Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?

Jose M. Faleiro and Daniel J. Abadi
Yale University

ABSTRACT

Recent research on multi-core database architectures has made the argument that, when possible, database systems should abandon the use of latches in favor of latch-free algorithms. Latch-based algorithms are thought to scale poorly due to their use of synchronization based on *mutual exclusion*. In contrast, latch-free algorithms make strong theoretical guarantees which ensure that the progress of a thread is never impeded due to the delay or failure of other threads. In this paper, we analyze the various factors that influence the performance and scalability of latch-free and latch-based algorithms, and perform a microbenchmark evaluation of latch-free and latch-based synchronization algorithms. Our findings indicate that the argument for latch-free algorithms' superior scalability is far more nuanced than the current state-of-the-art in multi-core database architectures suggests.

1. INTRODUCTION

Access to large-scale multi-core servers is becoming increasingly democratized. For instance, it is now possible to now obtain access to virtual machine instances consisting of 64 physical CPU cores on Amazon EC2 [8], while large multi-core servers have been available on the market for several years. This trend has led to significant research interest in database system architectures that effectively exploit parallelism on a single machine.

One perceived issue with conventional DBMS architectures is their widespread use of latches. In order to protect the integrity of shared data-structures within the database system, latches ensure that only one thread at a time can modify a shared data-structure. Latch-based algorithms are thought to be susceptible to performance problems at scale; if a thread is delayed while holding a latch, then no other thread in the system can acquire the same latch for at least the duration of this delay. In contrast to latch-based algorithms, *latch-free* algorithms provide strong theoretical progress guarantees which, at minimum, ensure that no thread is blocked due to the delay or failure of other threads [40,41,43]. Several research papers have therefore made the argument that latch-free algorithms scale better than or outperform latch-based algorithms [25, 44, 51, 52].

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

This paper argues that latch-free algorithms' theoretical guarantees are mostly *irrelevant* to performance and scalability on multi-core hardware. The most important factor that influences the scalability of a synchronization mechanism is its ability to avoid contention on global memory locations, irrespective of whether the mechanism is latch-based or latch-free. We argue, and show experimentally, that when latch-free algorithms' theoretical guarantees do become relevant, the performance problems in latch-based algorithms are not fundamental to their use of latches. Instead, the performance problems arise due to inefficient allocation of resources, such as using more OS threads than available CPU cores.

This paper highlights the oft under-appreciated fact that latch-free algorithms generally require idiosyncratic memory management mechanisms. Latch-free memory management mechanisms add complexity and overhead relative to latch-based algorithms. These mechanisms are often ignored or omitted in published descriptions of latch-free algorithms, and have consequently been a source of bugs in implementations of these algorithms [4, 6].

We perform a set of microbenchmarks that make an apples-to-apples comparison between a latch-free algorithm and three classes of latch-based algorithms. We find that while the latch-free algorithm does outperform simple busy-waiting latches, the results in comparison to more sophisticated backoff latches are mixed. Furthermore, we also find that the latch-free algorithm is never able to outperform a scalable queuing latch.

This paper is not intended to make a case *against* latch-free algorithms. Instead, we show that the argument for the superior scalability of latch-free algorithms is more nuanced than a scan of the current literature on multi-core database architectures would suggest (often exacerbated by the fact that papers compare latch-free algorithms against inefficient coarse-grained latching algorithms). We highlight the various factors that influence the performance and complexity of latch-free algorithms, and hope that this discussion will inform future research and database system implementations.

2. LATCH-FREE ALGORITHMS

Latch-free algorithms provide strong theoretical guarantees that distinguish them from latch-based algorithms. These guarantees pertain to the progress that threads can make during concurrent execution. While several progress guarantees exist [40,41,43]; at minimum, all of these guarantees ensure that no thread is blocked due to the delay or failure of other threads. Latch-based algorithms make no such guarantees. Latch-based algorithms guarantee correct concurrent execution of threads via *mutual exclusion*; in order to access a shared data-structure, a thread acquires a latch, which prevents other threads from simultaneously accessing the data-structure.

This section makes the case that the scalability and performance of a synchronization mechanism is dependent on its avoidance of

contention on global memory locations (Section 2.1). This is far more important than any progress guarantees made by the mechanism. Furthermore, the progress guarantees of latch-free algorithms can lead to *increased* overhead due to memory management issues not present in latch-based algorithms (Section 2.2) and other areas of additional complexity (Section 2.3).

2.1 Scalability

2.1.1 Synchronization performance

Modern database systems exploit parallelism in multi-core hardware by employing some form of multi-threading [1]. The threads in a system exchange information via shared data-structures. If multiple threads are allowed to concurrently access to a shared data-structure without any coordination, then the data-structure is susceptible to corruption due to race conditions. In order to prevent race conditions, threads must *synchronize* their accesses to shared data-structures [42].

Latches are an explicit form of synchronization, which are used to ensure that only one thread can ever obtain exclusive access to a data-structure. A latch typically consists of a single word in memory, whose value indicates whether or not a thread currently holds the latch. Threads use various combinations of reads and writes in order to acquire a latch (a specific combination yields a particular latching algorithm). In a latch-free algorithm, threads use atomic instructions to ensure that they correctly update or read a shared data-structure. These atomic instructions are executed on one or more words in shared memory. Threads in both classes of algorithms thus read and write one or more shared words in memory in order to correctly synchronize their access to shared data-structures. The performance of both classes of algorithms is therefore tied to the performance of concurrent reads and writes against a single word in memory.

There are two rules of thumb that determine the performance of concurrent reads and writes to a particular memory location on multi-core hardware [12,19,24,45]. First, atomic instructions, such as `cmp-and-swp` and `xchgq`, that write (or attempt to write) a particular word in memory are executed serially. Thus, if several cores concurrently attempt atomic update instructions on a particular word in memory, the time taken to process these instructions is proportional to the number of writing cores. Second, the new value of a recently written word in memory is serially propagated to reading cores. That is if a core writes the value of a word in memory, and several other cores read the value of the word, the new value of the word will propagate to the reading cores in time proportional to the number of readers.

Several latching implementations are unscalable under contention because they interact badly with the two characteristics of multi-core hardware above. In the simplest implementation of a latch, cores repeatedly attempt to atomically test-and-set the value of a word in memory from 0 to 1 (performed via an `xcghq` on x86 architectures). An atomic test-and-set unconditionally sets a memory to a specified value and returns the previous value of the word. If a core's test-and-set returns 0, then it means that the value of the word successfully transitioned from 0 to 1 due to the core's test-and-set. If a core's test-and-set returns 1, then it means that the previous value of the word was 1, and hence that another core has already acquired the latch. Performing test-and-sets in a tight loop puts pressure on the memory controller associated with the latch word and increases traffic across NUMA nodes. This can delay non-conflicting memory requests by cores not executing the test-and-set, including the core executing the critical section. Furthermore, in order to release the latch, the latch holder must atomically change the value of the latch word from 1 to 0. This write must compete with the test-and-sets performed by threads trying to acquire the latch. This delay in releasing the latch effectively increases the length of the critical section.

In order to rectify these issues, Segall and Rudolph proposed an enhancement to test-and-set latches; instead of repeatedly performing a test-and-set on the latch word in a tight loop, cores attempting to acquire the latch could first *read* the value of the word, and only perform a test-and-set if the tested value is 0 [67]. This latch was termed a test-and-test-and-set (TATAS) latch. TATAS latches allow cores to spin on locally cached copies of the latch word while another core holds the latch. TATAS latches seem to address both issues with simple test-and-set latches; since cores first spin on locally cached values of the latch word, they avoid generating pressure on memory controllers and NUMA interconnects. In addition, latch release does not have to compete with test-and-set requests by cores attempting to acquire the latch.

