Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency

Anders Gidenstam[1], Håkan Sundell[1], and Philippas Tsigas[2]

[1] School of Business and Informatics, University of Borås, Borås, SWEDEN
[2] Department of Computer Science and Engineering, Chalmers University of Technology, 412 96 Göteborg, SWEDEN.

Abstract. A lock-free FIFO queue data structure is presented in this paper. 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 new algorithm compared to previous fast lock-free 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 presents an efficient lock-free queue data structure for multiple producers and consumers. The algorithm is cache-aware in order to minimize its communication overhead. It works also on weak memory consistency models (due to out-of-order execution) without need for additional fence [4] instructions for reads and writes done in the algorithm towards the shared memory.

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 evolving 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* [3]. *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 correctly 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 strengthens the lock-free condition to ensure 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 paper, 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. [6][1][11][8][9][5] 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. This paper improves on previous results by combining the underlying approaches and designing the new algorithm cache-aware and tolerant to weak memory consistency models in order to maximize efficiency on contemporary multi-core platforms. The new 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 new 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 presents the new algorithm. The corresponding proofs and analysis are outlined in Section 4. In Section 5, some benchmark experiments are described. Finally, Section 6 concludes this paper.

2 Related Work

Lamport [6] presented a lock-free (actually wait-free) implementation of a queue based on a static array, with a limited concurrency supporting only one producer and one consumer. In this algorithm, 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 [11] presented a lock-free extension of [6] where synchronization is done both directly on the array elements and the shared head and tail indices using *CAS*[3], 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

[3] 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 mostly directly by hardware and in other cases in combination with system software.

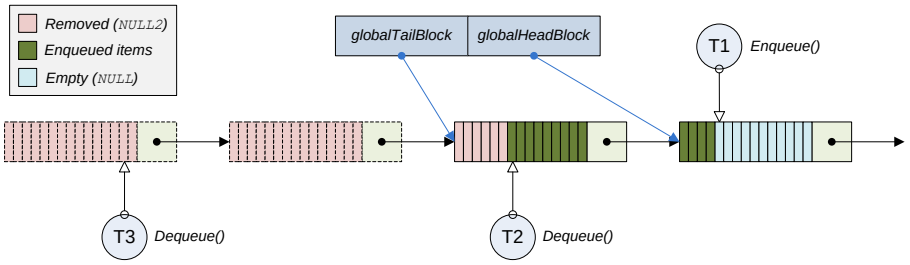

Fig. 1. A lock-free queue implemented using a linked list of arrays, where each thread is avoiding accesses to global pointers in order to reduce number of cache misses.

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 [6][1][11] the new algorithm uses arrays to store (pointers to) the items, and in resemblance to [11] it uses *CAS* and two null values. Moreover, shared indices [1] are avoided and scanning [11] is preferred as much as possible. In contrast to [6][1][11] 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 [8] 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. [7]. Moir et al. [9] 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. [5] 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 [7] and a strategy to avoid long chains of logically deleted nodes.

In resemblance to [8][9][5] the new algorithm is dynamic, and in resemblance to [5] removed blocks are logically deleted, blocks are being traversed and creation of long chains are avoided. In contrast to [9][5] the new algorithm employs no special strategy

Program 1 The functionality supported by the memory management scheme.

```
node_t * NewNode(int size);
void DeleteNode(node_t *node);
node_t * DeRefLink(node_t **link);
void ReleaseRef(node_t *node);
bool CASRef(node_t **link, node_t *old, node_t *_new);
void StoreRef(node_t **link, node_t *node);
```

Program 2 Callback procedures for the memory management.

```
void TerminateNode(block_t *node) {
    StoreRef(&node->next,NULL);
}
void CleanUpNode(block_t *node) {
    block_t *next = DeRefLink(&node->next);
    block_t *next2 = DeRefLink(&globalTailBlock);
    CASRef(&node->next, next, next2);
}
```

for increasing scalability besides allowing disjoint *Enqueue* and *Dequeue* operations to execute in parallel.

