Scaling Synchronization in Multicore Programs

ADVANCED SYNCHRONIZATION METHODS CAN BOOST THE PERFORMANCE OF MULTICORE SOFTWARE.

ADAM MORRISON, TEL AVIV UNIVERSITY

Designing software for modern multicore processors poses a dilemma. Traditional software designs, in which threads manipulate shared data, have limited scalability because synchronization of updates to shared data serializes threads and limits parallelism. Alternative distributed software designs, in which threads do not share mutable data, eliminate synchronization and offer better scalability. But distributed designs make it challenging to implement features that shared data structures naturally provide, such as dynamic load balancing and strong consistency guarantees, and are simply not a good fit for every program.

Often, however, the performance of shared mutable data structures is limited by the synchronization methods in use today, whether lock-based or lock-free. To help readers make informed design decisions, this article describes advanced (and practical) synchronization methods that can push the performance of designs using shared mutable data to levels that are acceptable to many applications.

PROS AND CONS OF SHARED MUTABLE DATA

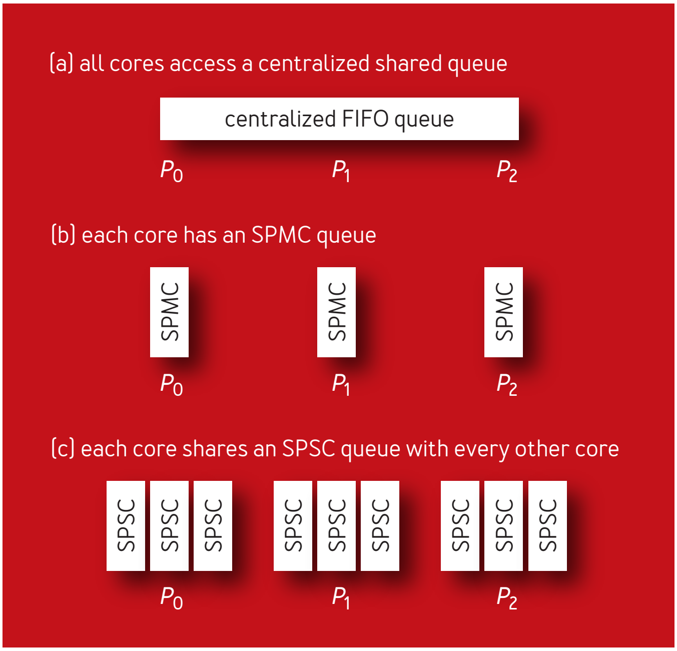
To get a taste of the dilemmas involved in designing multicore software, let us consider a concrete problem: implementing a *work queue*, which allows threads to enqueue and dequeue work items—events to handle, packets to process, and so on. Issues similar to those discussed here apply in general to multicore software design.[14]

Centralized shared queue

One natural work queue design (depicted in figure 1a) is to implement a centralized shared (thread-safe) version of the familiar FIFO (first in, first out) queue data structure—say, based on a linked list. This data structure supports enqueuing and dequeuing with a constant number of memory operations. It also easily facilitates dynamic load balancing: because all pending work is stored in the data structure, idle threads can easily acquire work to perform. To make the data structure thread-safe, however, updates to the head and tail of the queue must be synchronized, and this inevitably limits scalability.

Using locks to protect the queue serializes its operations: only one core at a time can update the queue, and the others must wait for their turns. This ends up creating a sequential bottleneck and destroying performance very quickly. One possibility to increase scalability is by replacing locks with *lock-free*

FIGURE 1: **POSSIBLE DESIGNS FOR A WORK QUEUE**


(a) all cores access a centralized shared queue
centralized FIFO queue
P_0
P_1
P_2
(b) each core has an SPMC queue
SPMC
SPMC
SPMC
P_0
P_1
P_2
(c) each core shares an SPSC queue with every other core
SPSC
P_0
P_1
P_2

synchronization, which directly manipulates the queue using atomic instructions,[1,11] thereby reducing the amount of serialization. (Serialization is still a problem because the hardware cache coherence mechanism[1] serializes atomic instructions updating the same memory location.) In practice, however, lock-free synchronization often does not outperform lock-based synchronization, for reasons to be discussed later.