Unfortunately, the above benefits only apply to lengthy critical sections. Anderson showed that if a critical section is relatively short, then the performance of the TATAS latch is dominated by transient behavior which occurs while the latch changes ownership across cores [12]. In particular, if a core C_0 releases the latch, then several cores C_1, C_2, ..., C_m will notice this change. The first of these cores to perform a subsequent test-and-set will then take ownership of the latch (say C_1). However, this ownership change will not prevent C_2, ... C_m from performing unsuccessful test-and-sets. This causes a transient flood of test-and-sets requests. Only when every one of C_2, ..., C_m has finished performing an unsuccessful test-and-set does the system quiesce. If this time to quiesce is comparable to the critical section length, then TATAS latches suffer from the same scalability problems as simple test-and-set latches.

The underlying problem with test-and-set and TATAS latches is that threads spin on a single *global* memory location. Spinning on a global location can cause serious scalability bottlenecks in latching algorithms that perform atomic modifications or reads on global locations. There exist two mechanisms to avoid spinning on global locations:

Backoff mechanisms in which each atomic modification or read is separated by some number of `noop` instructions. The backoff between successive iterations is often increased via an exponential distribution [12]. Another commonly used backoff mechanism is to rely on the operating system to deschedule threads. This backoff mechanism is used in the GNU C library's Pthread mutex implementation [2].

Scalable latching data-structures in which threads spin on thread-local or core-local data-structures. Spinning on local data-structures prevents the cost of spinning operations degrading with core counts. The most famous example of such a latching algorithm is the MCS latch [59]. The MCS latch constructs an explicit queue of threads waiting to acquire the latch. Each thread has an associated queue node, and this queue node is appended to a queue using an atomic operation (nodes are enqueued using a single `xchgq` instruction). The queue node at the beginning of the queue corresponds to the current latch holder. If a thread's appended queue node is not the first node in the queue, the thread spins on a flag in its local queue node. When the latch holder completes its critical section, it sets the flag in the next queue node. The corresponding thread then notices this change and begins executing the critical section. MCS latches

[1]Note that both multi-processing and multi-threading are equivalent for the purposes of this discussion.

avoid the scalability bottlenecks of conventional latch implementations because threads do not read or write a shared memory location in a loop; when the latch changes ownership, a single thread writes the next thread's queue node flag, and each thread spins on its local queue node's flag. In contrast, conventional latching implementations permit multiple threads to write to a single global memory location (via test-and-sets) in a loop, and are subject to slow downs because these test-and-set requests are serialized, and induce cache coherence traffic to invalidate and reload cache lines on cores which read the value of the latch word [12, 19, 24, 45].

Unlike latch-based algorithms, threads in latch-free algorithms never obtain mutually exclusive access to a shared data-structure. Threads in a latch-free algorithm only use atomic instructions to make their writes atomic. In the vast majority of latch-free algorithms, threads *speculatively* read shared state, perform some local computation based on this speculative read, and attempt to atomically "commit" the computation based on the speculative read [38, 44, 52, 62]. In the time between the thread's speculative read and attempt to commit, another thread may have invalidated the thread's read due to a conflicting update. Threads typically use the `cmp-and-swp` instruction to validate that the shared state did not change values.

The `cmp-and-swp` instruction takes three arguments, the address of a word in memory, the value which word is expected to contain, and a new value. `cmp-and-swp` checks that the word's value is equal to the old value, and if so, swaps the old value with the new value. If the word's value is not equal to the old value, the instruction leaves it unchanged.

Speculative latch-free algorithms are prone to the same scalability bottlenecks as latching algorithms. If updates fail often due to contention, then threads will repeatedly retry the operation. The repeatedly retried `cmp-and-swp` instructions are serialized, and lead to slowdowns because they are executed by threads that successfully speculate, as well as those whose speculation fails. These serialized instructions causes failures to slow down the successes. As a consequence, the "conflict window" of a speculative operation effectively increases. If the increase in the "conflict window" is comparable to the length of the speculative computation, then this effect is akin to increasing critical section size in TATAS latches.

Due to the optimistic nature of many latch-free algorithms, and the well-known costs of optimism (such as copying overhead — see Section 2.2.1), it is tempting to conclude that the differences between speculative latch-free and non-speculative latch-based algorithms are analogous to the differences between optimistic and pessimistic concurrency control in database systems [49]. However, this paper aims to show that the *opposite* is closer to the truth. Speculative latch-free algorithms and latch-based algorithms are more alike than different because both classes of algorithms perform updates to shared memory locations. As a consequence, under contention, both classes of algorithms are governed by the same underlying hardware performance characteristics. In contrast, the differences between optimistic and pessimistic concurrency control in database systems arise due to their contention handling mechanisms — pessimistic concurrency control blocks the execution of a transaction in the presence of another conflicting transaction, while optimistic concurrency control aborts transactions upon detecting conflicts. Optimistic and pessimistic concurrency control are thus governed by a different set of tradeoffs; optimistic concurrency control wastes resources in a fully loaded system under high contention, while pessimistic concurrency control produces unnecessarily conservative schedules in an under-loaded system under low contention [10].

Another point to note is that the techniques used by scalable latching implementations cannot be directly used by latch-free implementations. Scalable latching implementations effectively determine the order in which threads can take ownership of a latch a priori. If a thread is delayed after having determined its priority, then every later thread of lower priority is delayed. However, latch-free algorithms must guarantee that the delay or failure of a particular thread *never* prevents other threads from making progress. As a consequence, pre-determining the order in which threads execute a critical section is incompatible with latch-free algorithms' theoretical guarantees, and can, at best, only be used as an auxiliary contention handling mechanism; when using pre-determined thread priorities a latch-free algorithm must have a way to "time-out" of the pre-determined order and fall-back to using `cmp-and-swp` operations [26].

2.1.2 Scheduling requests

Database systems use the notion of *multi-programming levels* (MPLs) to determine the maximum number of requests that can simultaneously execute. Database systems typically implement MPLs by assigning each request to an abstract *DB worker*, which corresponds to the execution context of the request within the database. DB workers are then mapped to an operating system execution context, such as a process or thread [39]. The choice of mapping from DB workers to OS contexts can have a significant impact on performance.

If the number of OS contexts exceeds the number of available CPU cores, then at least two contexts will be multiplexed over a single CPU core. The scheduling of OS contexts is handled by the operating system, which assigns a fixed time slice to each context, and preempts a context when its slice expires. Despite proposals for workload-aware schedulers [13], operating systems are usually unaware of whether preempted contexts have acquired latches. A context may therefore be preempted *while* it is holding a latch.

Most (but not all) database systems assign each DB worker to a single OS context (by either creating a new context or maintaining a pool of free contexts), and rely on the OS to timeslice contexts on CPU cores [39]. In addition, database systems traditionally use MPLs far higher than the number of available CPU cores. They therefore typically have more DB workers than available CPU cores. In database systems in which there is a 1:1 correspondence between DB workers and OS contexts (including most widely-used database systems), the number of OS contexts significantly exceeds the number of available CPU cores. This can lead to thrashing due to latch holder preemption.

In this traditional database system architecture, the progress guarantees of latch-free algorithms are extremely valuable because a preempted context will never block or delay the execution of other contexts in the system (Section 2). Prior research has therefore (correctly) advocated that latch-free algorithms offer a "drop-in" solution to the preemption problem.

However, the root cause of the preemption problem is *not* database systems' use of latches, but rather that database systems use more OS contexts than available CPU cores and their reliance on an OS with insufficient knowledge about user-level synchronization to schedule these contexts. There is no fundamental reason that forces database systems to implement multi-programming by assigning requests to unique contexts. Instead of assigning requests to unique contexts, a database system could itself implement a scheduling mechanism in user-space without relying on OS support [39]. This scheduling mechanism could ensure that it never uses more contexts than available CPU cores. Several research prototypes and new main-memory DBMS products already use this process-

ing model [31,37,65,66,72,74,75]. Furthermore, database systems have a long tradition of implementing user-level scheduling of contexts (via DBMS threads) in environments where OS support for multi-processing was non-existent or inefficient [71].

