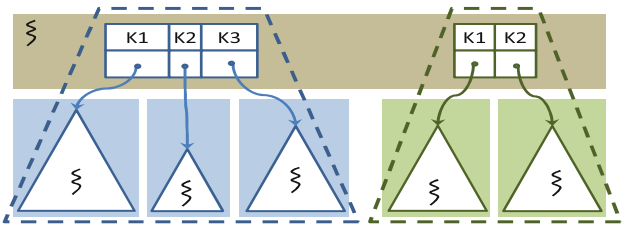

Fig. 10 The MRB+Tree and its integration with PLP

tables points to the root of a sub-tree, with the routing table becoming the root of the overall tree. The logical mapping of key ranges to agents thus becomes a durable part of index metadata. By moving DORA's routing tables to the physical layer, any given page reachable through an index will be accessed by only one agent. However, unlike the shared-nothing system, the partitioning is dynamic and malleable because the various sub-trees remain part of the same physical file (indicated by dashed lines in the figure). Rebalancing can be effected by simply rewriting pointers near the top of the tree and involves very little data movement (on the order of ten database pages per repartitioning). The tree is thus very robust to skewed access patterns, because the system can react rapidly to imbalances [70].

When deployed in a conventional system, the MRB+Tree eliminates latch contention at the index root (and possibly the false sharing at the leaves, if combined with a load balancing mechanism). Partitioning also reduces the expected tree height by at least one, because the routing layer manages tree roots and agents interact only with their sub-tree. Further, partitions containing hot data can be very small, giving additional performance benefits. In Fig. 10, for example, the middle partition of the left tree is significantly smaller than its peers. Finally, the MRB+Tree can also potentially benefit systems that use shared-nothing parallelism in a shared-memory environment, such as [42,68].

7.1.2 Latch-free index accesses

With the MRB+Tree in place, a PLP system assigns each sub-tree to a single agent, guaranteeing exclusive access and latch-free execution. As with DORA, the agents are unaware of the partitioning; the partition manager has exclusive access to all partition tables and ensures that all work assigned to an agent involves only data it owns. Agents thus start each tree probe from the root of their assigned sub-tree. All indexes in the system—primary, secondary, clustered, non-clustered—can be implemented as MRB+Trees. Some care is needed to handle secondary (non-clustered) indexes whose key does not identify tuples' partitions. In this case, the partitioning key must be stored with each leaf entry so the action can be forwarded to the correct agent for further processing.

By partitioning physical access, PLP thus eliminates all latching of affected pages. The MRB+tree suffers neither overhead nor contention from latching, avoids false sharing, and can perform structural modification operations (SMO) without concern for other agents in the system, with potential for significant simplification of the associated code paths. The latter two overheads are often linked, as ARIES indexes [53] allow only one SMO (such as a leaf split) to occur at a time, and complex protocols are required to prevent the SMO from interfering with concurrent probes by other agents [37,54]. We have not yet attempted the code refactoring required to exploit all these opportunities, so our reported results are conservative: [29] reports significant overheads during transaction processing approaching 50 % of total instructions executed, due to latching, locking, and index probe operations.

7.1.3 Heap page accesses

In the base PLP design, heap pages still require latching because the system may allocate in the same heap page records belonging to different partitions. The overhead of latching may be acceptable, because heap page accesses contribute far less than index pages to overall access time, and most heap pages will not span partitions in practice. However, allowing heap pages to span partitions prevents the system from responding automatically to false sharing or other sources of heap page contention that might arise. We therefore modify the MRB+Tree and heap management code paths so that heap page accesses are partitioned as well. We find that the best approach is to require every heap page to "belong" to only one index leaf page. That complicates free space management but preserves fast repartitioning. Details of this design trade-off are found elsewhere [57,70].

7.2 Overlay buffer pools

As a final step, we consider the database buffer pool manager. PLP was previously reported to leave only the buffer pool as a significant source of unscalable communication [57], but we will show in this section that the previous assessment was overly conservative. We also discuss how PLP can improve memory access locality in multisocket NUMA architectures, by using its stable access patterns to guide page placement. We give the name "overlay buffer pool" to a PLP-enabled bufferpool manager, because it overlays several logical buffer pools (one per worker) and allows them to share memory effectively. Page migration can improve locality after repartitioning occurs, but is not necessary for continued operations.

Recall that, under PLP, nearly all data pages in the database are each accessed by only a single agent, as guided by the logical partitioning in place at any given moment. Even without making any changes in buffer management code, this virtually eliminates unscalable communication within the buffer manager. Consider the three main sources of communication:

1. Agent–agent contention for database page accesses.
2. Agent–agent contention during page lookup.
3. Interference between agents and page cleaners.

PLP eliminates the first source of communication by ensuring only one agent requests any given data page at a time. The second case arises because a page could reside at multiple locations within the buffer pool, and the associative search (using a hash table in the case of Shore-MT and most other database engines) could lead to contention. In practice contention is minimal. First, agents only collide on hash collisions, which should be relatively rare. Further, most accesses are short and read-only, making them amenable to implementation using lock-free algorithms. Even if we treat them as shared communication, hash table lookups have negligible cost. The remaining case, interference from page cleaners leaves a small component of communication.

Fortunately, page cleaners in Shore-MT, and most database engines, are partitioned based on the number of disks, in order to write back dirty pages as efficiently as possible. Therefore, PLP involves two levels of partitioning: at the agent layer (ensuring that at each point of time a single agent accesses each data page) and at the page cleaner layer, where are as many cleaners are the number of disk spindles. Page cleaners in Shore-MT also make a copy of each page to a page cleaner private buffer before performing I/O, to minimize the window of vulnerability (the time to copy a memory page dwarfs to the time to do the page write I/O), and this strategy works even better under PLP.