Partially distributed queue

Alternative work-queue designs seek scalability by distributing the data structure, which allows for more parallelism but gives up some of the properties of the centralized shared queue. For example, figure 1b shows a design that uses one SPMC (single-producer/multiple-consumer) queue per core. Each core enqueues work into its queue. Dequeues can be implemented in various ways—say, by iterating over all the queues (with the starting point selected at random) until finding one containing work.

This design should scale much better than the centralized shared queue: enqueues by different cores run in parallel, as they update different queues, and (assuming all queues contain work) dequeues by different cores are expected to pick different queues to dequeue from, so they will also run in parallel.

What this design trades off, though, is the data structure's *consistency guarantee*. In particular, unlike the centralized shared queue, the distributed design does not maintain the cause and effect relation in the program. Even if core P_1 enqueues x_1 to its queue after core P_0 enqueues x_0 to its queue, x_1 may be dequeued before x_0. The design *weakens* the consistency guarantees provided by the data structure.

The fundamental reason for this weakening is that in a distributed design, it is hard (and slow) to combine the per-core data into a *consistent* view of the data structure—one that would have been produced by the simple centralized implementation. Instead, as in this case, distributed designs usually weaken the data structure's consistency guarantees.[5,8,14] Whether the weaker guarantees are acceptable or not depends on the application, but figuring this out—reasoning about the acceptable behaviors—complicates the task of using the data structure.

Despite its more distributed nature, this per-core SPMC queue design can still create a bottleneck when load is not balanced. If only one core generates work, for example, then dequeuing cores all swoop on its queue and their operations become serialized.

Distributed queue

To eliminate many-thread synchronization altogether, you can turn to a design such as the one depicted in figure 1c, with each core maintaining one SPSC (single-producer/single-consumer) queue for each other core in the system, into which it enqueues items that it wishes its peer to dequeue. As before, this design weakens the consistency guarantee of the queue. It also makes dynamic load balancing more difficult because it chooses which core will dequeue an item in advance.

Motivation for improving synchronization

The crux of this discussion is that obtaining scalability by distributing the queue data structure trades off some useful properties that a centralized shared queue provides. Usually, however, these tradeoffs are clouded by the unacceptable performance of centralized data structures, which obviates any benefit they might offer. This article makes the point that much of this poor performance is a result of inefficient synchronization methods. It surveys advanced synchronization methods that boost the performance of centralized designs and make them acceptable to more applications. With these methods at hand, designers can make more informed choices when architecting their systems.

SCALING LOCKING WITH DELEGATION

Locking inherently serializes executions of the critical sections it protects. Locks therefore limit scaling: the more cores there are, the longer each core has to wait for its turn to execute the critical section, and so beyond some number of cores, these waiting times dominate the computation. This scalability limit, however, can be pushed quite high—in some cases, beyond the scales of current systems—by *serializing more efficiently.* Locks that serialize more efficiently support more operations per second and therefore can handle workloads with more cores.

More precisely, the goal is to minimize the computation's *critical path*: the length of the longest series of operations that have to be performed sequentially because of data dependencies. When using locks, the critical path contains successful lock acquisitions, execution of the critical sections, and lock releases.

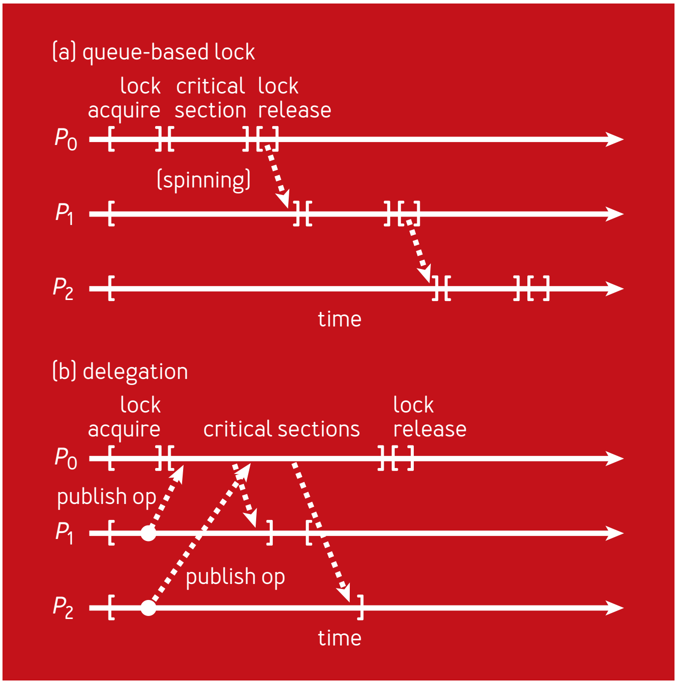
As an example of inefficient serialization that limits scalability, consider the *lock contention* that infamously occurs in simple spin locks. When many cores simultaneously try to acquire a spin lock, they cause its cache line to bounce among them, which slows down lock acquisitions/releases. This increases the length of the critical path and leads to a performance "meltdown" as core counts increase.[2]