Researchers in the software transactional memory (STM) community have also made the case that a system should limit the number of contexts it uses to the number of available CPU cores [30]. Early STM algorithms were designed to be non-blocking or wait-free so as to be robust to unexpected sources of delay beyond the control of the application (such as thread preemptions and page faults) [34]. However, more recent STM algorithms forego non-blocking and wait-free synchronization, and instead use latches to synchronize conflicting transactions [27, 28, 30]. Latches simplify STM algorithms and permit the use of important optimizations, such as in-place updates, which latch-free algorithms preclude. (Section 2.2 discusses some of these issues in detail.)

2.2 Memory management

Latch-based algorithms do not permit a thread access to a data-structure if one or more conflicting threads are concurrently accessing the data-structure. In contrast, latch-free algorithms cannot restrict a thread from accessing a data-structure due to the presence of conflicting threads, because the restricted thread is unable to make progress if any conflicting thread is delayed. These delays violate latch-free algorithms' stringent progress guarantees (Section 2) [40, 41, 43]. As a consequence, latch-free algorithms must permit threads *unrestricted* access to a data-structure. This requirement has subtle implications on latch-free algorithms' design.

2.2.1 Copying overhead

In a latching algorithm, threads acquire latches to update shared data-structures. A thread which has acquired a latch is guaranteed mutually exclusive access to the data-structure protected by the latch. The thread can therefore perform in-place updates on the data-structure. These updates may temporarily make the data-structure inconsistent with respect to program invariants [9]. These inconsistencies are safe because the latch prevents other threads from accessing the data-structure, and hence noticing temporary inconsistencies due to in-place updates.

In contrast, latch-free algorithms must permit a thread unrestricted access to a data-structure, even while other threads attempt conflicting reads or writes. Threads must always be permitted unrestricted access to a data-structure because of the stringent progress guarantees latch-free algorithms provide. In order to allow threads to make progress regardless of the presence of conflicting threads, the state of the data-structure must *always* be consistent. As a consequence, in order to update a complex data-structure, threads must make a copy of a data-structure (or a portion of a data-structure), and perform their updates against this local copy. These updates are made visible to other threads via an atomic instruction [11, 36, 43, 56].

Latch-free algorithms that perform updates on complex data-structures must therefore pay the extra cost of copying a portion of a data-structure. Furthermore, if the algorithm in question is speculative, then the cost of copying the data-structure extends the duration of conflict window in which the update has a chance of failing (Section 2.1.1). Finally, operating on copies of data-structures can lead to worse cache utilization than in-place updates.

2.2.2 Garbage collection

Since latch-free algorithms permit multiple threads to simultaneously operate on an object, if an object is deleted by a thread, then its memory cannot be immediately freed to the operating system or allocator. This is because one or more threads may still be accessing the deleted object. Latch-free algorithms on dynamic data-structures therefore typically use a form of deferred memory reclamation. Examples of deferred memory reclamation include hazard pointers [60] and epoch-based reclamation [58].

There are two problems associated with deferred memory reclamation. First, they impose extra overhead in order to determine when an object can be safely reclaimed. It should be noted that some techniques, such as the epoch-based mechanism used in read-copy-update [58], have very low overhead. However, the choice of reclamation technique is not independent of the algorithm [44].

Second, and more importantly, deferred memory reclamation cannot be used as a black box, in the way that conventional memory allocators are used. Instead, memory reclamation logic is *algorithm dependent*, and therefore entangled with the implementation of a latch-free algorithm. For instance, the addition of correct memory reclamation to Michael and Scott's lock-free queue [62] requires non-trivial changes to the algorithm itself [60].

2.2.3 Memory re-use

If a thread uses `cmp-and-swp` instructions for correctness, it may miss concurrent updates by other threads. Consider the following sequence of events. Thread T reads the value A from a word in memory. Thread T' changes the value of the word from A to B, and then back to A. If T then attempts to `cmp-and-swp` the value of the word based on its earlier read, it will succeed despite the fact that the word's value changed twice: from A to B, and back to A. This behavior, known as the *ABA problem*, can lead to subtle bugs if correctness depends on the the fact that no intervening updates occurred between T's read and its `cmp-and-swp` [42].

The ABA problem typically manifests in latch-free algorithms on pointer-based data-structures, such as linked-lists and queues. For instance, consider a latch-free implementation of a sorted linked-list [38]. In order to insert a new node with value 7 (N_7) in the linked-list, a thread traverses the list until it finds an appropriate pair of adjacent nodes. Suppose the pair of adjacent nodes contain values 5 and 9 (N_5 and N_9). The thread performing the insertion sets N_7's next pointer to reference N_9. Next the thread attempts to atomically set N_5's next pointer to N_7 while validating that N_5 still points to N_9 using a `cmp-and-swp` instruction. However, if N_5 is deleted by another thread and then re-inserted with new value 8, N_7's `cmp-and-swp` will still succeed. This is because N_7's `cmp-and-swp` finds that N_5's next pointer still points to N_9. However, this insertion renders the linked-list inconsistent because nodes are no longer sorted.

In the above example, the unfortunate sequence of events occurs because N_5 is re-used between the time N_7's thread performs a read and `cmp-and-swp` [42]. Preventing the ABA problem requires a mechanism that makes freed memory available to threads after no references to the freed memory can possibly exist. The ABA problem is subtly different from the garbage collection problem; garbage collection ensures that memory is not freed too early, while ABA prevention ensures that memory is not *re-used* too early.

2.3 Complexity

Latch-free algorithms are notoriously complex to specify, let alone implement. Indeed, even experts have designed incorrect algorithms that have required corrections to be incorporated over time. For instance, Valois' lock-free linked list algorithm [76] was shown to contain a race condition [61]. Michael and Scott's lock-free queue algorithm [62] contained two errors, identified and rectified two years later [63].

2.3.1 Modularity

Researchers have argued that latch-free algorithms can simplify the design of operating systems. Operating systems are susceptible to deadlocks if interrupt handler code and non-interrupt handler code acquire the same latch [9]. Since latch-free algorithms can never lead to deadlocks, researchers and practitioners have proposed using latch-free algorithms to simplify interrupt handler code [20, 36].

Fortunately, database system threads and processes never have to deal with interrupts. Latch-free algorithms therefore do not provide the same modularity benefits to databases as they do to OS kernels. Indeed, one could argue that latch-free algorithms *decrease* modularity in non-interruptible systems, such as databases, because they require idiosyncratic memory management code (Section 2.2).

2.4 Discussion

While the focus of this section has been the limitations and overheads of latch-free algorithms, there certainly exist scenarios where latch-free algorithms may provide better scalability than latch-based algorithms. For instance, certain latch-free algorithms permit multiple threads to make changes to a data-structure concurrently [38, 44, 76].

In general, developers and architects should determine how their algorithms interact with the performance characteristics of multi-core hardware (Section 2.1.1). Simply converting a latch-based algorithm to a latch-free algorithm is rarely a recipe for success. Indeed, there exist concurrent algorithms that *combine* latches with and synchronization-free operations to good effect. For example, the read-copy-update technique [58] (widely used in the Linux kernel) and the Masstree main-memory index structure [57] both update no meta-data for reads but use latches for writes.

3. CASE STUDY: TREE-BASED INDEXES

Indexes are well-known to be an important component of database systems. They allow fast access to database records, and typically implement an interface for inserting, deleting, and searching index entries. At any point in time, multiple insert, delete, and search requests may be concurrently executing against an index. Indexes must therefore support employ synchronization mechanisms to correctly order concurrent requests.

Tree-based indexes, such as B$^+$trees, are an important class of index because they provide an *ordered* record access method [22]. Ordered access methods can be used to evaluate range predicates and scans. B$^+$trees provide an associative mapping from index values to sets of records. Like all tree-based data-structures, B$^+$trees are hierarchical. Leaves store a set of record-identifiers corresponding to particular index values, while internal nodes only store meta-data to navigate to index values at the leaves [22]. Every B$^+$tree consists of a single root node, and every other node is accessed via the root. This hierarchy makes it challenging to implement B$^+$trees' interface in a scalable manner — every updater and reader must traverse the tree from the root, and must therefore synchronize their access to the root. The root and, in general, nodes higher up in the tree can therefore turn into scalability bottlenecks. This section discusses the design of concurrency control mechanisms for B$^+$trees that address the scalability challenges above.

