Overview - "A Wait-free Queue as Fast as Fetch-and-Add"

Loris Lucido
Author: Chaoran Yang, John Mellor-Crummey

Index Terms—non-blocking queue, wait-free, fast-path-slow-path

1 Introduction

Fast concurrent data structures are necessary to exploit the power of multi-core processors and multi-processor machines. To deliver high scalable performance, applications may need shared objects (data structures) with scalable operations. Each operation of an object (adding, removing, modifying an element) much perform *reasonably* well even when the number of threads sharing an object is high.

One way to construct concurrent data structures is by using an object with blocking operations. An example of said object would be a concurrent queue whose operations have been protected with mutual exclusion (locks). Unexpected delay from one thread prevents the others from using the object. The source of the delay is multiple: operating system preemption, cache miss, page fault, etc. Thus, this approach is not scalable and it may lead to poor performance under high contention, especially if the application uses more software threads than hardware threads. That's why designing non-blocking objects is the topic of lots of past and ongoing research.

Wait-free object We define here levels of *process guarantees* on each operation of a non-blocking object [3]. Those are meant to predict how an object will behave under high contention. The strongest guarantee is wait-free, which means an operation will complete in a finite number of steps regardless of how many threads are accessing this object and at what speed. Lock-free operations complete in a finite number of steps only for *some* threads. Obstruction-free is the weakest guarantee. Only *one* thread is guaranteed to complete an operation in a finite number of steps.

While an object can be wait-free, this doesn't mean it will enable the application to achieve high scalability. Actually, most past wait-free queue implementations perform worse than others lock-free or blocking queue. To achieve a wait-free guarantee, most wait-free implementations have to sacrifice parallelism.

Linearizability Having multiple threads concurrently modifying a shared object must not leave this object (a queue for example) in an *inconsistent* state. To ensure the designed object is correct, we need a *safety condition* for shared objects. The authors choose linearizability for their design of concurrent queue. A sequence of operations is linearizable if, each operation appears to "take effect" instantaneously from the point of view of other threads, regardless of when the operation started and returned. The order of non-concurrent operations must also be preserved [12]. The object is correct if every possible sequences are linearizable.

An atomic operation or an operation protected by mutual exclusion is by definition linearizable. When the operation is not atomic, one must consider the point in time when changes made to the shared object become visible to the other threads.

Memory reclamation Memory management is an important part of designing a concurrent non-blocking algorithm. Garbage collection is not available in all languages. Even if exists, one may need to do one's own memory management to be more efficient, especially if the garbage collection process is not lock-free or wait-free.

The use of `malloc`, for example, may also violate the wait-free property of the queue because `malloc` uses mutual exclusion to manage memory. One solution is to have each thread manages a pool of pre-allocated cells [9]. Enqueue operations take one cell from the pool, dequeue operations put the cell back into the pool. While this solution avoids lock contention, it is prone to the ABA problem which may corrupt memory.

The ABA problem occurs if one thread reads a shared address A, attempts to replace the value of this cell with a compare-and-swap (CAS) and succeeds when it should not. For example, if between the read and the comparison, another thread replaced the cell A to B, and then replaced B to A again, the compare-and-swap may succeed under the assumption that the cell is still the same as before.

One can also use other memory management implementations like `jemalloc`, `TCMalloc`, or `Lockless`.

Consensus number The use of atomic primitives such as compare-and-swap is critical in designing Lock-free algorithm. To understand why, we must consider the consensus problem. A consensus is reached between n threads if those n threads can *decide* on a value: each thread proposes one value to the others and then chooses one value. The consensus is reached if all threads decide on the same value (consistency requirement) and if the common decision is one of the initially proposed value (validity requirement).

Compare-and-swap primitive has an infinite consensus number [11]: it can solve the consensus problem for an infinite number of threads. Compare-and-swap holds an-

other important property which is universality. An object is universal for n threads if and only if it has a consensus number greater or equal to n. It is possible to implement any concurrent object in a wait-free manner if and only if the operating system provides a universal object. Thus, compare-and-swap can be used to implement any arbitrary concurrent wait-free object for any number of threads. This statement holds for other infinite consensus number primitive like load-linked/store-conditional.

Their Contribution Unfortunately, the use of compare-and-swap on a variable shared by several threads can drastically reduce performance even under low contention (compare-and-swap retry problem [4]). In this paper, C. Yang and J. Mellor-Crummey expose a new queue that is wait-free, linearizable, and delivers high performance.

They avoid common issues with past designs such as the compare-and-swap problem by also relying on a fetch-and-add primitive (FAA). Fetch-and-add has consensus number two but is less expensive than compare-and-swap. They designed their wait-free queue using a methodology called fast-path-slow-path [6]. It can transform lock-free queues implemented with compare-and-swap primitive into a wait-free one. They adapted it for queues also using fetch-and-add. They provide their own memory reclamation scheme which adds little overhead, unlike previous work.

