Efficient Lock-Free Queues that Mind the Cache

Anders Gidenstam
School of Business and Informatics
University of Borås
Borås, Sweden
anders.gidenstam@hb.se

Håkan Sundell
School of Business and Informatics
University of Borås
Borås, Sweden
hakan.sundell@hb.se

Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
philippas.tsigas@chalmers.se

ABSTRACT

This paper discusses a lock-free FIFO queue data structure that is presented in [3]. The algorithm supports multiple producers and multiple consumers and weak memory models. It has been designed to be cache-aware and work directly on weak memory models. It utilizes the cache behavior in concert with lazy updates of shared data, and a dynamic lock-free memory management scheme to decrease unnecessary synchronization and increase performance. Experiments on an 8-way multi-core platform show significantly better performance for the algorithm compared to previous fast lock-free queue algorithms.

1. INTRODUCTION

Lock-free implementation of data structures is a scalable approach for designing concurrent data structures. Lock-free data structures offer high concurrency but also immunity to deadlocks and convoying, in contrast to their blocking counterparts. Concurrent FIFO queue data structures are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. This paper discusses an efficient lock-free queue data structure for multiple producers and consumers presented in [3]. The algorithm is cache-aware in order to minimize its communication overhead. It works also on weak memory consistency models (e.g. due to out-of-order execution) without need for additional fence [5] instructions for reads and writes to shared memory done in the algorithm.

With the strongly emerging multi-core architectures for main-stream as well as high-performance computing, there is an increasing interest for efficient concurrent data structures that allow maximal exploitation of the available parallelism. With the ever more complex multithreaded architectures of applications and systems, there is also likely to be an increasing need for stronger progress and safety guarantees of components in supporting frameworks, and consequently non-blocking synchronization would fit very well thanks to both its possible advantages in performance and its progress properties.

Two basic non-blocking methods have been proposed in the literature, *lock-free* and *wait-free* [4]. *Lock-free* implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to complete it. However, real-time systems might have stronger requirements on progress, and thus in *wait-free* implementations each task is guaranteed to *correctly* complete any operation in a *bounded* number of its own steps, regardless of overlaps of the individual steps and the execution speed of other processes; i.e., while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom ensures individual progress for every task in the system.

Large efforts have been made on designing efficient concurrent queue data structures and blocking (or mixed with non-blocking techniques) implementations are available in most contemporary programming language frameworks supporting multithreading. In this work, we focus only on strictly non-blocking queue algorithms as implementations being just "concurrent" (and possibly efficient as e.g. "lock-less") are still prune to problems as e.g. deadlocks. Absence of explicit locks does not imply any non-blocking properties, unless the latter are proven to be fulfilled. A large number of lock-free (and wait-free) queue implementations have appeared in the literature, e.g. [7][1][12][9][10][6] being the most influential or recent and most efficient results. These results all have a number of specialties or drawbacks as e.g. limitations in allowed concurrency, static in size, requiring atomic primitives not available on contemporary architectures, and scalable in performance but having a high overhead. The algorithm improves on previous results by combining the underlying approaches and designing the algorithm cache-aware and tolerant to weak memory consistency models in order to maximize efficiency on contemporary multi-core platforms. The lock-free algorithm has no limitations on concurrency, is fully dynamic in size, and only requires atomic primitives available on contemporary platforms. Experiments on an 8-way multi-core platform show significantly better performance for the algorithm compared to previous lock-free implementations.

The rest of the paper is organized as follows. In Section 2, related work is discussed. Section 3 describes the algorithm. In Section 4, some benchmark experiments are described. Finally, Section 5 concludes this paper. The full details and analysis of the algorithm are available in [3].