Because at most one page cleaner and one agent thread contend for each buffer pool frame, there is potential for further optimizations that convert this communication (and corresponding critical sections) to light-weight message passing. In conventional systems page cleaners acquire a shared-mode page latch while making copies for write back. That cost can be reduced by a protocol similar to optimistic concurrency control and by using the page LSN to validate latch-free access. Since the agent updates the LSN before making changes, as part of the write-ahead logging protocol, a changed LSN is a conservative proxy for a changed page. Thus, the cleaner threads would have to retry if a page's LSN changed during the copy operation. A cleaner that repeatedly fails to take an accurate copy of a page ensures progress by enqueuing a request with the owning agent, though this case would be extremely rare; the agent would process the request at a convenient time and notify the page cleaner upon completion.

In short, the page cleaning protocol is as follows. Note that in order to be able to achieve this protocol, at the header of each data page we store also the identifier of the page owning agent or logical partition.

1. Copy the dirty page (copy out) and verify page LSN.
2. If the page LSN did not change, write the copy of the dirty page to disk.
3. Otherwise, retry the copy operation.
4. Excessive failed copy attempts trigger a request for the agent to copy the page manually.

A final observation is that this design can guide page placement in multisocket NUMA machines: assuming that agents are pinned to sockets, accesses to most pages (and thus buffer pool frames) originate from a single socket and the page (and frame) can be migrated to that agent's socket.[5] Such a NUMA-aware optimization is not possible in conventional designs. Once the system balances and reaches a steady state (facilitated by the robust re-balancing mechanism presented in [70]), we do not expect significant number of pages to migrate between bufferpools and sockets. The few shared metadata pages in the system can be easily identified as having no owner and can be distributed uniformly among sockets.

Although page placement can have a significant impact on performance in systems with slow interconnect or many sockets, as shown in [58], we defer this exploration to future work. We do not have access to appropriate hardware,[6] and such an investigation does not fit in the context of this paper, which primarily investigates the communication patterns between agents of a database engine during transaction processing.

8 Evaluation

In this section, we present a performance evaluation and comparison of the various techniques discussed in previous sections.

8.1 Experimental setup

Hardware. In order to show that the presented solutions improve performance in nearly any modern hardware, we use two multicore servers representing both the "lean" and the "fat" core camps [28]. In particular, our "lean" core camp representative is a single-socket Niagara II server, with 64 in-order hardware contexts each clocked at 1.4 GHz and 64 GB of main memory running Solaris 5.10. For the "fat" core camp, we use a two socket x86_64 server with octo-core Intel Xeon E5-2650 Sandy Bridge-EP processors (16 cores in total) clocked at 2 GHz and 64 GB of main memory running SUSE Linux Enterprise Server 11 SP2.

Workloads. Our experiments use benchmarks known to expose overheads in the database engine:

TATP (telecommunications).[7] Originally developed by Nokia, it models call tracking and forwarding in a cell phone network. Transactions are very small, affecting only a few rows each, but execute with strict latency requirements to avoid dropped calls. A significant fraction of transactions abort, stressing rollback paths.

TPC-B (banking).[8] Although simplistic, this benchmark utilizes heavily the logging and locking components of a database engines and makes a useful stress test.

TPC-C (commerce).[9] The de-facto standard OLTP workload, modeling a wholesaler's order processing operations. Transactions are small- or medium-sized, with the largest query touching perhaps a thousand records.

Log insert microbenchmark. To further stress the logging subsystem, we use a microbenchmark where an increasing number of threads repeatedly insert log records to a log buffer. The distribution of log record sizes follows Shore-MT's distribution of log record sizes.

Together, these workloads exercise virtually all DBMS features and provide a good indication of its overall scalability. We focus on short transactions because they expose overheads within the DBMS; longer transactions can amortize DBMS overheads over their lifetimes and tend to obscure scalability bottlenecks.

8.2 Anatomy of critical sections

The focus of this paper is the communication patterns of shared-everything transaction processing systems. As we already discussed, the points of inter-agent communication are shown as critical sections during transaction execution. Figure 11 shows the average number of critical sections entered by the different techniques presented in this paper, as they execute three transaction-heavy benchmarks: `TATP` (left), `TPC-B` (middle), and `TPC-C` (right). For `TPC-C`, we use only the two most frequent transactions, `NewOrder` and `Payment`, which comprise 88 % of the overall transactions executed in the benchmark.[10]

[5] Libraries such as libnuma support socket-aware allocation and migration of memory regions.

[6] The machines used in Sect. 8 do not suffer significant inter-socket communication latencies.

[7] http://tatpbenchmark.sourceforge.net/.

[8] http://www.tpc.org/tpcb.

[9] http://www.tpc.org/tpcc.

[10] Two of the other transactions in `TPC-C` are read-only, while the last one, `Delivery`, causes logical contention and low concurrency.