Lock contention has a known solution in the form of scalable queue-based locks.[2,10] Instead of having all waiting threads compete to be the next one to acquire the lock, queue-based locks line up the waiting threads, enabling a lock release to hand the lock to the next waiting thread. These hand-offs, which require only a constant number of cache misses per acquisition/release, speed up lock acquisition/release and decrease the length of the critical path, as depicted in figure 2a: the critical path of a queue-based lock contains only a single transfer of the lock's cache line (dotted arrow).

Can lock-based serialization be made even more efficient? The *delegation* synchronization method described in this section does so: it eliminates most lock acquisitions and releases from the critical path, and it speeds up execution of the critical sections themselves.

Delegation. In a delegation lock, the core holding the

FIGURE 2: **CRITICAL PATH OF LOCK-BASED CODE**



lock acts as a server and executes the operations that the cores waiting to acquire the lock wish to perform. Delegation improves scalability in several ways (figure 2b). First, it eliminates the lock acquisitions and releases that would otherwise have been performed by waiting threads. Second, it speeds up the execution of operations (critical sections), because the data structure is hot in the server's cache and does not have to be transferred from a remote cache or from memory. Delegation also enables new optimizations that exploit the semantics of the data structure to speed up critical-section execution even further, as will be described shortly.

Implementing delegation. The idea of serializing faster by having a single thread execute the operations of the waiting threads dates to the 1999 work of Oyama et al.,[13] but the overheads of their implementation overshadow its benefits. Hendler et al.,[6] in their *flat combining* work, were first to implement this idea efficiently and to observe that it facilitates optimizations based on the semantics of the executed operations.

In the flat combining algorithm, every thread about to acquire the lock posts the operation it intends to perform (e.g., `dequeue` or `enqueue(x)`) in a shared *publication list*. The thread that acquires the lock becomes the server; the remaining threads spin, waiting for their operations to be applied. The server scans the publication list, applies pending operations, and releases the lock when done. To amortize the synchronization cost of adding records to the publication list, a thread leaves its publication record in the list and reuses it in future operations. Later work explored piggybacking the publication list on top of a queue lock's queue,[4] and boosting cache locality of the operations by dedicating a core for the server role instead of having threads opportunistically become servers.[9]

Semantics-based optimizations. The server thread has a global view of concurrently pending operations, which it can leverage to optimize their execution in two ways:

- *Combining.* The server can combine multiple operations into one and thereby save repeated accesses to the data structure. For example, multiple counter-increment operations can be converted into one addition.
- *Elimination.* Mutually canceling operations, such as a counter increment and decrement, or an insertion and removal of the same item from a set, can be executed without modifying the data structure at all.

Deferring delegation. For operations that only update the data structure but do not return a value, such as `enqueue()`, delegation facilitates an optimization that can sometimes eliminate serialization altogether. Since these operations do not return a response to the invoking core, the core does not have to wait for the server to execute them; it can just log the requested operation in the publication list and keep running. If the core later invokes an operation whose return value depends on the state of the data structure, such as a `dequeue()`, it must then wait for the server to apply all its prior operations. But until such a time—which in update-heavy workloads can be rare—all of its operations execute asynchronously.