3.1 Contention for logical locks

The most well known mechanism for correctly synchronizing concurrent B$^+$tree operations is *latch coupling* [70]. Starting from the root, a request acquires a latch on a node, determines a child node to follow, and recursively continues this process for the child node. A search acquires intention-shared (IS) latches on nodes, and releases a node's latch once a child node's latch is acquired. An update (insert or delete) acquires shared-intention-exclusive (SIX) latches on nodes. An updater holds SIX latches on a sequence of consecutive internal nodes until it is certain that the nodes are safe from modification [70]. If an operation attempts to insert a new entry in a full leaf node, the leaf node is split into two nodes. This causes an insertion to occur in the parent node, which in turn may cause the parent to split if it is full, and so forth. A node may similarly be deleted when the number of elements it contains is below a threshold. Update requests convert their SIX latches to exclusive (X) latches for every internal node that needs to be updated due to the insertion or deletion of child nodes.

SIX latch requests are compatible with IS requests, but incompatible with other SIX requests. Thus, an updater allows readers to simultaneous access to a node, but prevents other updaters from accessing the node. This latching strategy is pessimistic; updaters acquire SIX latches in *anticipation* of modifications and thus block other updaters from accessing the node, even though the node may never actually be modified. It should be noted that because of B$^+$trees' hierarchical organization, the likelihood of a internal node being modified due to a deletion or insertion of a child node decreases *exponentially* from leaf to root. As a consequence, SIX latches on "higher up" internal nodes, such as the root, are mostly held for short durations, typically only while the updater checks whether child node is full. Nevertheless, even short duration SIX latches on the root can significantly impact the performance of B$^+$trees. Indeed, in their B$^+$tree performance study, Srinivasan and Carey found that SIX latches can significantly deteriorate performance even when the number of updates in a workload is much smaller than the number of searches [70].

Subsequent B$^+$tree algorithms were designed to avoid blocking requests on the root node, even for short durations. In these algorithms, updaters descend to the appropriate leaf using the same latch mode as searches (in S or IS modes). Upon reaching the leaf and detecting the need for node insertion or deletion, updaters perform internal node modifications moving from leaves to higher up nodes. Based on the techniques they use to correctly interleave tree descent and bottom-up modification requests, these algorithms can be classified into two categories. First, those such as ARIES/IM, which use a form of optimistic concurrency control to synchronize reads by descending requests and bottom-up modifications [64]. Second, those such as B$^{\text{link}}$trees, which maintain extra information in internal nodes so that reads can always correctly navigate the tree [50].

3.2 Contention on shared memory

In both ARIES/IM and the B$^{\text{link}}$tree, threads descending the tree latch couple their way down to leaves using shared latches. As a consequence, both algorithms permit significantly more concurrency than B$^+$tree algorithms in which descending update requests employ SIX latches [70]. On today's multi-core hardware, however, read latch acquisition is not a negligible cost in the presence of contention. Even though individual operations request the same compatible latch mode, each operation causes some modification of shared internal latch meta-data, such as a counter representing the number of readers [5, 66]. Since every operation traverses the tree via the root, this internal meta-data is updated on every operation, even if no pair of operations actually conflicts. If multiple requests concurrently attempt to traverse the tree, then the time to update the root latch's meta-data is proportional to the number of concurrent requests (Section 2.1). Modern B$^+$tree indexing algorithms are designed to avoid this frequent synchronization, while

using bottom-up algorithms for tree modifications (in the spirit of B^{link}trees [50] and ARIES/IM [64]).

Cha et al. proposed an optimistic reader-writer synchronization mechanism for B^{link}trees, OLFIT [21]. In OLFIT, requests acquire exclusive latches to update a node and increment a node-specific version number. To read a node, a request waits for the latch to be released, reads the version number, and optimistically reads the node's contents. This read is then validated by checking that the node's version number is unchanged. Reading a node, performed by every request while descending the tree, therefore requires no writes to shared memory locations. Furthermore while node updates acquire an exclusive latch, nodes higher up in the tree, such as the root, are updated exponentially less often than leaves. As a consequence, OLFIT eliminates frequent synchronization on the root. OLFIT's use of timestamps to validate optimistic reads is a powerful design pattern for scalable reader-writer synchronization. Timestamp-based validation forms the basis of several recent systems, including Masstree [57], a main-memory multi-core index, and Silo [75], an optimistic main-memory multi-core database.

In a similar vein, the Bw-tree uses multi-versioning to eliminate the need for reads to update shared meta-data [52]. The Bw-tree maintains a node's state as a linked-list of immutable "deltas", which must be combined to produce the state of the node. Requests update a node by encoding the update in a new immutable delta, and appending the delta to a node's linked-list. Requests read a node's state by combining the linked-list of updates. The linked-list is periodically compacted to bound the overhead of combining deltas. Bw-tree requests can read a node's state without interacting with updates — reads construct a snapshot of a node state by following a linked-list of immutable deltas, while updates perform a latch-free append on the tail of the linked-list. However, the increase in concurrency comes at the expense of increased read overhead due to pointer dereferences and CPU cycles involved in constructing a node's state from a linked-list of deltas.

The OLFIT and Bw-tree algorithms differ due to their use of latch-based and latch-free mechanisms, respectively. This difference *does not* impact the scalability of either algorithm. Instead, both algorithms are scalable because requests avoid updating frequently accessed shared memory, such as the root node, while descending the tree. Both algorithms can therefore avoid the scalability limitations of sequential updates to a single memory location at the hardware-level (Section 2.1).

4. EXPERIMENTAL EVALUATION

4.1 Microbenchmarks

This section compares the performance of a latch-free synchronization algorithm to three classes of latching algorithms; a busy-waiting spinlock (TATAS spinlocks), backoff latches which put threads to sleep under contention (Pthread mutexes), and scalable latches which avoid spinning on a global memory location (MCS latches). We evaluate each synchronization primitive on a benchmarking framework which precisely controls amount of parallel and serial work each thread must perform. Threads execute in two phases — a serial and parallel phase. Both serial and parallel phases consist of a fixed number of `noop` instructions. The duration of the serial and parallel phases is varied by changing the number of `noop` instructions to execute.

The synchronization algorithms differ in the mechanism they use to execute the serial phase. In the latching algorithms, each thread can only execute the serial phase if it owns the corresponding latch. Threads in the latch-free algorithm use the approach proposed in Herlihy's generic methodology for constructing latch-free objects [43]. Each thread reads the value of a counter prior to executing the serial phase, speculatively executes the serial phase, and then attempts to atomically `cmp-and-swp` the old counter value with its incremented value. The thread successfully executes its serial phase if the `cmp-and-swp` succeeds. If the `cmp-and-swp` fails, the thread reads the value of the counter again and attempts to re-execute the serial phase. The latch-free algorithm is representative of other implementations of linearizable latch-free data-structures [41,44,52,62]. The latch-free algorithm contains an optimization to reduce spurious `cmp-and-swp` instructions executed by a thread. After speculatively executing the serial phase, a thread will first read the value of the counter and check that the value is unchanged before attempting a `cmp-and-swp` instruction [18]. We do not account for latch-free algorithms' memory management overhead (Section 2.2).

We run our experiments on a single 80-core machine, consisting of eight 10-core Intel E7-8850 processors and 128GB of memory. The operating system used is Linux, kernel version 3.19.0-61. We perform three sets of experiments, each corresponding to a particular level of contention; low, medium, and high. The parallel phase in each experiment consists of 200,000 cycles, while the low, medium, and high contention serial phases respectively consist of 100, 1,000, and 10,000 cycles. These serial phase lengths respectively correspond to 0.05%, 0.5%, and 5% of the parallel phase.

4.1.1 *Low contention*

Figure 1 shows the results of the low contention experiment. All algorithms scale perfectly when the number of threads is less than or equal to the number of available CPU cores (note that the x-axis uses a log-scale). Figure 1b shows the latency distribution of each algorithm under 80 threads (the number of threads is equal to the number of available CPU cores) — there no significant difference between the algorithms.