2 Context and Related Work

MS-Queue The MS-Queue from M. M. Michael and M. L. Scott is a classical simple lock-free queue implemented with only compare-and-swap as a synchronization primitive [10]. Their queue is designed as a singly-linked list, with two references to the head and the tail of the queue. They use compare-and-swap to add (or remove) an element to the queue and to move the head and tail reference. As a consequence, it is subject to the compare-and-swap retry problem. Even with a low number of threads, performance is drastically reduced under contention because of high probability of compare-and-swap failures.

The MS-Queue avoids the ABA problem by using a compare-and-swap with modification counters. Each compare-and-swap on an address also increments a counter associated with that address so that two identical addresses can be distinguished by how much they have been modified [9] [10].

Practical wait-free queue One of the first implementation of a wait-free queue supporting multiples enqueuers and dequeuers was designed by A. Kogan and E. Petrank [8]. This queue is based on the MS-Queue. Each operation is divided into three atomic steps. Threads can help one another to complete a step without letting the possibility for steps interleaving among the same type of operation (enqueue or dequeue). Because of the overhead of this helping mechanism, it doesn't perform as well as the MS-Queue.

Combining-based queues The P-Sim queue (wait-free) and CC-Queue (blocking) from P. Fatourou and N. D. Kallimanis uses *operation combining* to try to achieve better scalability than compare-and-swap-based queue by reducing synchronization cost [7] [5].

To do so, threads don't directly modify the queue but publish a request. A single thread browses a list of pending operations and applies them serially on the queue. While the CC-Queue performs better than the P-Sim queue, it is still a blocking object and, also because of the lack of parallelism, the CC-queue doesn't scale well.

FAA-based LCRQ The LCRQ is a lock-free queue designed by A. Morrison and Y. Afek [4]. It is implemented as a circular ring queue. Threads use fetch-and-add to get a cell index on the queue, enqueue and dequeue are then done with double width compare-and-swap, which is not universally available. However, by also relying on fetch-and-add, this queue avoids the compare-and-swap retry problem.

Fast-path-slow-path Designed by A. Kogan and E. Petrank, the objective of the fast-path-slow-path methodology is to construct a wait-free algorithm from a lock-free one [6]. The resulting algorithm relies on a compare-and-swap-based lock-free algorithm *most* of the time (fast-path), and switch to a wait-free algorithm if too many compared-and-swap failures occurred by using a helping mechanism (slow-path). It aims to achieve lock-free performance with a wait-free guarantee. They experimented it with the lock-free MS-queue as the fast path.

3 Contribution of the Article

The authors started from a basic obstruction-free queue similar to the base used in the LCRQ algorithm. It is given in Listing 1. With this queue, they constructed a wait-free queue by using the fast-path-slow-path methodology.

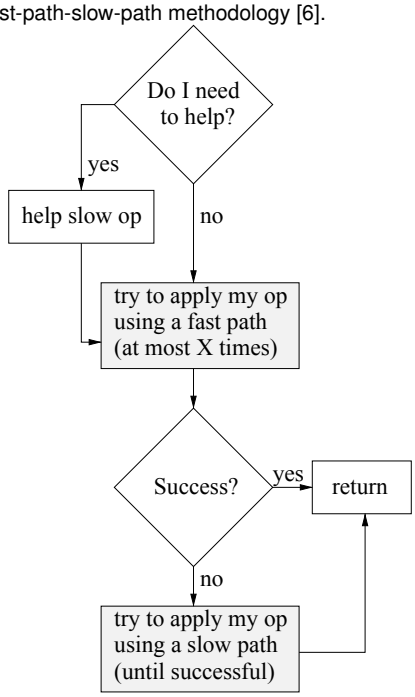
$FAA(x, v)$ atomically reads the value stored in the x variable, increments it by v and returns the x value pre-incrementation. $CAS(x, t, v)$ atomically reads the value stored in x, compares it to t and, if x is equal to t (success), replaces the value of x by v. CAS returns whether it has successfully replaced the value or not.

Listing 1. An obstruction-free queue using an infinite array.

```
Q: queue
T: pointer to tail
H: pointer to head
enqueue(x: var) {
  do t := FAA(&T, 1);
  while (!CAS(&Q[t], ⊥, x));
}
dequeue(x: var) {
  do h := FAA(&H, 1);
  while (CAS(&Q[h], ⊥, ⊤) and T > h);
  return (Q[h] == ⊤ ? EMPTY : Q[h]);
}
```

Basic queue This queue uses a shared infinite array (emulated) to store elements. We will see later how this is done and how memory can be reclaimed. Two special values are reserved : $\bot$ (bottom) and $\top$ (top). $\bot$ stands for empty cells as the queue is initially filled with $\bot$. $\top$ stands for unusable cells. When one thread dequeues an element, it marks the cells with $\top$ to prevent other threads to enqueue an element in it.