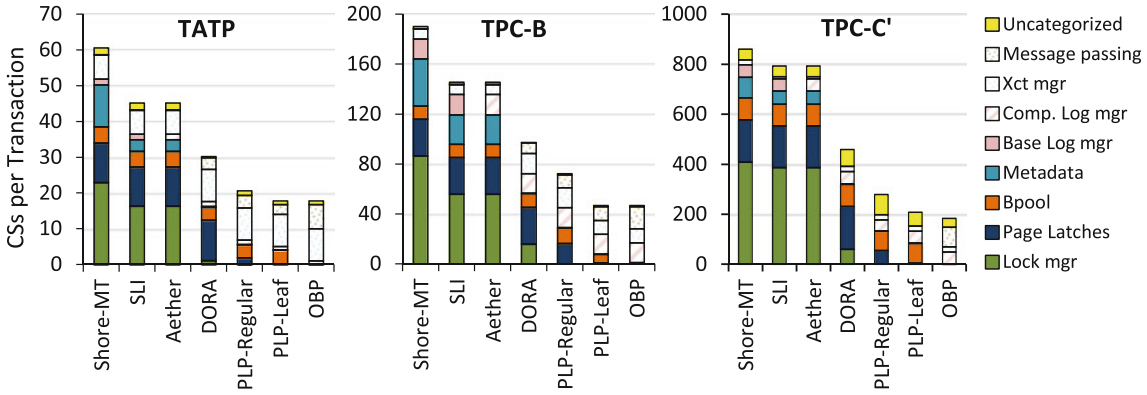

Fig. 11 Average number of critical sections imposed by various components of the transaction processing system as it executes three transactional benchmarks, TATP (*left*), TPC-B (*middle*), and TPC-C (*right*). Unbounded critical sections use *dark solid colors*, cooperative ones use stripes, and fixed ones use *light solid colors*. Better designs have fewer unbounded critical sections (color figure online)

In each graph, the three left-most bars show the critical sections entered by variations of our "conventional" baseline transaction processing system, Shore-MT. The left most is the original release of Shore-MT, which achieves competitive scalability and performance versus other open source transaction processing systems [38]. The next two bars show the results of optimizations we described in Sect. 5. In particular, the second bar is Shore-MT with speculative lock inheritance (SLI) active, and the third bar (labeled as "Aether") is Shore-MT both SLI and composable logging active.[11]

The second bars of each graph show the anatomy of critical sections when SLI is active. We observe that the number of critical sections entered on average is reduced, for two reasons. First, SLI is efficient in picking the right locks to inherit, so transactions interact with the lock manger less frequently. Also, as we discussed in Sect. 5.2.1, metadata information propagates from one transaction to the next, reducing the interaction with the metadata manager as well. The final result is that the number of un-scalable critical sections drops by more than 20 %. SLI has little impact on TPC-C, because there are few opportunities for inheriting locks. Composable logging converts the majority of unscalable critical sections in the log manager to cooperative communication, reducing the number of unbounded critical sections by another 5.5, 12, and 6.6 % for the three benchmarks (and eliminating logging as a source of contention). A significant number of unbounded critical sections remains, however, particularly those related to locking, metadata management, latching, and buffer pool accesses.

The last four bars of Fig. 11 examine the impact of the designs based on data-oriented execution (DORA, PLP, and Overlay Bufferpools). Data-oriented transaction execution results shifts communication patterns dramatically. DORA eliminates nearly all the critical sections related to the centralized lock manager and metadata, but has little impact on latching. The two PLP variants (without and with heap page partitioning) eliminate the vast majority of critical sections from page latching as well, leaving only space management page latching. Overlay Bufferpools (marked as OBP) take care of the remaining buffer pool-related critical sections. As described in Sect. 7.2, with Overlay Bufferpools, the buffer pool latching is converted to point-to-point communication between the asynchronous page cleaning thread responsible for a particular frame and the agent responsible for that page. In OBP, we categorize the buffer pool latching as message passing. We note that transaction management contributes a fair amount of communication, which arises because multiple DORA worker threads can access the same transaction object concurrently (recall that the data-oriented designs exploit intra-transaction parallelism); such contention is fixed because it depends only on the dataflow of a given transaction, not on the number of transactions.

Compared with the baseline conventional system, DORA reduces the number of critical sections acquired by around 50 % in all benchmarks, and PLP and OBP in turn reduce the critical section count by another 40 % or more across all benchmarks compared with DORA. Even more dramatic is the reduction in the count of unscalable critical sections. When compared with the optimized conventional system, DORA reduces the unscalable critical sections by 55 %. PLP and OBP reduce the unscalable critical section count further, by 70 and 93 % in TATP, 85 and 96 % in TPC-B, and 75 and 96 % in TPC-C. Overall, a system employing PLP with overlay bufferpools eliminates over 99 % of unbounded critical sections for all workloads. We note that the code paths for such critical sections are not merely executed less commonly—many become dead code that can be safely removed.

[11] The current public release of Shore-MT contains all these features.

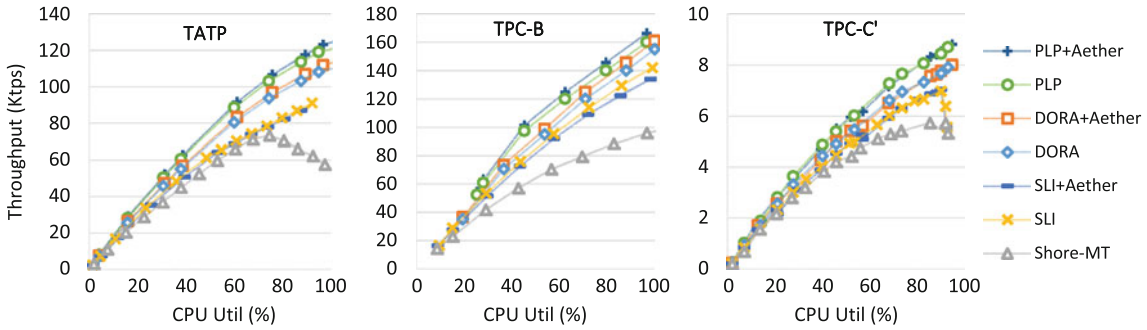

Fig. 12 Comparison of the scalability and performance of the various techniques are the run the TATP, TPC-B, and TPC-C benchmarks