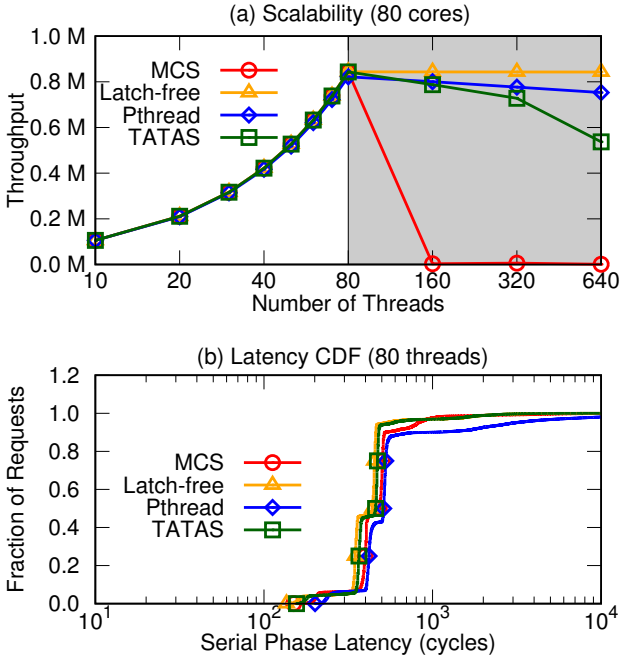

Figure 1: Performance of synchronization algorithms under low contention. Serial phase = 100 cycles. Parallel phase = 200,000 cycles. Serial phase = 0.05% Parallel phase.

When the number of threads exceeds the available CPU cores, we see differences between each algorithm. The MCS latch's throughput completely collapses. The MCS latch constructs a queue of threads waiting to acquire the latch. If any thread in the queue is preempted, even if it does not yet own the latch, then all later threads are delayed until the preempted thread is rescheduled. In contrast, the TATAS latch's throughput does not degrade as significantly because, unlike in the MCS latch, preemptions of threads that do not hold the latch do not affect other threads. The Pthread latch's throughput also degrades with increasing thread count, but outperforms the TATAS latch. The Pthread latch puts threads to sleep in the kernel if they fail to acquire the latch [2]. If a thread is preempted while holding the latch, other threads will fail to acquire the latch and get put to sleep in the kernel. The kernel is eventually left with no choice but to execute the thread which holds the latch because other threads get put to sleep before their scheduling quantum expires. This has the effect of diminishing the impact of preemption on the Pthread latch's throughput.

In contrast to the latch-based algorithms, the latch-free algorithm's throughput is unaffected by increasing the number of threads beyond available CPU cores. Throughput does not decrease because thread preemption in the latch-free algorithm never impedes other threads. Throughput does not increase because the server's physical CPU resources are fully utilized at 80 threads. (The server contains 80 CPU cores.) The use of more threads, beyond 80, therefore does not increase throughput.

4.1.2 *Medium contention*

Figure 2 shows the results of the medium contention experiment (serial phase is 0.5% of the parallel phase).

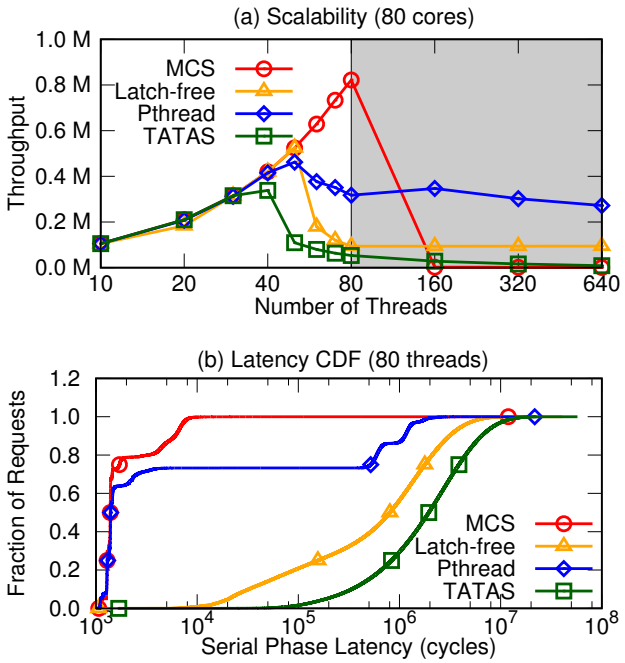

Figure 2: Performance of synchronization algorithms under medium contention. Serial phase = 1,000 cycles. Parallel phase = 200,000 cycles. Serial phase = 0.5% Parallel phase.

The TATAS latch scales until 40 cores, but its throughput drops dramatically thereafter; its throughput at 80 cores is less than half its throughput at 10 cores. The reason for the drop in throughput is the transient flood of `xchgq` instructions executed by cores when the latch changes ownership. These `xchgq` instructions impede latch holding threads when they attempt to release the latch, effectively increasing the length of the serial phase. Furthermore, the effect of spurious `xchgq` instructions gets worse with increasing core count because more cores contribute to the transient flood of `xchgq` instructions.

The latch-free algorithm scales to 50 cores, but like the TATAS latch, its throughput drops significantly thereafter. The reason for the drop in throughput is also similar. Threads read the value of a counter before executing the serial phase, and validate that the value of the counter is the same at the end of the serial phase using a `cmp-and-swp` instruction. If multiple threads attempt to execute the n^{th} iteration of the serial phase, then only one thread will succeed. However, some of the threads will also attempt spurious `cmp-and-swp` instructions because they do not notice the change in the counter value (despite the optimization which checks the value of the counter before attempting the `cmp-and-swp`). These spurious `cmp-and-swp` instructions will delay threads attempting to execute later iterations because atomic instructions on the same memory word are executed sequentially. As in the TATAS latch, this effectively increases the length of the serial phase.

The Pthread latch's throughput does not collapse after peaking. This is because the Pthread latch has a built-in contention handling mechanism. Threads attempt to acquire the latch with two successive `cmp-and-swp` attempts; if they fail, they are put to sleep in the Linux kernel [2]. The Pthread latch's latency distribution shows the effect of backing-off (Figure 2b). About 75% of serial phase executions occur without threads backing off, while the other 25% are executed by threads that are put to sleep in the kernel — indicated by the two distinct latency profiles of requests. The variance in latency of serial phases executed without backoff (at the left-hand side of the graph) occurs because of competing `cmp-and-swp` requests.

The MCS latch scales perfectly when the number of threads does not exceed the number of CPU cores. When acquiring the latch, threads perform a single `xchgq` instruction on the latch word and then spin on a local cache line [59], avoiding the overheads associated with transient floods of spurious `cmp-and-swp` instructions and cache invalidations (Section 2.1.1). The MCS latch's latency distribution has a much smaller mean and variance than other synchronization algorithms. The differences in latency across the last 20% of requests is due to queuing delay.

4.1.3 *High contention*

Figure 3 shows the results of the high contention experiment. We set the size of the serial phase to 5% of the parallel phase. The MCS latch's throughput increases until 20 cores and then plateaus. The reason is a lack of parallelism in the workload (1/20th of each transaction is serial).

The TATAS and latch-free algorithms exhibit the same behavior as in the medium contention experiment. Throughput increases until 20 cores and then begins to decrease. The difference between both lines occurs because the TATAS latch's transient behavior when the latch changes ownership *does not* depend on the length of the critical section. This is because cores using the TATAS latch spin in a tight loop. In contrast, cores using the latch-free algorithm speculatively execute the serial phase between attempts to commit via `cmp-and-swp` instructions.

The Pthread latch does not outperform the latch-free algorithm's throughput in this experiment. As Figure 3b indicates, a much larger fraction of threads are put to sleep in the Linux kernel (about 75%).