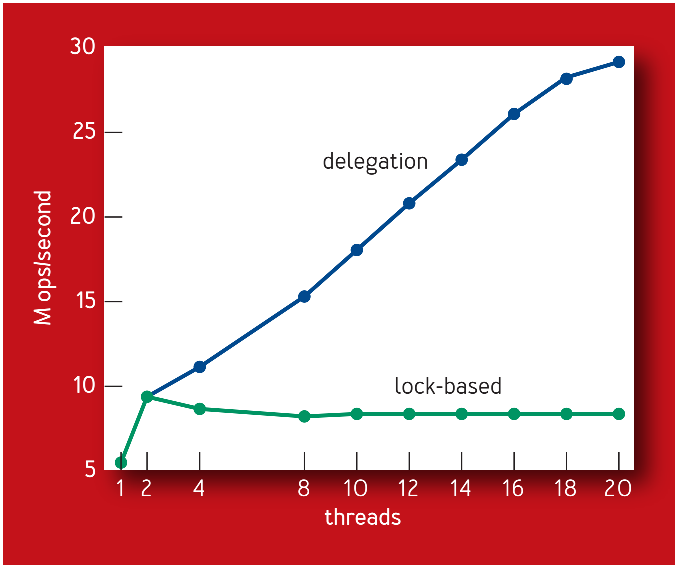
The original implementation of this optimization still required cores to synchronize when executing these deferred operations, since it logged the operations in a centralized (lock-free) queue.[7] Boyd-Wickizer et al.,[3] however, implemented deferred delegation without any synchronization on updates by leveraging systemwide synchronized clocks. Their OpLog library logs invocations of responseless update operations in a per-core log, along with their invocation times. Operations that read the data structure become servers: they acquire the locks of all per-core logs, apply the operations in timestamp order, and then read the updated data-structure state. OpLog thus creates scalable implementations of data structures that are updated heavily but read rarely, such as LRU (least recently used) caches.

Performance

To demonstrate the benefits of delegation, let's compare a lock-based work queue to a queue implemented using delegation. The lock-based algorithm is Michael and Scott's *two-lock queue*.[11] This algorithm protects updates to the queue's head and tail with different locks, serializing operations of the same type but allowing enqueues and dequeues to run in parallel. Queue-based CLH (Craig, Landin, and Hagerstein) locks are used in the evaluated implementation of the lock-based algorithm. The delegation-based queue is Fatourou and Kallimanis' *CC-Queue*,[4] which adds delegation to each of the two locks in the lock-based algorithm. (It thus has two servers running: one for dequeues and one for enqueues.)

Figure 3 shows enqueue/dequeue throughput comparison (higher is better) of the lock-based queue and its delegation-based version. The benchmark models a generic application. Each core repeatedly accesses the data structure, performing pairs of enqueue and

FIGURE 3: **ENQUEUE/DEQUEUE THROUGHPUT COMPARISON**



dequeue operations, reporting the throughput of queue operations (i.e., the total number of queue operations completed per second). To model the work done in a real application, a period of "think time" is inserted after each queue operation. Think times are chosen uniformly at random from 1 to 100 nanoseconds to model challenging workloads in which queues are heavily exercised. The C implementations of the algorithms from Fatourou and Kallimanis's benchmark framework (https://github.com/nkallima/sim-universal-construction) are used, along with a scalable memory allocation library to avoid `malloc` bottlenecks. No semantics-based optimization is implemented.

This benchmark (and all other experiments reported in this article) was run on an Intel Xeon E7-4870 (Westmere EX) processor. The processor has ten 2.40 GHz cores, each of which multiplexes two hardware threads, for a total of twenty hardware threads.

Figure 3 shows the benchmark throughput results, averaged over ten runs. The lock-based algorithm scales to two threads, because it uses two locks, but fails to scale beyond that amount of concurrency because of serialization. In contrast, the delegation-based algorithm scales and ultimately performs almost 30 million operations per second, which is more than 3.5 times that of the lock-based algorithm's throughput.

AVOIDING CAS FAILURES IN LOCK-FREE SYNCHRONIZATION

Lock-free synchronization (also referred to as nonblocking synchronization) directly manipulates shared data using atomic instructions instead of locks. Most lock-free algorithms use the CAS (compare-and-swap) instruction (or equivalent) available on all multicore processors. A CAS takes three operands: a memory address `addr`, an `old` value, and a `new` value. It atomically updates the value stored in `addr` from `old` to `new`; if the value stored in `addr` is not `old`, the CAS fails without updating memory.