Fig. 1. The fast-path-slow-path methodology [6].



To enqueue an element, one thread tries to find an available cell on the array, a cell marked with $\bot$. One cannot enqueue an element in a $\top$ marked cell because it could violate the FIFO property of the queue. To get a unique index on the array, one thread uses fetch-and-add to increment the shared tail reference. Considering the reference is shared by all threads, using compare-and-swap instead of fetch-and-add would result in lots of failures. After one thread is given a unique index, it needs to certify that the cell is empty. It thus uses compared-and-swap to enqueue an element into the array. In the event of compare-and-swap failure, the thread redoes the whole process until the element is enqueued.

The dequeue operation works in a similar manner, one thread tries to find either a cell filled with an element or marked with $\top$ by using fetch-and-add on a shared head reference. If a thread stops on a $\top$ marked cell, the dequeue returns `EMPTY`.

This queue is designed to be fast. Enqueue and dequeue may both fail if a dequeuer thread marks the candidate cell for the enqueue unusable. As a result, it is neither wait-free or lock-free because it is prone to livelocking. As only one thread is guaranteed to progress, this queue is only obstruction-free.

Fast-path-slow-path To construct a wait-free realization of the previous queue, the authors use the fast-path-slow-path methodology. As shown in figure 1, one thread tries to enqueue or dequeue an element using a fast-path, an algorithm similar to the previous queue. A maximum number of failures is set. If a thread failed too many times to apply its operation, it falls back on a slow-path.

For example, if a thread fails to enqueue an element, it publishes an enqueue request. When other threads want to dequeue an element, they look at all pending enqueue requests, and eventually help one request to complete. The enqueuer thread then keeps trying to enqueue the element until it or another succeeds.

Threads are linked in a ring, they keep a reference to a *peer* to which they help the operation to complete. Each time a thread successfully helps another thread, it updates his *peer* to the next thread in the ring so that each thread happens to help every one at some point.

Memory reclamation The authors designed their queue with an infinite array. To do so, the queue is split into segments as a linked list. Each segment contains a fixed number of cells. The list of segments is expanded as necessary. Because the tail and head pointers are never decremented, segments no longer in use need to be freed. A segment can be freed when the head and tail pointers of the queue move past the range of cells handled by this segment.

After one thread dequeues an element, the thread tries to reclaim memory from unused segments. They use compare-and-swap to achieve mutual exclusion so that if several threads try to reclaim memory, only one should succeed. Each thread keeps a reference to its currently used segment. The *cleaner* thread ensures that no thread keeps a reference to a segment about to be freed and changes it if needed.

Wait-free guaranty The number of tries on the obstruction-free fast-past is bounded. After sufficient failures, the algorithm falls back on the slow-path. Each thread helps every other thread at some point. If the operation of one thread continuously fails to complete, it will definitely complete when all other threads become his helper.

4 Discussion

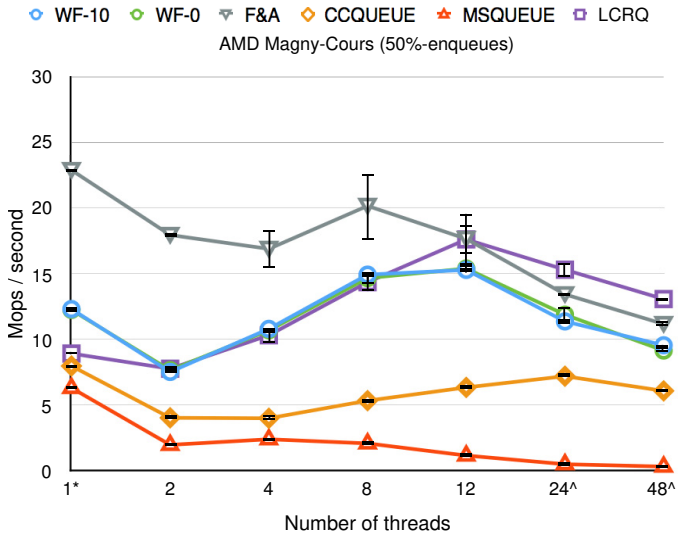
The authors experimented on several architectures. Some of their experiments are shown figure 2. They evaluated the MS-Queue (lock-free), CC-Queue (blocking), LCRQ (lock-free) and their wait-free design named WF-0 (one try on the fast-path) and WF-10 (10 tries on the fast-path). They also evaluated a microbenchmark, simulating enqueue and dequeue operations using only one fetch-and-add. It should serve as an upper bound for any fetch-and-add-based queue.