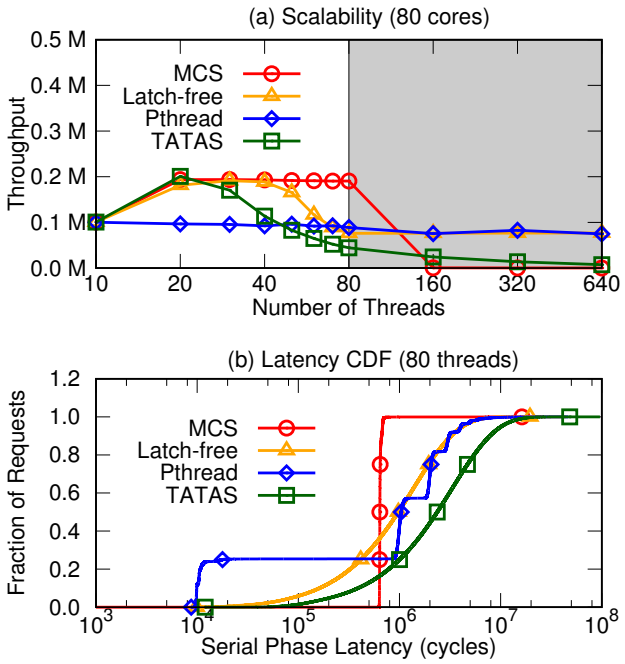

Figure 3: Performance of synchronization algorithms under high contention. Serial phase = 10,000 cycles. Parallel phase = 200,000 cycles. Serial phase = 5% Parallel phase.

Figure 3b also shows that the MCS latch provides more reliable performance than any other algorithm. This is because MCS latches determine the priority of threads prior to the execution of the serial phase. The latency trend has an important implication for concurrent applications; if the progress of a system depends on stragglers (for instance, algorithms based on barrier synchronization [42]), then non-scalable synchronization algorithms can cause serious degradation in performance.

4.2 Queuing experiment

This section shows the effect of avoiding repeated updates against a single shared memory location on a concrete example. We built two versions of a concurrent queue, one latch-based, the other latch-free. Both versions have been used in recently published systems. Jung et al., and Wang and Kimura used the latch-based concurrent queue to construct lists of logical locks in a multi-core optimized lock manager [47, 78]. The latch-based queue is also used to determine thread priorities in the MCS latch [59]. Levandoski et al. use the latch-free version of the queue to construct linked-lists of delta updates for Bw-tree nodes (Section 3.2) [52].

```
1  xchg_enq(Node **tail, Node *qnode):
2    qnode->prev = INVALID
3    old_tail = xchgq(tail, qnode)
4    qnode->prev = old_tail
```

Figure 4: Pseudocode for latch-based enqueue operations using the `xchq` instruction.

Figure 4 shows pseudo-code for the latch-based enqueue algorithm. The algorithm takes two arguments, a reference to the tail of the list (which is itself a pointer to a node), and a reference to the node to be inserted. The new node's `prev` pointer is first marked as `INVALID` (line 2). The algorithm then atomically changes the tail to point to new node using the `xchgq` instruction (line 3). The `xchgq` instruction returns the prior value of the tail, and the new node's `prev` pointer is then changed to reference the old tail value (line 4). The list is temporarily rendered inconsistent between lines 3 and 4 — after atomically changing the tail to reference the new node (line 3), the node's `prev` pointer does not yet point to the valid prior node. To prevent threads concurrently traversing the list from observing this inconsistency, the new node's `prev` pointer is marked `INVALID` on line 2. Traversing threads spin on any node's `INVALID prev` pointer until it is changed by the inserting thread on line 4. A new node's `prev` pointer effectively serves as an exclusive latch to prevent traversing threads from observing inconsistent state.

```
1  cmpswp_enq(Node **tail, Node *qnode):
2    while True:
3        qnode->prev = *tail
4        if cmpswp(tail, qnode->prev, qnode):
5          break
```

Figure 5: Pseudocode for latch-free enqueue operations using the `cmp-and-swp` instruction.

Figure 5 shows pseudo-code for the latch-free enqueue algorithm. The latch-free enqueue algorithm takes a reference to the list tail and a reference to the new node as input. The algorithm first and optimistically sets the new node's `prev` pointer to the value of the tail. The tail's value is obtained by performing a read from memory (line 3). The algorithm then attempts to atomically insert the new node into the list by using an atomic `cmp-and-swp` instruction. The `cmp-and-swp` atomically compares the latest value of the tail with the new node's `prev` pointer and sets the tail's value to reference the new node if the comparison succeeds (line 4). If the comparison fails, the `cmp-and-swp` does not write the tail, and the algorithm retries the steps above.

We compare the performance of these two algorithms using a simple multi-threaded experiment. Each thread repeatedly enqueues new nodes to the tail of a shared list using one of the algorithms above. On successfully performing an enqueue, each thread waits for a specified duration before attempting the next enqueue. We vary contention in the experiment by varying this duration between enqueue requests. We measure the overall throughput of each algorithm (as the number of enqueues performed per second) under low and high contention.

Figure 6a shows the result of the high contention experiment. In this experiment, the duration between a successful enqueue and the next enqueue on every thread is set to 10,000 cycles. Both algorithms suffer from the scalability bottleneck of frequently updating the tail — `xchgq` and `cmp-and-swp` instructions are executed sequentially against the tail. However, the latch-based algorithm outperforms the latch-free algorithm by nearly 3x at 80 threads. The difference arises because the latch-free algorithm executes both successful and unsuccessful `cmp-and-swp` operations against the tail. There are two sources of unsuccessful `cmp-and-swp` operations. First, several threads may read the latest value of the tail and subsequently attempt to atomically insert their new nodes into the list, but only one will succeed. Second, due to hardware delays in propagating changes in the tail's value (Section 2.1), some threads may read a stale tail value and perform a spurious `cmp-and-swp` which will never succeed. The latch-free algorithm in our microbenchmark evaluation experienced similar

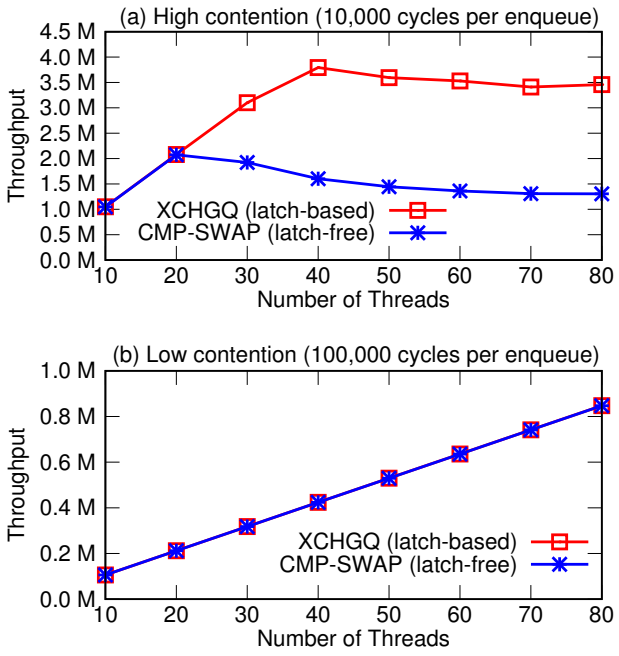

Figure 6: Scalability of latch-free and latch-based queues under high and low contention. Throughput is measured as number of successful enqueues per second.

sources of overhead (Section 4.1). In contrast, the latch-based algorithm performs strictly as many `xchgq` instructions as there are enqueues, and therefore experiences significantly less contention for the tail of the linked-list.

Note that in the latch-based `xchgq` algorithm, threads traversing the list must sometimes pay the cost of waiting for a newly inserted node's `prev` pointer to transition from `INVALID` to a valid reference. However, this cost is minimal because a newly inserted node's `INVALID` pointer is changed in the instruction following the `xchgq` (Figure 4). Furthermore, since a node's `prev` pointer can only be updated by the corresponding inserting thread, the thread experiences no contention while changing a node's `prev` pointer.

Figure 6b shows the result of the low contention experiment. The duration between a successful enqueue and the next enqueue in this case is set to 100,000 cycles. There is no difference in the throughput of the algorithms under low contention.