2. RELATED WORK

Lamport [7] presented a lock-free (actually wait-free) implementation of a queue based on a static array, with limited concurrency supporting only one producer and one consumer. In this algorithm,

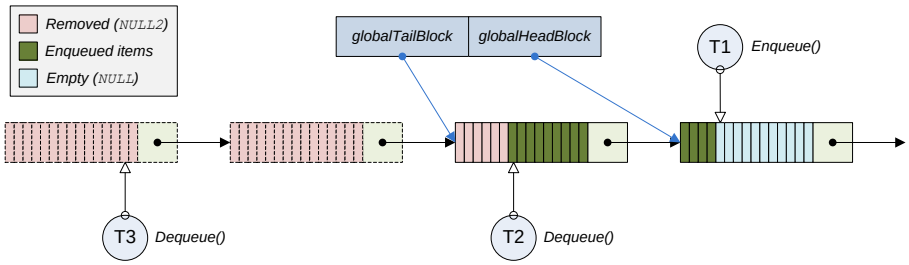

Figure 1: A lock-free queue implemented using a linked list of arrays, where each thread is avoiding accesses to global pointers in order to reduce number of cache misses.

synchronization is done via shared indices indicating the current head and tail array element. Giacomoni et al. [1] presented a cache-aware modification which instead synchronize directly on the array elements. Tsigas and Zhang [12] presented a lock-free extension of [7] where synchronization is done both directly on the array elements and the shared head and tail indices using *CAS*[1], thus supporting multiple producers and consumers. In order to avoid the ABA problem when updating the array elements, the algorithm exploits using two (or more) null values; the ABA problem is due to the inability of *CAS* to detect concurrent changes of a memory word from a value (A) to something else (B) and then again back to the first value (A). Moreover, for lowering the memory contention the algorithm alternates every other operation between scanning and updating the shared head and tail indices.

In resemblance to [7][1][12] the algorithm described here uses arrays to store (pointers to) the items, and in resemblance to [12] it uses *CAS* and two null values. Moreover, shared indices [1] are avoided and scanning [12] is preferred as much as possible. In contrast to [7][1][12] the array is not static or cyclic, but instead more arrays are dynamically allocated as needed when new items are added, making our queue fully dynamic.

Michael and Scott [9] presented a lock-free queue based on a linked list, supporting multiple producers and consumers. Synchronization is done via shared pointers indicating the current head and tail node as well via the next pointer of the last node, all updated using *CAS*. The queue is fully dynamic as more nodes are allocated as needed when new items are added. The original presentation used unbounded version counters, and therefore required double-width *CAS* which is not supported on all contemporary platforms. The problem with the version counters can easily be avoided by using some memory management scheme as e.g. [8]. Moir et al. [10] presented an extension where elimination is used as a back-off strategy and increasing scalability when contention on the queue's head or tail is noticed via failed *CAS* attempts. However, elimination is only possible when the queue is close to be empty during the operation's invocation. Hoffman et al. [6] takes another approach to increase scalability by allowing concurrent *Enqueue* operations to insert the new node at adjacent positions in the linked list if contention is noticed during the attempted insert at the very end of the linked list. To enable these "baskets" of concurrently inserted nodes, removed nodes are logically deleted before the actual removal from the linked list, and as the algorithm traverses through the linked list it requires stronger memory management than [8] and a strategy to avoid the risk of long chains of logically deleted nodes developing.

In resemblance to [9][10][6] the algorithm discussed here is dynamic, and in resemblance to [6] removed blocks are logically deleted, blocks are being traversed and creation of long chains are avoided. In contrast to [10][6] the algorithm employs no special strategy for increasing scalability besides allowing disjoint *Enqueue* and *Dequeue* operations to execute in parallel.

3. THE ALGORITHM

The underlying data structure that our algorithmic design uses is a linked list of arrays, and is depicted in Fig. 1. In the data structure every array element contains a pointer to some arbitrary value. Both the *Enqueue* and *Dequeue* operations use increasing array indices as each array element gets occupied or removed. To ensure consistency, items are inserted or removed from array elements using the *CAS* atomic synchronization primitive. To ensure that a *Enqueue* operation will not succeed with a *CAS* at an array index where a concurrent *Dequeue* operation have already removed an item, we need to enable the *CAS* primitive to distinguish (i.e., avoid the ABA problem) between "used" and "unused" array indices. For this purpose two null pointer values [12] are used; one (`NULL`) for the empty indices and another (`NULL2`) for the removed indices. As each array block gets fully occupied (or emptied), new array blocks are added to (or removed from) the linked list data structure. Two shared pointers, globalHeadBlock and globalTailBlock, are globally indicating the first and last active blocks respectively. These shared pointers are concurrently updated using *CAS* operations as the linked list data structure changes. However, as these updates are done lazily (not atomically together with the addition of a new array block), the actually first or last active block might be found by following the next pointers of the linked list.