3 The New Algorithm

The underlying data structure that our algorithmic design uses is a linked list of arrays, and is depicted in Figure 1. In the data structure every array element contains a pointer to some arbitrary value. Both the *Enqueue* and *Dequeue* operations are using increasing array indices as each array element gets occupied versus removed. To ensure consistency, items are inserted or removed into each array element by using the *CAS* atomic synchronization primitive. To ensure that a *Enqueue* operation will not succeed with a *CAS* at a lower array index than where the concurrent *Dequeue* operations are operating, 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 [11] are used; one (`NULL`) for the empty indices and another (`NULL2`) for the removed indices. As each array gets fully occupied (or removed), 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 also 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 the other threads that concurrently access that pointer, the overall strategy for improving performance and scalability of the new algorithm is to avoid accessing pointers that can be concurrently updated [5]. Moreover, our algorithm achieves fewer updates by not having shared variables with explicit information regarding which array index currently being the next active for the *Enqueue* or *Dequeue*. Instead each thread is storing its

Program 3 The block structure and auxiliary functions.

```
1  struct block_t : public node_t {
2      void * nodes[BLOCK_SIZE];
3      int head;
4      int tail;
5      bool deleted;
6      block_t * next;
7  };
8  block_t * NewBlock() {
9      block_t * block = NewNode(sizeof(block_t));
10     block->next = NULL;
11     block->head = 0;
12     block->tail = 0;
13     block->deleted = false;
14     for(int i=0;i<BLOCK_SIZE;i++) block->nodes[i]=NULL;
15     return block;
16 }
17 void InitQueue() {
18     block_t * block = NewBlock();
19     StoreRef(&globalHeadBlock,block);
20     StoreRef(&globalTailBlock,block);
21 }
22 void InitThread() {
23     threadHeadBlock = DeRefLink(&globalHeadBlock);
24     threadTailBlock = DeRefLink(&globalTailBlock);
25     threadHead = threadHeadBlock->head;
26     threadTail = threadTailBlock->tail;
27 }
28 // Shared variables
29 block_t * globalHeadBlock, globalTailBlock;
30 // Thread-local storage
31 block_t * threadHeadBlock, threadTailBlock;
32 int threadHead, threadTail;
```

own[4] pointers indicating the last known (by this thread) first and active block as well as active indices for inserting and removing 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 it's beneficial compared to scanning. The *Dequeue* operation to be performed by thread T3 in Figure 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 placed next to each other in memory, the scan can normally be done without any extra cache misses (besides the ones caused by concurrent successful *Enqueue* and *Dequeue* operations) and also without any constraint 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.

For our implementation of the new lock-free queue algorithm, we have selected the lock-free memory management scheme proposed by Gidenstam et al. [2] which makes use of the *CAS* and *FAA* atomic synchronization primitives. The interface defined by the memory management scheme is listed in Program 1 and are fully described in [2]. Using this scheme we can assure that an array block can only be reclaimed when there

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

Program 4 The new Enqueue operation.

```
void Enqueue(void *item) {
    int head = threadHead;
    block_t *block = threadHeadBlock;
    for(;;) {
        if(head==BLOCK_SIZE) {
            block_t *oldBlock = block;
            block->head = head;
            block = DeRefLink(&block->next);
            if(block == NULL) {
                block = (queueblock_t *) NewBlock();
                while(globalHeadBlock != oldBlock && oldBlock->next==NULL) {
                    queueblock_t *headBlock = DeRefLink(&globalHeadBlock);
                    if(headBlock->next != oldBlock) break;
                    if(CASRef(&globalHeadBlock,headBlock,oldBlock)) break;
                }
                if(CASRef(&oldBlock->next,NULL,block))
                    CASRef(&globalHeadBlock,oldBlock,block);
                else {
                    DeleteNode(block);
                    block = DeRefLink(&oldBlock->next);
                }
            }
            else if(block->head==BLOCK_SIZE && block->next!=NULL)
                block = DeRefLink(&globalHeadBlock);
            threadHeadBlock = block;
            head = block->head;
        }
        else if(block->nodes[head]==NULL) {
            if(CAS(&block->nodes[head],NULL,item)) {
                threadHead = head+1;
                return;
            }
        }
        else head++;
    }
}
```

is no next pointer in the linked list pointing to it and that there are no local references to it from pending concurrent operations or from pointers in thread-local storage. By supplying the scheme 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 function for breaking cycles, see the *CleanUpNode* procedure in Program 2, is to update 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 *TerminateNode* procedure is called by the memory management scheme when the memory of an array block is possible to reclaim.

The specific fields of each array block are described in Program 3 as it is used in this implementation. Note that the linked list data structure always contains at least one array block. Note also that the additional fields head and tail in the array block are only used for indicating either fullness or emptiness of the whole array, and not any intermediate status. In order to simplify the description of our new algorithm, we have omitted some of the details of applying the operations of the memory management [2].