The experiments in this section show the importance of designing synchronization mechanisms that minimize repeated updates on contended shared memory locations. Repeated updates can cause scalability issues because they are processed sequentially, and therefore increase the amount of sequential execution in a concurrent program. A synchronization algorithm's ability to reduce the number of these sequential operations on shared memory locations is the single biggest factor that influences its scalability.

5. IMPLICATIONS

The results of our experimental evaluation indicate that, at the hardware level, the only factor that affects the scalability of a synchronization mechanism is its ability to avoid repeatedly reading or writing a particular location in memory. This can be achieved by designing synchronization algorithms in which threads spin on different memory locations (as in the MCS latch) or use backoff-based contention management mechanisms (as in the Pthread latch). However, we also found that when a system is over-subscribed, preemptive scheduling can impact the performance of user-space synchronization mechanisms. Based on these observations, this section outlines avenues for future research in the design of multi-core database systems.

5.1 Context scheduling mechanisms

The limitations of user-level scheduling mechanisms arise because context scheduling software underneath the database, such as the operating system or hypervisor, is unaware of user-level scheduling. For instance, in our experimental evaluation (Section 4), both MCS and TATAS latches were built using user-level mechanisms, which performed badly when latch-holding threads were preempted because the number of available threads exceeded the number of available cores. This is far from a database specific problem. For instance, Microsoft's distributed actor-based programming model, Orleans, employs user-level cooperative scheduling of tasks. The authors of the system explicitly state that Orleans is not intended to be run in a multi-tenant environment [16]. This assumption is often perfectly acceptable, even in virtualized environments. Indeed, Orleans is widely deployed on Microsoft Azure [3]. Furthermore, several classes of Amazon EC2 guarantee that a single virtual core is assigned exclusive access to a hardware hyperthread [1].

The utility of cooperative scheduling for server applications has long been recognized, and has received recent attention from the database research community. For instance, Johnson et al. propose a user-level load control mechanism which detects when a system may be overloaded and dynamically returns threads to the operating system [46]. Giceva et al. propose building custom operating system mechanisms for non-preemptive task-based scheduling [35]. In addition, there exists a rich history of research on operating system support, such as scheduler activations [13], for user-level scheduling. These mechanisms have seen recent adoption in Windows, which supports *User-Mode Scheduling* (UMS), a form of cooperative user-level thread-scheduling [7].

5.2 Message-passing

In general, any concurrent programming model based on communication via shared memory cannot avoid synchronization. If threads can perform conflicting modifications to data, then some form of synchronization – using latching or latch-free mechanisms – is necessary in order to prevent shared data-structures from being rendered inconsistent due to race conditions. The deleterious impact of synchronization on scalability is therefore a fundamental aspect of concurrent programming models based on shared memory.

As an alternative to shared memory, threads can use explicit message-passing as a communication mechanism. The basic idea behind message-passing is to avoid sharing state across multiple threads. Instead, each thread maintains local state, and only that thread is permitted to read or update that state. Explicit message-passing can circumvent the synchronization overhead associated with shared-memory communication. The problem with shared memory synchronization is that its overhead gets worse with increasing core counts. On the other hand, message-passing explicitly bounds synchronization overhead.

As a consequence of its attractive properties with respect to shared memory, several multi-core systems have been built with the explicit goal of using message-passing as a communication mechanism. The Barrelfish operating system [15] runs independent kernel instances on each CPU core in a multi-core server. Remote

core locking [55] is a user-space library which employs a subset of a machine's cores as "server" cores. A particular critical section is assigned to a server core, which executes critical sections on behalf of threads. Ren et al. use explicit message-passing to avoid synchronization on concurrency control meta-data in database systems [66].

One drawback of message-passing is queuing delay of messages on cores. This queuing delay can impact the overall performance of a system, if a particular request must go through a sequence of multiple messages. For instance, Ren et al. found that queuing delay can impact the performance of higher level abstractions built on top of message-passing, such as concurrency control [66].

Finally, the distinction between message-passing and shared--memory communication is not rigid. There exist several mediums in between. For example, it is possible to limit shared-memory interactions to threads which execute on a single NUMA socket, while using explicit message-passing to communicate across sockets.

5.3 Advanced planning

Fundamentally, synchronization constrains the schedules of conflicting operations on shared data. The precise nature of a valid schedule depends on the application and algorithm. For example, networking algorithms are resilient to stale information by design, and can hence tolerate reading stale information to make routing decisions [58]. Other applications, such as key-value stores may guarantee linearizability for single key operations [57]. The latter imposes more constraints on schedules of conflicting operations, and this difference is in turn reflected in the synchronization mechanisms used by each algorithm. Synchronization *dynamically* constrains the execution of conflicting operations by forcing their corresponding contexts to coordinate using latching or latch-free mechanisms.

A system could alternatively determine valid schedules *prior* to executing operations. Pre-determining schedules eliminates or reduces synchronization overhead when running operations because a valid schedule has already been determined. At a high level, a pre-determined schedule effectively partitions operations into sets, such that operations in two different sets do not conflict. The operations in different sets can therefore be executed concurrently without the need for any synchronization. The PALM B-tree index is an example of an algorithm that employs such a scheduling mechanism [68]. PALM supports normal B-tree index operations, such as search, delete, and insert. PALM accumulates batches of index operation requests, and performs an analysis on the operations in each batch. During its analysis, PALM divides operations into non-conflicting sets, and then assigns a single thread to execute the operations in a particular set. Threads are thus assigned independent pieces of work to obviate any synchronization during insert, delete, and lookup operations. In contrast, a conventional B-tree implementation requires threads to perform some form of synchronization in order to correctly perform updates and lookups [21, 50–52, 57, 70].

Recent work on multi-core concurrency control for database systems also makes use of advanced planning [31–33]. The key insight underlying these systems is that pre-determining schedules can avoid the overheads of concurrency control mechanisms based on locking or optimistic validation. These systems totally order transactions prior to their execution, and then relax the total order into a partial order based on actual conflicts between transactions. The partial ordering relationship between transactions is represented using an explicit dependency graph constructed during an analysis phase *prior* to transactions' execution. These systems use an event-driven task parallel execution model, where ordering constraints between tasks are encoded via the explicit dependency graphs above.

While advanced planning can eliminate or reduce the need for synchronization, its mileage varies depending on the application for two reasons. First, it introduces a tradeoff between scalability and latency — advanced planning improves scalability by reducing synchronization, but creates schedules by *batching* pending operations, which increases latency. This increased latency may hurt overall system performance. For instance, in the B-tree example above, increased latency of index operations may cause logical locks on the corresponding data items to be held for longer or may increase the chances for optimistic validation errors to manifest [66]. Advanced planning therefore typically requires an end-to-end understanding of the impact of increased latency on other unrelated components or higher level abstractions.

Second, in order to construct schedules with opportunities for concurrent execution, advanced planning requires that conflicts between operations can be deduced *prior* to their execution. It should be noted however, that conflicts between operations need not always be *precise*. Advanced planning can only assign concurrent work to tens or hundreds of physical CPUs. The number of non-conflicting sets of operations therefore does not necessarily need to be very large.

The multi-core transaction processing mechanisms above are explicitly designed to work around these two limitations. First, in order to avoid the deleterious impact of increased latency, transaction schedules are created *prior* to their execution. This extra latency involved in creating schedules does not increase the duration for which conflicting transactions are blocked due to conflicts. On the contrary, it permits significantly more concurrency between conflicting transactions than corresponding state-of-the-art concurrency control protocols by permitting the construction of aggressive serializable transaction schedules [31, 32]. Second, to determine whether transactions conflict, these transaction processing mechanisms speculatively execute a subset of transactions' logic or instead employ coarse-grained conflict information, at the granularity of partitions or even entire tables, which can be obtained via static analysis of transactions [32].

5.4 Asynchronous coordination

The best way to avoid synchronization is to use algorithms that do not require it. There is a rich body of work on distributed systems designed to avoid synchronization [14, 23, 53, 54, 69, 73]. These systems carefully constrain the guarantees they provide applications so that they can be implemented using asynchronous coordination mechanisms.