Their wait-free queue outperforms most tested designs when run with only one thread. With several threads, their design performance is close to the lock-free LCRQ design. Their memory reclamation scheme adds no overhead on x86 architecture, which is unprecedented.

In their experiments, the authors haven't compared their queue to another wait-free queue constructed with the fast-path-slow-path methodology like the one from A. Kogan and E. Petrank [6], which uses MS-Queue as fast-path. The reason given is that this design can only be as good as the MS-Queue. Because the number of failures of compare-and-swap isn't bounded for MS-Queue, it is not clear that this wait-free queue can't be faster, under heavy contention, than the original MS-Queue.

Recent research has suggested that some lock-free objects [1] and even blocking objects [2] can *practically* be considered wait-free given how unlikely conflicts occur. For example, for the queue presented in this overview, experimental results show that 99.97% of the enqueues and at most 95.95% of the dequeues are done with only one try on the fast-path

Fig. 2. Throughput of different queues. A thread decides uniformly at random to execute an enqueue or dequeue with equal odds. The benchmark executes 10^7 operations partitioned evenly among all threads. ^: executions that involve multiple processors. [3]



(which is obstruction-free). I believe the research emphasises the idea that applications needing a state-of-the-art wait-free queue instead of a simpler, faster obstruction-free or lock-free queue are quite rare.

The authors remind us that fetch-and-add are not universally supported. For example, the Power, SPARC and ARM processors don't provide a native fetch-and-add primitive. While it can be emulated with load-linked/store-conditional primitive, the queue loses its wait-free property.

Fetch-and-add may fail in a similar way as compare-and-swap fails but at hardware level which blurs the difference between wait-free fetch-and-add-based queue and lock-free queue. When several fetch-and-add conflict on the same memory location, I postulate that they must be treated in a starvation-free manner by the hardware to keep the wait-free property of the queue.

When multiple processors are involved and on architecture where the fetch-and-add is natively available, the fetch-and-add microbenchmark doesn't always perform better than the LCRQ which uses fetch-and-add and compare-and-swap, as show in figure 2 This makes me reconsider the effectiveness and wait-free property of fetch-and-add.

5 Conclusion

C. Yang and J. Mellor-Crummey designed a wait-free queue from a simple obstruction-free queue using the fast-path-slow-path methodology. They provide the first wait-free queue with performance comparable to prior lock-free and blocking designs, which make it usable on most multi-core architectures and for most applications. Their also implemented a memory reclamation mechanism without any overhead on x86 architecture, unlike previous works.

However, they significantly increased the design complexity. In the case where one application can neglect the low probability of the worse-case scenario (slow-path), using a lock-free or obstruction-free algorithm may be an acceptable solution because of a simpler implementation.

References

[1] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. "Are Lock-Free Concurrent Algorithms Practically Wait-Free?" In: *J. ACM* 63.4 (Sept. 2016), 31:1–31:20. ISSN: 0004-5411. DOI: 10.1145/2903136.

[2] Tudor David and Rachid Guerraoui. "Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. New York, NY, USA: ACM, 2016, pp. 337–348. ISBN: 978-1-4503-4210-0. DOI: 10.1145/2935764.2935774.

[3] Chaoran Yang and John Mellor-Crummey. "A Wait-free Queue As Fast As Fetch-and-add". In: *SIGPLAN Not.* 51.8 (Feb. 2016), 16:1–16:13. ISSN: 0362-1340. DOI: 10.1145/3016078.2851168.

[4] Adam Morrison and Yehuda Afek. "Fast Concurrent Queues for x86 Processors". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 103–112. ISSN: 0362-1340. DOI: 10.1145/2517327.2442527.

[5] Panagiota Fatourou and Nikolaos D. Kallimanis. "Revisiting the Combining Synchronization Technique". In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 257–266. ISSN: 0362-1340. DOI: 10.1145/2370036.2145849.

[6] Alex Kogan and Erez Petrank. "A Methodology for Creating Fast Wait-free Data Structures". In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 141–150. ISSN: 0362-1340. DOI: 10.1145/2370036.2145835.

[7] Panagiota Fatourou and Nikolaos D. Kallimanis. "A Highly-efficient Wait-free Universal Construction". In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. New York, NY, USA: ACM, 2011, pp. 325–334. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989549.

[8] Alex Kogan and Erez Petrank. "Wait-free Queues with Multiple Enqueuers and Dequeuers". In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 223–234. ISSN: 0362-1340. DOI: 10.1145/2038037.1941585.

[9] Maurice P. Herlihy and Nir Shavit. *The art of multiprocessor programming*. Elsevier/Morgan Kaufmann, 2008, pp. 244–262.

[10] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proc. 15th ACM Symp. on Principles of Distributed Computing*. 1996, pp. 267–275.

[11] Maurice Herlihy. "Wait-free Synchronization". In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: 10.1145/114005.102808.

[12] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.