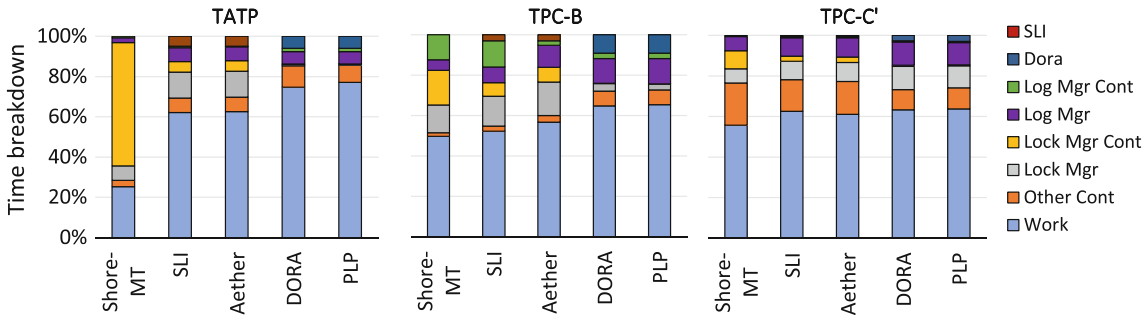

Fig. 13 Time breakdown of the various techniques when running the TATP, TPC-B, and TPC-C benchmarks

8.3 Impact on scalability and performance

Next we examine what is the impact of the different critical section breakdowns to the scalability and performance of the systems under comparison (for DORA and PLP we have consolidated buffer inserts active). Figure 12 compares the performance of the various systems as the load of the Niagara machine increases while executing the TATP, TPC-B, and the modified TPC-C benchmarks. Figure 13 shows the time breakdown when the machine is almost fully utilized. The first observation we can make is that indeed profiling the communication patterns (or critical sections) is a reliable predictor of the scalability and performance of the systems. The fewer the points of unscalable communication, the better the performance.

In detail, with SLI active, none of the workloads has a large contribution from lock manager contention. This indicates that SLI is effective in identifying and passing the locks which cause most lock manager contention. We also note that SLI has low overhead—even in the worst case it adds only <5 % overhead, usually with a corresponding decrease in lock manager overhead. For example, locks which are inherited but not re-used must still be released and that overhead counts toward SLI, not the lock manager. For most transaction profiles, contention in the lock manager was replaced by useful work, though in some cases other bottlenecks replace the lock manager. Overall, SLI improves the fraction of useful utilization. This translates to performance improvements of 30–50 % for workloads that stress the lock manager heavily, such as TATP. Again, SLI does not significantly reduce the single-thread overheads imposed by database locking. These overheads still account for up to 15–20 % of total execution time.

The consolidated log buffer inserts affect the log-related time component leaving the remaining components the same. On the update-heavy TPC-B benchmark, the impact of consolidated log buffer inserts is a bit over 10 %, but on the other two benchmarks its impact is low. In Sect. 8.5, we explain why the impact of log buffer inserts is relatively low.

DORA's impact is twofold. Not only does it virtually eliminate contention at the central lock manager, but at the same time the lock manager overhead ("useful" lock-related work) is reduced significantly. In particular, DORA achieves 15–20 % higher performance than the optimized baseline system (which has SLI active). From the left graph of Fig. 12, we see that for TATP, DORA delivers a 21 % speedup over the optimized baseline case. The primary source of additional performance is the lighter-weight lock-related operations. DORA's overhead comes to less than 3 %.

The PLP design shows superior scalability, as evidenced by the widening performance gap with the other systems as utilization increases. For example, in Fig. 12 in its turn, PLP delivers an additional 10 % over DORA, or nearly 35 % over the optimized baseline. A significant fraction of PLP's speedup actually comes from the MRB+Tree probes, which are effectively one level shallower, since threads bypass the

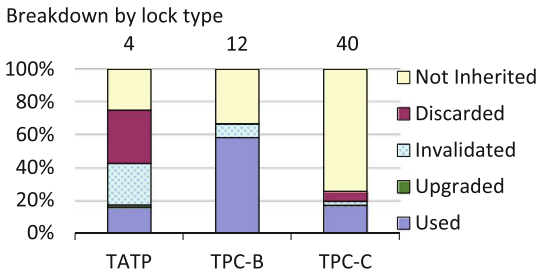

Fig. 14 Breakdown of outcomes for locks considered by SLI

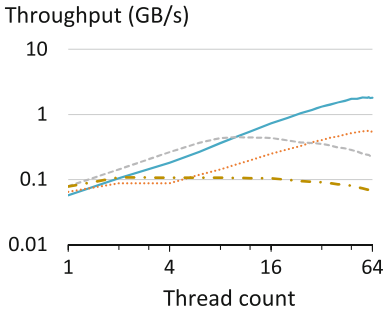

Fig. 15 Log insert microbenchmark results showing the scalability of four different log buffer designs: (B)aseline, (C)onsolidated, (D)ecoupled, and hybrid (CD)

"root" partition table node during normal operation. All in all, the index probes are the most expensive operation in a PLP system, accounting for roughly 30–50 % of all compute cycles in the workloads examined. Therefore, we expect significant performance improvements from incorporating a latch-free, lock-free B+Tree, such as one optimized for main memory systems, e.g., [60,61].

