Brief Announcement: Jiffy: A Fast, Memory Efficient, Wait-Free Multi-Producers Single-Consumer Queue

Dolev Adas
Technion – Israel Institute of Technology, Haifa, Israel
sdolevfe@cs.technion.ac.il

Roy Friedman
Technion – Israel Institute of Technology, Haifa, Israel
roy@cs.technion.ac.il

Abstract

In applications such as sharded data processing systems, data flow programming and load sharing applications, multiple concurrent data producers are feeding requests into the same data consumer. This can be naturally realized through concurrent queues, where each consumer pulls its tasks from its dedicated queue. For scalability, wait-free queues are often preferred over lock based structures.

The vast majority of wait-free queue implementations, and even lock-free ones, support the multi-producer multi-consumer model. Yet, this comes at a premium, since implementing wait-free multi-producer multi-consumer queues requires utilizing complex helper data structures. The latter increases the memory consumption of such queues and limits their performance and scalability. Additionally, many such designs employ (hardware) cache unfriendly memory access patterns.

In this work we study the implementation of wait-free multi-producer single-consumer queues. Specifically, we propose Jiffy, an efficient memory frugal novel wait-free multi-producer single-consumer queue and formally prove its correctness. We then compare the performance and memory requirements of Jiffy with other state of the art lock-free and wait-free queues. We show that indeed Jiffy can maintain good performance with up to 128 threads, delivers better throughput than other constructions we compared against, and consumes less memory.

2012 ACM Subject Classification Software and its engineering → Concurrency control; Theory of computation → Concurrency

Keywords and phrases Wait-freedom, MPSC Queues, Concurrent data-structures

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.50

Funding Israel Science Foundation grant #1505/16.

1 Background

Concurrent queues are a fundamental data-exchange mechanism in multi-threaded applications. A queue enables one thread to pass a data item to another thread in a decoupled manner, while preserving ordering between operations. The thread inserting a data item is often referred to as the *producer* or *enqueuer* of the data, while the thread that fetches and removes the data item from the queue is often referred to as the *consumer* or *dequeuer* of the data. In particular, queues can be used to pass data from multiple threads to a single thread - known as *multi-producer single-consumer queue* (MPSC), from a single thread to multiple threads - known as *single-producer multi-consumer queue* (SPMC), or from multiple threads to multiple threads - known as *multi-producer multi-consumer queue* (MPMC).

MPSC is useful in sharded software architectures, resource allocation and data-flow computation schemes. In many sharded architectures, a single thread is responsible for each shard, in order to avoid costly synchronization while accessing a specific shard. In this case, multiple feeder threads (e.g., that communicate with external clients) insert requests into the

queues according to the shards. Each thread that is responsible for a given shard repeatedly dequeues the next request for the shard, executes it, dequeues the next request, etc. Similarly, in a data flow graph, multiple events may feed the same computational entity, e.g., a reducer that reduces the outcome of multiple mappers. Here too each computational entity can be served by a single thread while multiple threads are sending it items (requests) to be handled.

MPMC is the most general form of a queue and can be used in any scenario. Therefore, MPMC is also the most widely studied data structure [5, 6, 7, 9, 10, 11, 12]. Yet, this may come at a premium compared to using a more specific queue implementation.

Specifically, concurrent accesses to the same data structure require adequate concurrency control to ensure correctness. The simplest option is to lock the entire data structure on each access, but this usually dramatically reduces performance due to the sequentiality and contention it imposes [4]. A more promising approach is to reduce, or even avoid, the use of locks and replace them with *lock-free* and *wait-free* protocols that only rely on atomic operations such as *fetch-and-add* (FAA) and *compare-and-swap* (CAS), which are supported in most modern hardware architectures [3]. Wait-free implementations are particularly appealing since they ensure that each operation always terminates in a finite number of steps.

Alas, known MPMC wait-free queues suffer from large memory overheads, intricate code complexity, and low scalability. In particular, it was shown that wait-free MPMC queues require the use of a helper mechanism [1]. On the other hand, as discussed above, there are important classes of applications for which MPSC queues are adequate. Such applications could therefore benefit if a more efficient MPSC queue construction was found. This motivates studying wait-free MPSC queues, which is the topic of this paper.