CAS-based lock-free algorithms synchronize with a CAS loop pattern: a core reads the shared state, computes a new value, and uses CAS to update a shared variable to the new value. If the CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise, the core must retry. Figure 4 shows an example of such a CAS loop for linking a node to the head of a linked list, taken from

FIGURE 4: **LOCK-FREE LINKING OF A NODE TO THE HEAD OF A LINKED LIST**

```
struct Node {
    struct Node* next;
    void* value;
}
// Pointer to head of the list
Node* head = NULL;

void enqueue (void* v) {
     Node *old, *new = malloc();
     new->value = v;
     while (true) {
          old = head;
          new->next = old;
          if  ( CAS(&head, old, new) )
              return;
}   }
```

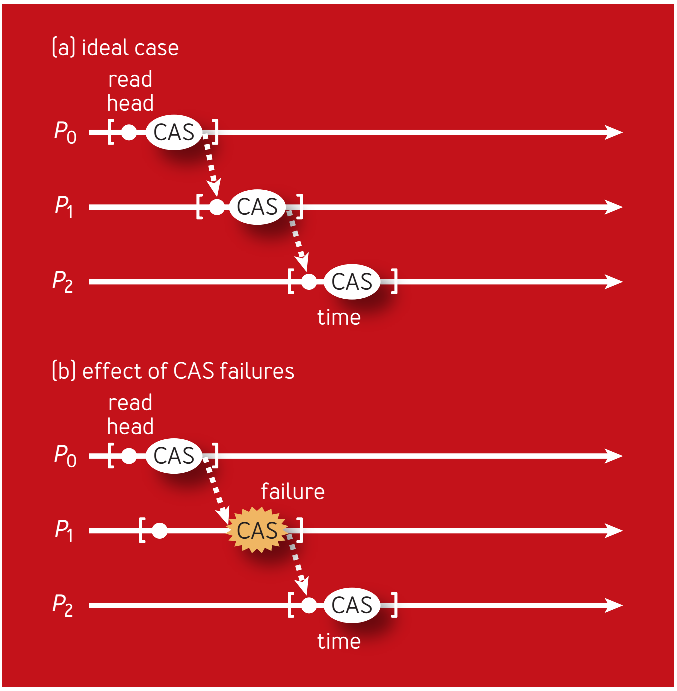
Treiber's classic LIFO (last in, first out) stack algorithm.[15] Similar ideas underlie the lock-free implementations of many basic data structures such as queues, stacks, and priority queues, all of which essentially perform an entire data-structure update with a single atomic instruction.

The use of (sometimes multiple) atomic instructions can make lock-free synchronization slower than a lock-based solution when there is no (or only light) contention. Under high contention, however, lock-free synchronization has the potential to be much more efficient than lock-based synchronization, as it eliminates lock acquire and release operations from the critical path, leaving only the data structure operations on it (figure 5a).

In addition, lock-free algorithms guarantee that some operation can always complete and thus behave gracefully under high load, whereas a lock-based algorithm can grind to a halt if the operating system preempts a thread that holds a lock.

In practice, however, lock-free algorithms may not live up to these performance expectations. Consider, for example, Michael and Scott's lock-free queue algorithm.[11] This algorithm implements a queue using a linked list, with

FIGURE 5: **CRITICAL PATH OF LOCK-FREE UPDATING HEAD OF LINKED LIST**



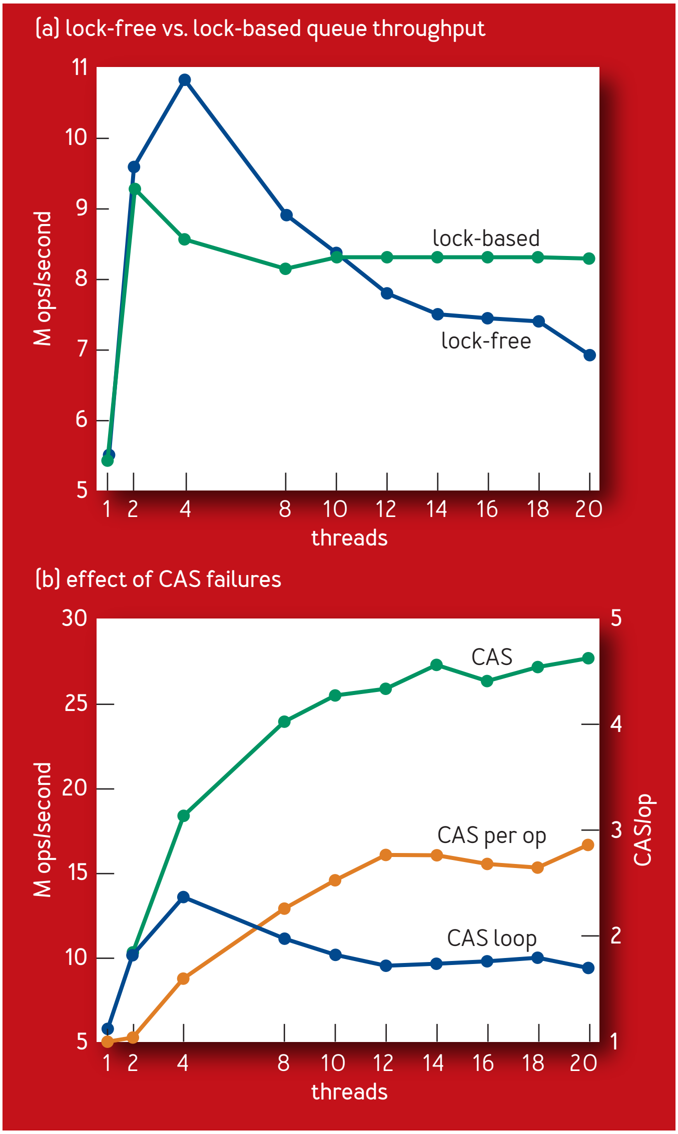
items enqueued to the tail and removed from the head using CAS loops. (The exact details are not as important as the basic idea, which is similar in spirit to the example in figure 4.) Despite this, as figure 6a shows, the lock-free algorithm fails to scale beyond four threads and eventually performs worse than the two-lock queue algorithm.