Program 5 The new Dequeue operation.

```
void * Dequeue() {
    int tail = threadTail;
    block_t *block = threadTailBlock;
    for(;;) {
        if(tail==BLOCK_SIZE) {
            block_t *oldBlock = block;
            block->tail = tail;
            block=DeRefLink(&block->next);
            if(block == NULL)
                return NULL;
            else {
                if(!oldBlock->deleted) {
                    while(globalTailBlock != oldBlock && !oldBlock->deleted) {
                        block_t *tailBlock= DeRefLink(&globalTailBlock);
                        if(tailBlock->next != oldBlock) continue;
                        if(CASRef(&globalTailBlock,tailBlock,oldBlock))
                            DeleteNode(tailBlock);
                    }
                    if(CAS(&oldBlock->deleted,false,true)) {
                        if(CASRef(&globalTailBlock,oldBlock,block))
                            DeleteNode(oldBlock);
                    }
                }
                if(block->deleted)
                    block=DeRefLink(&globalTailBlock);
            }
            threadTailBlock = block;
            tail = block->tail;
        }
        else {
            void *data = block->nodes[tail];
            if(data==NULL2)
                tail++;
            else if(data==NULL && CAS(&block->nodes[tail],NULL,NULL)) {
                threadTail = tail;
                return NULL;
            }
            else if(CAS(&block->nodes[tail],data,NULL2)) {
                threadTail = tail+1;
                return data;
            }
        }
    }
}
```

In actual implementations, *ReleaseRef* calls should be inserted at appropriate places whenever a variable holding a safe pointer goes out of scope or is reassigned.

The *Enqueue* operation is described in Program 4. After scanning for the first empty (i.e., an array element containing `NULL`) array index, 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:

1. 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.

2. 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, henceforth the operation continues scanning for an empty array index in that block.

The *Dequeue* operation is described in Program 5. After scanning for the first non-empty (i.e., an array element with neither `NULL` or `NULL2`) array index, 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. If `NULL` is found during scanning, the queue is (after also ensuring the `NULL` value to be globally consistent using *CAS*) 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:

1. If the global tail pointer is pointing to the current array block, it tries to logically mark the block as deleted using a *CAS*.
2. 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 Correctness and Analysis

In this section we show that the new queue algorithm is linearizable and lock-free. Line numbers given for actions in *Enqueue* operations refer to Program 4, while line numbers for actions in *Dequeue* operations refer to Program 5. Due to space limitations some of the detailed proofs have been omitted in this version of the paper.

Assumption 1 (Memory order) *All CAS operations are atomic.*

A CAS operation behaves as a memory barrier for a thread's memory reads and writes. All reads and writes done before the CAS in program order are committed to memory before the CAS takes effect and none of the reads and writes following a CAS are visible in memory before the CAS takes effect.

Definition 1. *The linearization point of an Enqueue operation is the successful CAS at line 29 in Enqueue.*

Definition 2. *The linearization point of a Dequeue operation is either:*
i) the CAS at line 34 in Dequeue (Program 5) iff `NULL` *is returned; or*
ii) the successful CAS at line 38 in Dequeue otherwise.

4.1 Properties of an array block

Definition 3. *A **full** array block is a block where all array elements have been changed from* NULL *to another value (i.e., there is no array element with value* NULL*). An array block is **marked full** when its* head *field is set to* BLOCK_SIZE.

Definition 4. *An **emptied** array block is a block where all array elements have been changed to* NULL2*. An array block is **marked emptied** when its* block.tail *field is set to* BLOCK_SIZE.

Lemma 1 (Block array element life cycle). *An array element in a block can change value at most two times during the life time of the block in the following order:*
i) first from initial value NULL *to an item; and subsequently*
ii) from an item to NULL2.

Lemma 2 (Thread-local head lag). *The thread-local static variable threadHead is never ahead of the true head index (i.e., the index of the first* NULL *value in the block) of the block at the starting point of an Enqueue operation.*

Lemma 3 (Thread-local tail lag). *The thread-local static variable threadTail is never ahead of the true tail position (i.e., the index after the last* NULL2 *in the block) of the block at the starting point of a Dequeue operation.*

4.2 Properties of the chain of array blocks

Definition 5. *An **active** array block is a block that has been created, has been published in a shared variable (i.e., in globalHeadBlock, globalTailBlock or the next pointer) and not yet been marked as deleted by setting the block's deleted flag.*

Definition 6. *A **valid** array block is a block that has been created and has not (yet) become reclaimable.*

Lemma 4 (Block next pointer). *The next pointer in an active block initially contains* NULL *and can change at most once while the block is active, from* NULL *to a pointer to a new block.*

Lemma 5 (Unique head block). *At any time there is exactly one valid block that has a next pointer with the value* NULL.

Lemma 6 (At least one active block). *There is always at least one active block in the queue.*

Lemma 7 (globalHeadBlock). *The global variable globalHeadBlock always points to either:*
i) the block at the head of the chain of blocks; or
ii) the block immediately before the head of the chain of blocks.

Lemma 8 (globalTailBlock). *The global variable globalTailBlock always points to either:*
i) the first active block in the chain of blocks; or
ii) the block immediately before the first active block in the chain of blocks.