As a successful update of a shared pointer will cause a cache miss to other threads that concurrently access that pointer, the overall strategy for improving performance and scalability is to avoid accessing pointers that can be concurrently updated [6]. Moreover, our algorithm achieves fewer updates by not having shared variables with explicit information regarding which array index is currently the next active for the *Enqueue* or *Dequeue*. Instead each thread is storing its own[2] pointers indicating the last known (by this thread) first and active block as well as active indices for inserting and re-

[1]The Compare-And-Swap (CAS) atomic primitive will update a given memory word, if and only if the word still matches a given value (e.g. the one previously read). CAS is generally available in contemporary systems with shared memory, supported directly by hardware or in other cases in combination with system software.

[2]Each thread have their own set of variables stored in separate memory using thread-local storage (TLS).

moving items. When a thread recognizes its own pointers to be inaccurate and stale, it performs a scan of the array elements and array blocks towards the right, and only resorts to reading the global pointers when we expect it to be beneficial compared to scanning. The *Dequeue* operation to be performed by thread T3 in Fig. 1 illustrates a thread that has a stale view of the status of the data structure and thus needs to scan. As array elements are adjacent in memory, scanning can normally be done without extra cache misses (besides those caused by concurrent successful *Enqueue* and *Dequeue* operations) and also without any constraints on in which order memory updates are propagated through the shared memory, thus allowing weak memory consistency models without the need for additional memory fence instructions.[3]

For our implementation of the lock-free queue algorithm we have selected the lock-free memory management scheme proposed by Gidenstam et al. [2] which uses the *CAS* and *FAA* atomic synchronization primitives. Using this scheme we can assure that an array block is not reclaimed when there is any next pointer in the linked list pointing to it or any local references to it from pending concurrent operations or from pointers in thread-local storage. When supplied with appropriate callback functions the scheme automatically reduces the length of possible chains of deleted nodes (held from reclamation by late threads holding a reference to an old array block), and thus enables an upper bound on the maximum memory usage for the data structure. The task of the callback functions *CleanUpNode* is to break chains of deleted array blocks by updating the next pointer of a deleted array block such that it points to an active array block, in a way that is consistent with the semantics of the *Enqueue* and *Dequeue* operations.

The linked list data structure always contains at least one array block. Each array block contains the additional fields head and tail that are used only to indicate either fullness or emptiness of the whole array block.

The *Enqueue* operation operates as follows: after scanning for the first empty (i.e., an array element containing `NULL`) array index in its current array block, it tries to insert the new item by updating the array element with *CAS*. If this fails (due to a concurrent successful *Enqueue*), it continues scanning until the end of the array. If the end of the array is reached, it first assures lock-freedom and accuracy of the global head pointer:
(i) If the global head pointer is not pointing to the current block, the operation (after it verifies that the global head pointer is pointing to the previous block) updates the head pointer to do so by using a *CAS* operation.
(ii) If the global head pointer is pointing to the current array block, the algorithm tries to insert a new array block by updating the next pointer using a *CAS*. If this fails, this is due to some concurrent *Enqueue* operation having already added a new block, so the operation continues scanning for an empty array index in that block.

The *Dequeue* operation operates as follows: after scanning for the first non-empty (i.e., an array element with neither `NULL` or `NULL2`) array index in its current array block, it tries to remove the found item by updating the array element with a *CAS*. If this fails (due to a concurrent successful *Dequeue*), it continues scanning until the end of the array block. If `NULL` is found during scanning, the queue is (after also ensuring the `NULL` value to be globally consistent using *CAS*[4]) recognized to be empty and the operation returns an empty value. If the end of the array is reached, the algorithm first assures lock-freedom and accuracy of the global tail pointer:
(i) If the global tail pointer is pointing to the current array block, it tries to logically mark the block as deleted using a *CAS*.
(ii) If the global tail pointer was not pointing to the current block, it is (after verified that it is pointing to the previous block) updated to do so using a *CAS*. Whenever the global tail pointer is successfully updated (either when helping or after a successful logical deletion), the previously global tail-block is sent for memory reclamation.