8.4 Important critical sections

Another observation we can make from the graphs of Sect. 8.3 is not all critical sections contribute the same. That is, there are critical sections which impact performance more than others. This can perhaps be seen most clearly with SLI. SLI aims to reduce contention within the lock manager—so it does not impede scalability—by identifying only some "hot" locks for which to bypass the lock manager. From Fig. 13, we see that SLI indeed alleviates contention in the lock manager. Figure 14 shows, per workload, the outcome of all locks that SLI could potentially act on. The categorization shows how often heritable locks were actually used, and how effectively. We observe that, in spite of the large impact on scalability, only a small fraction of heritable locks is actually used. Intuitively, only a few locks can be heavily contended at any given moment, because contending threads must spend little time elsewhere. However, SLI's targeted approach leaves locking overheads largely unchanged.

8.5 Impact of consolidating log buffer inserts

In Fig. 12, we see that the consolidated log buffer inserts have relatively small impact in performance. The overall log insert bandwidth demanded by those workloads is too low to differentiate consolidated log buffer inserts from the basic log insertion mechanism.

Figure 15 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 discussed in Sect. 5.1. 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 threads combine their requests as contention increases, and performance scales linearly. In contrast, decoupled insertions avoid the initial performance penalty and perform better, but eventually growing contention degrades performance. Overall, we see that a combination of decoupling and consolidation moves large data copy operations off the critical path while allowing bursts of log records to combine for lower contention. The hybrid log buffer insert design makes the most difference when throughput passes 300 MB/s, well above the amount these workloads generate on our systems.

8.6 Secondary and non-aligned accesses

Typically, if a transaction accesses a record based on a non-primary key column, a secondary index is required. Secondary indexes are very important in transaction processing, but they also pose challenges to data-oriented execution. As a transaction may use arbitrary columns for accessing records, those columns might not be part of the partitioning scheme in data-oriented execution, leading to a two-step process where the secondary index probe identifies the primary key and owning partition, whose agent then fetches the requested data.

We explore the impact of secondary indexes in data-oriented execution in Fig. 16. We break the analysis of secondary index accesses into two cases: When the secondary index is aligned with the partitioning scheme and when it is not. The workload consists of an increasing number of clients submitting transactions that contain a (secondary) index scan of increasing range of 10, 100 and 10 K records, respectively. Figure 16 plots the performance, as more hardware contexts are utilized, of *Shore-MT* and *SLI* (the conventional designs), with a DORA and a PLP system whose secondary indexes are aligned with the system's partitioning scheme, labeled *DORA-Aligned* and *PLP-Aligned*, and a DORA and a PLP system whose secondary indexes are

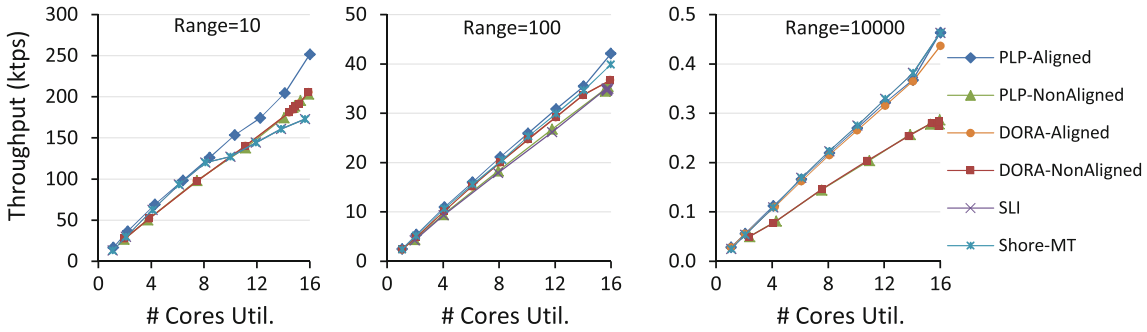

Fig. 16 Performance on transactions with partitioning aligned and non-aligned secondary index scans

not aligned with the partitioning scheme, labeled *DORA-NonAligned* and *PLP-NonAligned*. *PLP-Aligned* improves performance over *Shore-MT* by 46, 14, and 1 %, respectively, for ranges 10, 100, and 10 K. On the other hand, even though *PLP-NonAligned* improves performance by 11 % when 10 records are scanned, for larger ranges it hinders performance; *PLP-NonAligned* is 3 and 38 % slower than *Shore-MT* for ranges 100, and 10 K, respectively. These overheads arise because even a small number of records can spread over multiple partitions, multiplying the cost of each data access.

As expected, the performance improvement for *PLP-Aligned* gets smaller as the range of the index scan increases, because more records amortize the cost of the probe and leave less opportunity. However, as long as the index scans of partition-aligned secondary indexes are selective and touch a relatively small number of records, PLP provides decent performance. For *PLP-NonAligned*, however, such workloads are very unfriendly, though unless the scan range is over 1,000 records it is not disastrous.

Overall, logical partitioning and PLP are quite successful in significantly improving scalability. The key to their success lies in removing unscalable communication patterns and preventing opportunities for contention from arising at all.

9 Related work

There exists a large body of research in several sub-areas of computer science which touch on the work summarized in this paper. In the interest of space, we categorize briefly here the general types of related research and refer the reader to the original published versions of the different techniques presented here for more specific discussion and citations.