The reason for this poor performance is *CAS failure*: as the amount of concurrency increases, so does the chance that a conflicting CAS gets interleaved in the middle of a core's read-compute-update CAS region, causing its CAS to fail. CAS operations that fail in this way *pile useless work on the critical path*. Although these failing CASes do not modify memory, executing them still requires obtaining exclusive access to the variable's cache line. This delays the time at which later operations obtain the cache line and complete successfully (see figure 5b, in which only two operations complete in the same time that three operations completed in figure 5a).

To estimate the amount of performance wasted because of CAS failures, figure 6b compares the throughput of successful CASes executed in a CAS loop (as in figure 4) to the total CAS throughput (including failed CASes). Observe that the system executes contending atomic instructions at almost three times the rate ultimately observed in the data structure. If there were a way to make every atomic instruction useful toward completing an operation, you would significantly improve performance. But how can this be achieved, given that CAS failures are inherent?

The key observation to make is that the x86 architecture supports several atomic instructions that always succeed. One such instruction is FAA (fetch-and-add), which

FIGURE 6: **LOCK-FREE SYNCHRONIZATION CAS FAILURE PROBLEM**



atomically adds an integer to a variable and returns the previous value stored in that variable. The following section describes the design of a lock-free queue based on FAA instead of CAS. The algorithm, named LCRQ (for linked concurrent ring queue),[12] uses FAA instructions to

FIGURE 7: **INFINITE ARRAY QUEUE**

```
// The following defines a node
struct Cell {
   void* value;
}
// Queue is infinite array of nodes,
// with head and tail pointers.
Cell Q [] = { ⊥, ⊥, ...};
int head = 0;
int tail = 0;

void enqueue(void* x) {
  while (true) {
   t = FAA(&tail, 1)
   if ( CAS(&Q[t], ⊥, x) ) return
} }
void *dequeue() {
  while (true) {
   h = FAA(&head, 1)
   if ( !CAS(&Q[h], ⊥, ⊤) ) return Q[h]
   if ( tail ≤ h+1 ) return NULL
} }
```

spread threads among items in the queue, allowing them to enqueue and dequeue quickly and in parallel. LCRQ operations typically perform one FAA to obtain their position in the queue, providing exactly the desired behavior.

The LCRQ algorithm

This section presents an overview of the LCRQ algorithm; for a detailed description and evaluation, see the paper.[12] Conceptually, LCRQ can be viewed as a practical realization of the following simple but unrealistic queue algorithm (figure 7). The unrealistic algorithm implements the queue using an *infinite* array, `Q`, with (unbounded) `head` and `tail` indices that identify the part of `Q` that may contain items. Initially, each cell `Q[i]` is empty and contains a reserved value ⊥ that may not be enqueued. The `head` and `tail` indices are manipulated using FAA and are used to spread threads around the cells of the array, where they synchronize using (uncontended) CAS.