Whenever an array element is successfully updated with `NULL2` using *CAS*, the found item is returned by the *Dequeue* operation.

4. EXPERIMENTS

We have evaluated the performance of our lock-free queue algorithm by the means of some custom micro-benchmarks. The purpose of these experiments is to help estimate how well the algorithm compares with other known lock-free queues under high contention and increasing concurrency. The benchmarks are the following:
(i) Random 50%/50%. Each thread is randomly (the sequence is decided beforehand) executing either an *Enqueue* or a *Dequeue* operation.
(ii) Random 50%/50% Bias 1000. Performed as the previous benchmark, but with the queue initialized with 1000 items.
(iii) 1 Producer / N-1 Consumers. Each thread (out of N) is either a producer or consumer, throughout the whole experiment. The producer is repeatedly executing *Enqueue* operations, whereas the consumers are executing *Dequeue*.
(iv) N-1 Producers / 1 Consumer. Same as the previous benchmark, with the producer and consumer distributions interchanged.

For comparison we have implemented the dynamic lock-free queues by Michael and Scott [9], ditto with elimination [10], the baskets queue [6], and the static cyclic array lock-free queue presented in [12]. All dynamic queues have been implemented to support queue sizes only limited by the system's memory, i.e., using lock-free management schemes [8] or [2] and lock-free free-lists where appropriate. For the new implementation, the size of the array block is chosen to fit within one cache line. All implementations are written in C and compiled with the highest optimization level. In our experiments, each concurrent thread is started at the very same time and each benchmark runs for one second for each implementation. Exactly the same sequence of operations was performed for all different implementations compared. A clean-cache operation was performed just before each run.

The results from the experiments with up to 8 threads are shown in Fig. 2. The benchmarks have been executed on an Intel Core i7 920 2.67 GHz with 6 GB DDR3 1333 MHz system running Windows 7 64-bit. This processor has 4 cores, capable of executing 2 threads each. The results of benchmarks 1-2 show the number of successful (*Dequeue*s finding the queue empty are not counted) operations executed per second in the system in total. The results of benchmarks 3-4 show the number of items per second that have passed through the queue (i.e., the number of successful *Dequeue* operations). In all of the benchmarks, the two array-based implementations perform significantly better than the other implementations. The worse performance of the other implementations compared to the static array-based implementation can be

[3]We require only that *CAS* operations are atomic and that each *CAS* operation behaves as a memory barrier for the thread's own reads and writes to memory.

[4]CAS is faster than using memory barriers on the platform we used.