Behavior on modern hardware. Many studies have examined the execution of database engine code on modern hardware, concluding that database codes are highly unfriendly to sophisticated architectural techniques such as out-of-order execution [8,59] and large caches [3,28]. The multicore trends toward simpler but more numerous cores therefore fits nicely with the needs of database engines [20]. However, significant additional effort is required to utilize effectively constrained shared resources such as cache capacity and memory bandwidth [15,16]. In addition, recently special attention has been in the performance of data management systems on multisocket multicore [4,58]. Such efforts are important for performance and are mostly orthogonal to much of what we present in this paper.

Relaxing consistency. The increased need for scalable data management services has driven systems to relax ACID requirements (e.g., [71]) and/or drop "traditional" data processing capabilities of database management systems (e.g., the Key-Value stores [14,22]) for increased performance. In cases where the application can tolerate relaxed consistency or can function with only Key-Value accesses, existing "NoSQL" designs reduce the frequency of communications in the engine with low design complexity. However, these approaches leave the application to implement any necessary coordination manually, and reasoning correctly under relaxed consistency is notoriously difficult. In addition, a large class of enterprise and other applications cannot tolerate such compromises. ACID is thus either required or preferable as long as the system can provide sufficiently high performance [67].

Shared-everything versus shared-nothing. The debate between shared-everything and shared-nothing disciplines is not new, dating back to the early days of computer architecture. It is now known that the two models are equivalent [7,46]—in the sense that each can be expressed in terms of the other—but shared-memory and message passing usually provide very different performance for a given combination of hardware platform and workload. In general, shared-memory excels at abstracting away communication patterns, while message passing makes all communication explicit and therefore requires more programmer effort to expose communication patterns. This equivalent but not equal status led to the dominance of shared-memory models during an era of small-scale parallelism (e.g., 4–8 processors at most), and currently motivates a resurgence of message-passing and data-flow parallelism in operating systems [9,36], program-

ming languages [17], and software architectures [12,30,68]. In general, we find that message passing, while not removing problematic communication patterns, at least makes them easier to identify; however, shared memory remains a highly convenient abstraction even in highly parallel systems.

In a recent work, Porobic et al. [58] conducted a comparison between shared-everything and shared-nothing configurations of various size in large multisocket servers of multicore processors. They show that in perfect partitionable transactional workloads, the performance of fine-grained shared-nothing configuration can be up to 4× higher than the performance of a corresponding shared-everything configuration. On the other hand, even within a single node the cost of executing distributed transactions becomes a burden for the performance of shared-nothing configurations, especially of those with numerous small partitions. A configuration that combines most of the benefits of the two extremes is a shared-nothing configuration where the resources of each socket are used by a separate database instance.

A large body of related work targets partitioning costs, but most of it focuses on clustered (shared-nothing) environments. For example, Achyutuni et al. [1] compare different approaches for index reorganization during repartitioning in shared-nothing deployments. Lee et al. [48] propose an index structure, similar to the MRB+tree, which eases the index reorganization during repartitioning in a shared-nothing system.

Event-driven software architectures. The data-oriented execution model is essentially an event-driven (e.g., [72]) transaction processing architecture. As an event-driven system, DORA shares similarities with systems such as persistent workflows and staged database systems [30], with the difference that in transaction processing, the requests are much shorter lived and hence many aspects of the design have to be revisited. Also, event-based systems have been shown to provide strong scalability benefits to near-stateless applications such as web serving, and DORA/PLP applies that concept effectively within the database engine, with a minimum of so-called stack ripping (PLP achieves good scalability without inverting large sections of control flow).

Lock-free algorithms and transactional memory. The difficulty and overhead of enforcing pessimistic concurrency control and mutual exclusion have led to a family of lock-free algorithms [33] which employ carefully crafted sequences of atomic operations in order to update data structures without mutual exclusion. Those in turn have motivated proposals for hardware- and software-based transactional memory [34,62] systems which allow more optimistic (and lower-overhead) concurrency control, often with hardware support. The concepts behind transactional memory are strongly related with database transactions, with some promise of making parallel programming easier to "get right."

Unfortunately, performant transactional memory requires hardware support which is unlikely to become available in the near future. Further, as we noted in the introduction of this paper, transactional memory is only a means to respond to contention, rather than a way to prevent it arising, and therefore performs poorly in the presence of bottlenecks. Together these challenges mean that a well-designed system using mutex locks almost always outperforms an equivalent version based on transactional memory as parallelism increases [24]. Similar issues usually afflict lock-free data structures as well; in both cases, contention tends to turn them into particularly expensive spin loops unless some form of pessimistic concurrency control is used to moderate the problem. Nevertheless, we find that lock-free algorithms provide important capabilities which are difficult to achieve in other ways, and we use them extensively in Shore-MT.

Lock manager. Overheads of locking are well known [29], and several techniques have been proposed to mitigate or eliminate the costs of locking. Examples include single-threaded engines such as H-Store [68], and lock caching schemes that minimize passing of locks between nodes in a distributed DBMS [50]; Speculative Lock Inheritance is inspired by the latter technique, though the implementation and use cases differ significantly. Microsoft's SQL Server also incorporates a technique called "superlatching"[12] which resembles SLI but for page latches instead of locks. We note that, just as DORA largely obviates the need for SLI (by removing locking code paths entirely), PLP similarly reduces the need to invent clever latching techniques.

At a higher level of abstraction, multiversioning [11] and other optimistic concurrency control protocols [44] for database systems provide important benefits (such as allowing readers to continue using stale copies of data which have since been overwritten), but again suffer a tendency to turn transactions into (very) expensive spin loops under high contention. The Oracle DBMS, which employs optimistic concurrency control heavily, also features prominently the sophisticated rollback and recovery mechanisms needed to minimize the overhead of repeatedly retrying transactions which conflict.