An `enqueue(x)` operation obtains a cell index `t` via a FAA on `tail`. The enqueue then atomically places `x` in `Q[t]` using a CAS to update `Q[t]` from ⊥ to `x`. If the CAS succeeds, the enqueue operation completes; otherwise, it repeats this process.

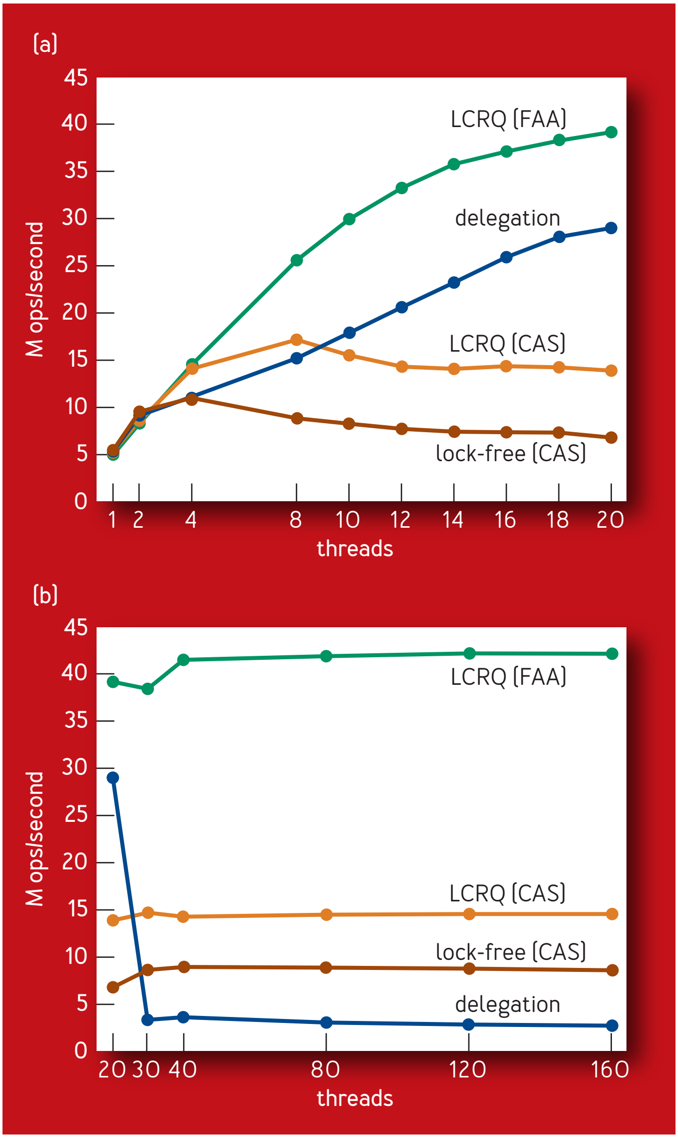
A dequeue, `D`, obtains a cell index `h` using FAA on `head`. It tries to atomically CAS the contents of `Q[h]` from ⊥ to another reserved value ⊤. This CAS fails if `Q[h]` contained some $x \neq \perp$, in which case `D` returns `x`. Otherwise, the fact that `D` stored ⊤ in the cell guarantees that an enqueue operation that later tries to store an item in `Q[h]` will not succeed. `D` then returns NULL (indicating the queue is empty) if `tail ≤ h + 1` (the value of `head` following `D`'s FAA is `h + 1`). If `D` cannot return NULL, it repeats this process.

This algorithm can be shown to implement a FIFO queue correctly, but it has two major flaws that prevent it from being relevant in practice: using an infinite array and susceptibility to livelock (when a dequeuer continuously writes `T` into the cell an enqueuer is about to access). The practical LCRQ algorithm addresses these flaws.

The infinite array is first collapsed to a concurrent ring (cyclic array) queue—CRQ for short—of `R` cells. The `head` and `tail` indices still strictly increase, but now the value of an index modulo `R` specifies the ring cell to which it points. Because now more than one enqueuer and dequeuer can concurrently access a cell, the CRQ uses a more involved CAS-based protocol for synchronizing within each cell. This protocol enables an operation to avoid waiting for the completion of operations whose FAA returns smaller indices that also point to the same ring cell.