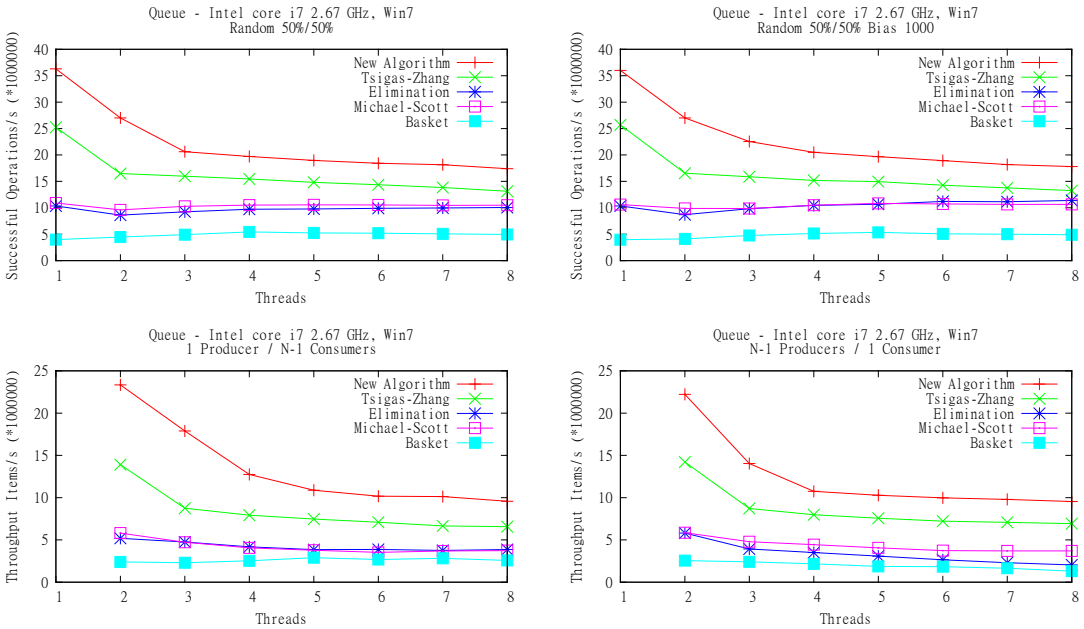

Figure 2: Experiments on a 8-way Intel Core i7 processor system.

explained to be mainly due to the costs of having dynamic allocation of nodes. Interestingly, the dynamic implementation of the algorithm described here performs significantly better than the implementation with a static array. This can be explained by the benefits of the cache-awareness (also causing fewer shared updates) apparently being significantly higher than the corresponding costs of having dynamic allocation of array blocks.

5. CONCLUSIONS

We have described an algorithm for implementing a lock-free queue data structure, originally presented in [3]. To the best of our knowledge, this is the first lock-free queue algorithm with all of the following properties:
(i) Cache-aware algorithmic handling of shared pointers including lazy updates to decrease communication overhead.
(ii) Linked-list of arrays as underlying structure for efficient dynamic algorithmic design.
(iii) Exploitation of thread-local static storage for efficient communication.
(iv) Fully dynamic in size via lock-free memory management.
(v) Lock-free design for supporting concurrency.
(vi) Algorithmic support for weak memory consistency models, allowing more efficient implementation on contemporary hardware.

The algorithm has been shown to be lock-free and linearizable. Experiments on a contemporary multi-core platform show significantly better performance for the algorithm compared to previous state-of-the-art lock-free implementations. We believe that our implementation should be of highly practical interest to contemporary and emerging multi-core and multi-processor system thanks to its high performance, its strong progress guarantees, and its support for weak memory consistency models. We are currently incorporating it into the NOBLE [11] library.

Acknowledgments

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706 and 13671-60582-29.

6. REFERENCES

[1] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and practice of parallel programming (PPoPP '08)*, pages 43–52. ACM, 2008.

[2] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. on Parallel and Distributed Systems*, 20(8):1173–1187, 2009.

[3] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proc. of the 14th Int. Conf. on Principles of Distributed Systems (OPODIS 2010)*, 2010. To appear.

[4] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[5] L. Higham and J. Kawash. Impact of instruction re-ordering on the correctness of shared-memory programs. In *Proc. of the 8th Int. Symp. on Parallel Architectures, Algorithms and Networks*, pages 25–32. IEEE, 2005.

[6] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proc. of the 11th Int. Conf. on Principles of Distributed Systems (OPODIS 2007)*, volume 4878 of *LNCS*, pages 401–414. Springer, 2007.

[7] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[8] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[9] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th annual ACM Symp. on Principles of distributed computing*, pages 267–275. ACM, 1996.

[10] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of the 17th annual ACM Symp. on Parallelism in algorithms and architectures (SPAA '05)*, pages 253–262. ACM, 2005.

[11] H. Sundell and P. Tsigas. Noble: non-blocking programming support via lock-free shared abstract data types. *SIGARCH Comput. Archit. News*, 36(5):80–87, 2008.

[12] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. of the 13th annual ACM Symp. on Parallel Algorithms and Architectures (SPAA '01)*, pages 134–143. ACM, 2001.