Further, snapshot isolation (the isolation level normally provided by multiversioning) is prone to subtle anomalies which are difficult to prevent and make it difficult to port existing code correctly [41]. Serialization can be enforced in snapshot isolation with some care, but it is conservative and imposes overheads. For example, the abort rates are reminiscent of wound-wait deadlock detection schemes:

[12] See, e.g., http://blogs.msdn.com/b/psssql/archive/2009/01/28/hot-it-works-sql-server-superlatch-ing-sub-latches.aspx.

pre-emptively abort transactions that add "backward"-facing edges in a dependency graph, a sufficient but not necessary policy for avoiding cycles. Such schemes also tend to be non-transparent, as modifications to the application are often required. Further, we observe that a multiversion buffer pool would actually benefit a DORA/PLP implementation, by simplifying scheduling of local requests within a partition.

In addition, our techniques focus on the underlying communication patterns and are mostly orthogonal to the concurrency control mechanism which is employed. As an example, [45] presents a lightweight multiversioning concurrency control mechanism that incorporates lessons learned in this study.

Lightweight intent locks [43] are a recently proposed alternative to SLI which exploits fast atomic fetch-and-op in shared multicore caches to avoid contention for intent locks. However, performance degrades rapidly as "distance" between cores increases, either due to slower caches or to multiple sockets.

Log manager. Database log performance is a long-standing target of optimization efforts. Techniques such as group commit [32] and asynchronous commit aim to reduce the latency and overheads of log-related disk I/O, but do little to address contention at the log buffer.[13] The problem is similar to many encountered in distributed computing, including counting networks [6] and scalable aggregation employed in map/reduce frameworks [21], but log record sequencing is especially challenging because the log entry must record not only its own sequence number, but also any preceding ones that it depends on. Differential logging [47] allows multiple logs that can be recovered independently and in parallel, but it does not address the insertion-time complexities that make distributed logs unattractive for avoiding contention.

10 Concluding remarks

Database engines face the challenging task of extracting scalable performance from today's multicore hardware while still providing an effective means to arbitrate communication patterns exhibited at the application level. Because we cannot eliminate all communication in the general case, database engines must instead focus on minimizing the communication patterns which limit scalability. Unbounded communication patterns, where most threads in the system are involved with a given communication point, pose the most fundamental threat. But as we show in this paper, careful modifications to the database engine can convert them into less-threatening fixed or cooperative patterns which bound the amount of contention in the system.

[13] See, e.g., http://www.oracle.com/technetwork/database/clustering/overview and http://www.postgresql.org/docs/9.0/static/wal-async-commit.html.

Acknowledgments The authors are deeply grateful for all the members of Carnegie Mellon's StagedDB/CMP team and EPFL's DIAS laboratory who made this work possible through their research efforts, helpful feedback, and encouragement. We are especially indebted to Pinar Tözün who helped us with an additional set of experiments. We would also like to thank the many anonymous reviewers whose thoughtful and constructive remarks helped improve both this paper and the research papers summarized here. This research was supported by an IBM PhD fellowship; grants and equipment from Intel and Sun; a Sloan research fellowships; an IBM faculty partnership award; NSF grants CCR-0205544, CCR-0509356, IIS-0133686, and IIS-0713409; an ESF EurYI award; and Swiss National Foundation funds.

References

1. Achyutuni, K.J., Omiecinski, E., Navathe, S.B.: Two techniques for on-line index modification in shared nothing parallel databases. In: SIGMOD, pp. 125–136 (1996)
2. Ailamaki, A., DeWitt, D.J., Hill, M.D.: Walking four machines by the shore. In: CAECW (2001)
3. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs on a modern processor: where does time go? In: VLDB, pp. 266–277 (1999)
4. Albutiu, M.C., Kemper, A., Neumann, T.: Massively parallel sort-merge joins in main memory multi-core database systems. PVLDB **5**(10), 1064–1075 (2012)
5. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS, pp. 483–485 (1967)
6. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM **41**(5), 1020–1048 (1994)
7. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
8. Barroso, L.A., Gharachorloo, K., Bugnion, E.: Memory system characterization of commercial workloads. In: ISCA, pp. 3–14 (1998)
9. Baumann, A., et al.: The multikernel: a new OS architecture for scalable multicore systems. In: SOSP, pp. 29–44 (2009)
10. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. In: SIGFIDET, pp. 107–141 (1970)
11. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM TODS **8**(4), 465–483 (1983)
12. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: PACT, pp. 72–81 (2008)
13. Carey, M.J., et al.: Shoring up persistent applications. In: SIGMOD, pp. 383–394 (1994)
14. Chang, F., et al.: Bigtable: A distributed storage system for structured data. In: OSDI, p. 15 (2006)
15. Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Improving hash join performance through prefetching. ACM TODS **32**(3), 116–127 (2007)
16. Cieslewicz, J., Ross, K.A.: Adaptive aggregation on chip multiprocessors. In: VLDB, pp. 339–350 (2007)
17. Clark, K.L., McCabe, F.G.: Go! a multi-paradigm programming language for implementing multi-threaded agents. Ann. Math. Artif. Intell. **41**, 171–206 (2004)
18. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. PVLDB **3**, 48–57 (2010)