2 Contributions

In this work we present Jiffy, a fast memory efficient wait-free MPSC queue. Jiffy is unbounded in the the number of elements that can be enqueued without being dequeued (up to the memory limitations of the machine). Yet the amount of memory Jiffy consumes at any given time is proportional to the number of such items and Jiffy minimizes the use of pointers, to reduce its memory footprint.

To obtain these good properties, Jiffy stores elements in a linked list of arrays, and only allocates a new array when the last array is being filled. Also, as soon as all elements in a given array are dequeued, the array is released. This way, a typical enqueue operation requires little more than a simple FAA and setting the corresponding entry to the enqueued value and changing its status from `empty` to `set`. Hence, operations are very fast and the number of pointers is a multiple of the allocated arrays rather than the number of queued elements.

To satisfy linearizability and wait-freedom, a dequeue operation in Jiffy may return a value that is already past the head of the queue, if the enqueue operation working on the head is still on-going. To ensure correctness, we devised a novel mechanism to handle such entries both during their immediate dequeue as well as during subsequent dequeues.

Another novel idea in Jiffy is related to its buffer allocation policy. Naively, when the last buffer is full, any enqueuer at that point should allocate a new buffer and try adding it to the queue with a CAS. When multiple enqueuers try this concurrently, only one succeeds and the others need to free their allocated buffer. However, this both creates contention on the end of the queue and wastes CPU time in allocating and freeing multiple buffers each time. To alleviate these, in Jiffy the enqueuer of the second entry in the last buffer already allocates the next buffer and tries to add it using CAS. This way, almost always, when enqueuers reach the end of a buffer, the next buffer is already available for them with no contention.

We have implemented Jiffy and evaluated its performance in comparison with three other leading lock-free and wait-free implementations, namely WFqueue [13], CCqueue [2], and MSqueue [7]. We also examined the memory requirements for the data and code of all measured implementations using valgrind [8]. The results indicate that Jiffy is up to 50% faster than WFqueue and roughly 10 times faster than CCqueue and MSqueue in a multiple enqueuers single dequeuer workload. Jiffy is also more scalable than the other queue structures we tested, enabling more than 20 million operations per second even with 128 threads. Finally, the memory footprint of Jiffy is roughly 90% better than its competitors in the tested workloads, and provides similar benefits in terms of number of cache and heap accesses. Jiffy obtains better performance since the size of each queue node is much smaller and there are no auxiliary data structures. For example, in WFqueue, which also employs a linked list of arrays approach, each node maintains two pointers, there is some per-thread meta-data, the basic slow-path structure (even when empty), etc. Further, WFqueue employs a lazy reclamation policy, which according to its authors is significant for its performance. Hence, arrays are kept around for some time even after they are no longer useful. In contrast, the per-node meta-data in Jiffy is just a 2-bit flag, and arrays are being freed as soon as they become empty. This translates to a more (hardware) cache friendly access pattern. Also, in Jiffy dequeue operations do not invoke any atomic (e.g., FAA & CAS) operations at all.

References

1. Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015.
2. Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the Combining Synchronization Technique. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47 (8), pages 257–266, 2012.
3. Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
4. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.
5. Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-Free FIFO Queues. In *Distributed Computing, 18th International Conference (DISC)*, pages 117–131, 2004.
6. Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and Efficient Bounded FIFO Queues. In *25th International Symposium on Computer Architecture and High Performance Computing*, pages 144–151. IEEE, 2013.
7. Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.
8. Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.
9. William N. Scherer, Doug Lea, and Michael L. Scott. Scalable Synchronous Queues. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 147–156, 2006.
10. Michael L. Scott. Non-Blocking Timeout in Scalable Queue-Based Spin Locks. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 31–40, 2002.
11. Michael L. Scott and William N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, 2001.
12. Nir Shavit and Asaph Zemach. Scalable Concurrent Priority Queue Algorithms. In *Proc. of the 18th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 113–122, 1999.
13. Chaoran Yang and John Mellor-Crummey. A Wait-Free Queue as Fast as Fetch-and-Add. *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2016.