Database system components which do not require synchronous coordination can therefore be implemented using these techniques. An example of this is the physical redo phase in recovery protocols such as Silo's [75, 79]. Physical redo log records correspond to a single database object, and Silo timestamps physical redo log records with sequence numbers. If it encounters multiple log records to redo against the same page, then the latest log record must win. Physical replay of redo log records can therefore be implemented in a coordination-free manner because we can apply the *last-writer-wins* rule to resolve race conditions [77] — the state of a particular object is guaranteed to be deterministically recovered regardless of the order in which log records are processed.

Even if an algorithm cannot be directly implemented using asynchronous coordination, it may be possible to exploit domain-specific knowledge to use asynchronous coordination under restricted settings. A good example of such an algorithm is the read-copy-update

(RCU) read-writer synchronization mechanism used in the Linux kernel [58]. RCU uses context-switch information to determine whether a deleted object has any pending references. When an object is deleted by a particular thread, RCU permits concurrent threads to hold a reference to the object, and *defers* freeing the object's memory until no thread holds a reference to the object. Since threads can only obtain access to an object while they are in the kernel, a deleted object's memory can be reclaimed after every CPU core has performed at least one context switch. RCU eliminates the need for more expensive mechanisms of tracking live references to an object, such as reference counting. Maintaining reference counts can be expensive because it requires writes to shared memory. These writes could turn into a scalability bottleneck on frequently read objects [5, 66].

Recently proposed multi-version databases systems use memory management techniques based on RCU [31, 48]. These systems assign transactions to batches or epochs, and maintain a low-watermark corresponding to the latest epoch such that every transaction in the epoch has finished executing. The memory corresponding to versions of records deleted at or before the low-watermark epoch can be reclaimed because the deleted objects can never be accessed by transactions from epochs that follow the low-watermark.

Centiman is a distributed optimistic database system that validates transactions using asynchronous coordination [29]. Centiman shards database objects across validator nodes, whose only role is to validate transactions. Centiman clients drive transaction execution, and forward transactions to the appropriate set of validators to obtain a commit decision. Each validator checks if a transaction's reads are invalidated by the writes of earlier transactions, and if not, determines that the transaction can commit and remembers the writes performed by the transaction on its partition. The client commits the transaction if every validator determines that a transaction can commit. Validators which determine that a transaction can commit even though it actually aborts (due to conflicts found on other validators), continue to validate later transactions as if the aborted transaction had committed. As a consequence, later transactions may be conservatively aborted even though they could have committed. Importantly, however, this protocol guarantees that Centiman never commits a transaction that should have aborted. Centiman uses an asynchronous protocol to periodically phase out data on validators that locally committed transactions which were actually aborted. Centiman effectively trades off precision during commit processing for avoiding synchronous coordination in conventional commit protocols such as two-phase commit [17].

Although they both solve problems that seem to necessitate synchronous coordination, RCU and Centiman demonstrate that it is possible to exploit domain-specific knowledge to implement efficient asynchronous coordination mechanisms.

6. CONCLUSIONS

Latch-free algorithms usually (but not always) outperform their latch-based counterparts when the number of OS contexts exceeds the number of cores in the system. However, as modern database systems — especially main-memory database systems — move to a process model where there is a one-to-one mapping between OS contexts and processing cores, the progress guarantees of latch-free algorithms are marginalized. Instead they are subject to the same types of synchronization overheads as latch-based algorithms. Our generic latch-free algorithm was never able to outperform a scalable queue-based latching algorithm in such an environment. Furthermore, latch-free algorithms often necessitate idiosyncratic memory management mechanisms and other complexities not present in latch-based systems. We thus caution the database community from rushing to implement latch-free algorithms without a careful investigation of scalable latch-based alternatives.

Finally, we emphasize that designers of scalable multi-core database systems should focus on avoiding frequent synchronization on a single location in shared memory. The scalability of a system is determined by its ability to avoid this frequent synchronization rather than its use of latch-based or latch-free algorithms.

7. ACKNOWLEDGMENTS

We thank Joseph Hellerstein, Hideaki Kimura, Justin Levandoski, Ippokratis Pandis, Julian Shun, and the anonymous CIDR 2017 reviewers for their insightful comments on earlier versions of this paper. This work was sponsored by the NSF under grant IIS-1527118.

8. REFERENCES

[1] EC2 instance types — amazon web services. https://aws.amazon.com/ec2/instance-types/.
[2] The GNU C library (glibc). https://www.gnu.org/software/libc/.
[3] Microsoft orleans. https://dotnet.github.io/orleans/.
[4] Possible typo/bug in the michael and scott queue white paper psuedo-code. https://tinyurl.com/zx7dqsb.
[5] Reader/writer locks and their (lack of) applicability to fine-grained synchronization. https://tinyurl.com/gv7zfjt.
[6] Use-after-free bug in maged m. michael and michael l. scott's non-blocking concurrent queue algorithm. https://tinyurl.com/h4jlutf.
[7] Windows User-Mode Scheduling. https://goo.gl/WJPSL3.
[8] X1 instances for ec2 — ready for your memory-intensive workloads. https://tinyurl.com/hc7bgzz/.
[9] Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2011/xv6.html.
[10] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *TODS*, 12(4), 1987.
[11] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *PODC*, 1992.
[12] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *TPDS*, 1(1), 1990.
[13] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *TOCS*, 10(1), 1992.
[14] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3), 2013.
[15] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.
[16] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, 24, Microsoft Research, 2014.
[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[18] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, CMU Computer Science, 1991.
[19] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *OLS*, 2012.
[20] B. Cantrill and J. Bonwick. Real-world concurrency. *Queue*, 6(5), 2008.
[21] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.
[22] D. Comer. The ubiquitous b-tree. *CUSR*, 11(2), 1979.
[23] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[24] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, 2013.

[25] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[26] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. *CoRR*, abs/1305.5800, 2013.

[27] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, 2006.

[28] D. Dice and N. Shavit. Tlrw: return of the read-write lock. In *SPAA*, 2010.

[29] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.

[30] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[31] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[32] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5), 2017.

[33] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[34] K. Fraser and T. Harris. Concurrent programming without locks. *TOCS*, 25(2), 2007.

[35] J. Giceva, G. Zellweger, G. Alonso, and T. Rosco. Customized os support for data-processing. In *DaMoN*, 2016.

[36] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *OSDI*, 1996.

[37] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[38] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[39] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a database system*. Now Publishers, 2007.

[40] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991.

[41] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *DISC*, 2003.

[42] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[43] M. P. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP*, 1990.

[44] T. Horikawa. Latch-free data structures for dbms: design, implementation, and evaluation. In *SIGMOD*, 2013.

[45] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *VLDBJ*, 23(1), 2014.

[46] R. F. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS*, 2010.

[47] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.

[48] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD*, 2016.

[49] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.

[50] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *TODS*, 6(4), 1981.

[51] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *DAMON*, 2016.

[52] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

[53] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[54] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.

[55] J.-P. Lozi, F. David, G. Thomas, J. L. Lawall, G. Muller, et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[56] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *PVLDB*, 8(11), 2015.

[57] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[58] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *OLS*, 2001.

[59] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1), 1991.

[60] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15(6), 2004.

[61] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR-599, University of Rochester Computer Science, 1995.

[62] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[63] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *JPDC*, 51(1), 1998.

[64] C. Mohan and F. Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *SIGMOD*, 1992.

[65] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[66] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*, 2016.

[67] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *ISCA*, 1984.

[68] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *PVLDB*, 4(11), 2011.

[69] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, Inria – Centre Paris-Rocquencourt, 2011.

[70] V. Srinivasan and M. J. Carey. Performance of b+ tree concurrency control algorithms. *VLDBJ*, 2(4), 1993.

[71] M. Stonebraker. Operating system support for database management. *CACM*, 24(7), 1981.

[72] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[73] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[74] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[75] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.

[76] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, 1995.

[77] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.

[78] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2), 2016.

[79] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, 2014.