19. Daniels, D.S., Spector, A.Z., Thompson, D.S.: Distributed logging for transaction processing. SIGMOD Rec. **16**, 82–96 (1987)
20. Davis, J.D., Laudon, J., Olukotun, K.: Maximizing CMP throughput with mediocre cores. In: PACT, pp. 51–62(2005)
21. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, p. 10 (2004)
22. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. SIGOPS OSR **41**(6), 205–220 (2007)
23. Dewitt, D.J., et al.: The Gamma database machine project. IEEE TKDE **2**(1), 44–62 (1990)
24. Dragojevic, A., Guerraoui, R., Kapalka, M.: Dividing transactional memories by zero. In: TRANSACT (2008)
25. Graefe, G.: Hierarchical locking in B-tree indexes. In: BTW, pp. 18–42 (2007)
26. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD, pp. 173–182 (1996)
27. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco (1992)
28. Hardavellas, N., et al.: Database servers on chip multiprocessors: limitations and opportunities. In: CIDR, pp. 79–87 (2007)
29. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
30. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: QPipe: a simultaneously pipelined relational query engine. In: SIGMOD, pp. 383–394 (2005)
31. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR, pp. 132–141 (2007)
32. Helland, P., et al.: Group commit timers and high volume transaction systems. In: HPTS, pp. 301–329 (1989)
33. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
34. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2), 289–300 (1993)
35. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**, 33–38 (2008)
36. Hunt, G.C., Larus, J.R.: Singularity: rethinking the software stack. SIGOPS OSR **41**(2), 37–49 (2007)
37. Jaluta, I., Sippu, S., Soisalon-Soininen, E.: B-tree concurrency control and recovery in page-server database systems. ACM TODS **31**, 82–132 (2006)
38. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: EDBT, pp. 24–35 (2009)
39. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. PVLDB **3**, 681–692 (2010)
40. Jones, E., Abadi, D.J., Madden, S.: Low overhead concurrency control for partitioned main memory databases. In: SIGMOD, pp. 603–614 (2010)
41. Jorwekar, S., Fekete, A., Ramamritham, K., Sudarshan, S.: Automating the detection of snapshot isolation anomalies. In: VLDB, pp. 1263–1274 (2007)
42. Kemper, A., Neumann, T.: HyPer—a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206 (2011)
43. Kimura, H., Graefe, G., Kuno, H.: Efficient locking techniques for databases on modern hardware. In: ADMS (2012)
44. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM TODS **6**(2), 213–226 (1981)
45. Larson, P.A., et al.: High-performance concurrency control mechanisms for main-memory databases. PVLDB **5**(4), 298–309 (2011)
46. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. SIGOPS OSR **13**, 3–19 (1979)
47. Lee, J., Kim, K., Cha, S.K.: Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In: ICDE, pp. 173–184 (2001)
48. Lee, M.L., Kitsuregawa, M., Ooi, B.C., Tan, K.L., Mondal, A.: Towards self-tuning data placement in parallel database systems. In: SIGMOD, pp. 225–236 (2000)
49. Lomet, D.: Recovery for shared disk systems using multiple redo logs. Technical report CRL-90-4 (1990)
50. Lomet, D., Anderson, R., Rengarajan, T.K., Spiro, P.: How the Rdb/VMS data sharing system became fast. Technical report CRL-92-4 (1992)
51. Magnusson, P.S., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: ISPP, pp. 165–171 (1994)
52. Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. SIGPLAN Not. **26**(7), 106–113 (1991)
53. Mohan, C.: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In: VLDB, pp. 392–405 (1990)
54. Mohan, C., Levine, F.: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: SIGMOD, pp. 371–380 (1992)
55. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA, pp. 253–262 (2005)
56. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB **3**(1), 928–939 (2010)
57. Pandis, I., Tözün, P., Johnson, R., Ailamaki, A.: PLP: page latch-free shared-everything OLTP. PVLDB **4**(10), 610–621 (2011)
58. Porobic, D., Pandis, I., Branco, M., Tözün, P., Ailamaki, A.: OLTP on hardware islands. PVLDB **5**(11), 1447–1458 (2012)
59. Ranganathan, P., Gharachorloo, K., Adve, S.V., Barroso, L.A.: Performance of database workloads on shared-memory systems with out-of-order processors. In: ASPLOS, pp. 307–318 (1998)
60. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: VLDB, pp. 78–89 (1999)
61. Rao, J., Ross, K.A.: Making B+-trees cache conscious in main memory. In: SIGMOD, pp. 475–486 (2000)
62. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
63. Smith, A.J.: Sequentiality and prefetching in database systems. ACM TODS **3**, 223–247 (1978)
64. Soisalon-Soininen, E., Ylönen, T.: Partial strictness in two-phase locking. In: ICDT, pp. 139–147 (1995)
65. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM, pp. 149–160 (2001)
66. Stonebraker, M.: The case for shared nothing. IEEE Database, Eng. Bull. **9**, 4–9 (1986)
67. Stonebraker, M.: Stonebraker on nosql and enterprises. Commun. ACM **54**, 10–11 (2011)
68. Stonebraker, M., et al.: The end of an architectural era: (it's time for a complete rewrite). In: VLDB, pp. 1150–1160 (2007)
69. Thomson, A., Abadi, D.J.: The case for determinism in database systems. PVLDB **3**, 70–80 (2010)
70. Tözün, P., Pandis, I., Johnson, R., Ailamaki, A.: Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. VLDB J 1–25 (2012)
71. Vogels, W.: Eventually consistent. Commun. ACM **52**, 40–44 (2009)
72. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. In: SOSP, pp. 230–243 (2001)
73. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: ISCA, pp. 24–36 (1995)