The CRQ's crucial performance property is that in the common fast path, an operation executes only one FAA instruction. The LCRQ algorithm then builds on the CRQ to prevent the livelock problem and handle the case of the CRQ filling up. The LCRQ is essentially a Michael and Scott linked list queue[11] in which each node is a CRQ. A CRQ that fills up or experiences livelock becomes closed to further enqueues, which instead append a new CRQ to the list and begin working in it. Most of the activity in the LCRQ therefore occurs in the individual CRQs, making contention (and CAS failures) on the list's head and tail a nonissue.

FIGURE 8: **ENQUEUE/DEQUEUE THROUGHPUT COMPARISON OF ALL QUEUES**



Performance

This section compares the LCRQ to Michael and Scott's classic lock-free queue,[11] as well as to the delegation-based variant presented in the previous section. The impact of CAS failures is explored by testing LCRQ-CAS, a version of LCRQ in which FAA is implemented with a CAS loop.

Figure 8a shows the results. LCRQ outperforms all other queues beyond two threads, achieving peak throughput of ≈ 40 million operations per second, or about 1,000 cycles per queue operation. From eight threads onward, LCRQ outperforms the delegation-based queue by 1.4 to 1.5 times and the MS (Michael and Scott) queue by more than three times. LCRQ-CAS matches LCRQ's performance up to four threads, but at that point its performance levels off. Subsequently, LCRQ-CAS exhibits the throughput "meltdown" associated with CAS failures. Similarly, the MS queue's performance peaks at two threads and degrades as concurrency increases.

Oversubscribed workloads can demonstrate the graceful behavior of lock-free algorithms under high load. In these workloads the number of software threads exceeds the hardware-supported level, forcing the operating system to context-switch between threads. If a thread holding a lock is preempted, a lock-based algorithm cannot make progress until it runs again. Indeed, as figure 8b shows, when the number of threads exceeds 20, the throughput of the lock-based delegation algorithm plummets by 15 times, whereas both LCRQ and the MS queue maintain their peak throughput.

CONCLUSION

Advanced synchronization methods can boost the performance of shared mutable data structures. Synchronization still has its price, and when performance demands are extreme (or if the properties of centralized data structures are not needed), then distributed data structures are probably the right choice. For the many remaining cases, however, the methods described in this article can help build high-performance software. Awareness of these methods can assist those designing software for multicore machines.

References

1. Al Bahra, S. 2013. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM* 56(7): 50–61.
2. Boyd-Wickizer, S., Frans Kaashoek, M., Morris, R., Zeldovich, N. 2012. Non-scalable locks are dangerous. In *Proceedings of the Ottawa Linux Symposium*: 121–132.
3. Boyd-Wickizer, S., Frans Kaashoek, M., Morris, R., Zeldovich, N. 2014. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR2014-019.
4. Fatourou, P., Kallimanis, N. D. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 257– 266.
5. Haas, A., Lippautz, M., Henzinger, T. A., Payer, H., Sokolova, A., Kirsch, C. M., Sezgin, A. 2013. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In

Proceedings of the ACM International Conference on Computing Frontiers: 17:1–17:9.

6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*: 355–364.
7. Klaftenegger, D., Sagonas, K., Winblad, K. 2014. Delegation locking libraries for improved performance of multithreaded programs. In *Proceedings of the 20th International European Conference on Parallel and Distributed Computing*: 572–583.
8. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L. P. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*: 211–222.
9. Lozi, J.-P., David, F., Thomas, G., Lawall, J., Muller, G. 2012. Remote core locking: migrating critical section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*: 65–76.
10. Mellor-Crummey, J. M., Scott, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9(1): 21–65 .
11. Michael, M. M., Scott, M. L. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*: 267–275.
12. Morrison, A., Afek, Y. 2013. Fast concurrent queues

for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 103–112.

13. Oyama, Y., Taura, K., Yonezawa, A. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*: 182–204.
14. Shavit, N. Data structures in the multicore age. 2011. *Communications of the ACM* 54(3): 76–84.
15. Treiber, R. K. 2006. Systems programming: coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center.

Adam Morrison *works on making parallel and distributed systems simpler to use without compromising their performance; his research interests span computer architecture, systems software and theory of distributed computing. He is an assistant professor at the Blavatnik School of Computer Science, Tel Aviv University, Israel. He received a PhD in computer science from Tel Aviv University and then spent three years as a postdoctoral fellow at the Technion—Israel Institute of Technology. He has been awarded an IBM PhD Fellowship as well as Intel and Deutsch prizes.*