4.3 Linearizability

Lemma 9 (Linearizability I). *The operation Enqueue is linearizable with respect to other Enqueue and Dequeue operations with linearization points according to Definition 1 and Definition 2.*

Proof. First observe that from Lemma 5 and Lemma 6 there is always a well defined array block at the head of the chain of array blocks.

Consider two concurrent *Enqueue* operations $Enq_1(A)$ and $Enq_2(B)$, enqueuing the elements A and B respectively. According to Lemma 2 we can, without loss of generality, assume that both operations start with their threadHead variables set to 0. Both operations do a linear search for the first array element in the block at the head of the chain of blocks that contains NULL and will try to update that array element using CAS (line 29 in Program 4). Only one can succeed and that *Enqueue* will be linearized at that point. The other will retry from line 4.

Consider an *Enqueue* operation $Enq(A)$ and a concurrent *Dequeue* operation Deq. The critical case is when the queue is initially empty. According to Lemma 2 and Lemma 3 we can, without loss of generality, assume that the operations start with their threadHead and respectively threadTail variables set to 0. Assume towards a contradiction that Deq returns A despite being linearized before $Enq(A)$. The contradiction is obvious since there is no way that Deq can return A before A is written into the array block, which occurs at the linearization point of $Enq(A)$ (line 29 in Program 4).

For the opposit case assume towards a contradiction that Deq returns NULL despite being linearized after $Enq(A)$. To return NULL Deq must traverse the array block until it finds NULL. In particular, it must have read the first index that contained NULL, which is where $Enq(A)$ will write A using *CAS* (*Enqueue* line 29). Since *CAS* is atomic according to our assumption on memory order a read returning NULL must have occured before the *CAS*. Since this read is the linearization point of Deq we have a contradiction with the assumption that Deq was linearized after $Enq(A)$. □

Lemma 10 (Linearizability II). *The operation Dequeue is linearizable with respect to other Dequeue and Enqueue operations with linearization points according to Definition 1 and Definition 2.*

Proof. Consider two *Dequeue* operations, Deq_1 and Deq_2 on a non-empty queue. The operations will first search the first active block, via their threadTailBlock variables and globalTailBlock, where the latter is guaranteed to point to the first active block or the block immediately before it by Lemma 8. Once a *Dequeue* has reached the first active block it will scan it, looking for an array element that is not NULL2. If such an array element is found the Deq operation tries to change that element to NULL2 using *CAS* (line 38). Assume towards a contradiction that the Deq_1 returning B is linearized before Deq_2 returning A where A was enqueued before B (in the same array block). From Lemma 9 we know that A is in an array element with lower index than B. Since *Dequeue* only scans past NULL2 values (line 32), Deq_1, which must have scanned past the index of A to reach B, must have read NULL2 from A's array element. According to our memory order assumption all local memory reads that precede a CAS must have occured before the *CAS*. Hence, Deq_1 read NULL2 from the array element of A before

its linearization point. From Lemma 1 we know that an array element can only change to NULL2 once which contradicts our assumption that Deq_2 which is linearized after Deq_1 returns A.

Consider two *Dequeue* operations, Deq_1 and Deq_2 on a queue containing exactly one item A. Assume towards a contradiction that Deq_1 returns NULL despite being linearized before Deq_2 returning A. As above by Lemma 1 NULL can only occur at a higher array element index than that of A and consequently Deq_1 have to read NULL2 from that location before its *CAS* operation from NULL to NULL at line 34 succeeds giving a contradiction.

That *Dequeue* is linearizable with respect to concurrent *Enqueue* operations is shown in the proof of Lemma 9 above.

Note that the scan procedure in *Dequeue* is performing speculative reads that might have taken effect out of program order. If the scan was performing at least one search step, the preceding speculative reads in the steps before the last step must have read the NULL2 value (as line 34 must have been executed). These speculative NULL2 reads must have taken effect before the last atomic NULL read during the *CAS* at line 34, as the *CAS* implies a memory barrier and must have taken effect after the previous speculative reads. □

4.4 Lock-freedom

Lemma 11 (Lock-free I). *The operation Enqueue is lock-free.*

Proof. The *Enqueue* operation contains two nested loops. There are three cases to consider:

First consider the case where threadHeadBlock points to a block that is not marked full. According to Lemma 2 the value of the threadHead variable will be smaller or equal to the index of the first NULL value in the block when the *Enqueue* operation starts. The operation will finish if it finds an array element in the block containing NULL and successfully puts its item there using a *CAS*.The index it looks at increases in each iteration, except when an unsuccessful *CAS* occurs, something that according to Lemma 1 can only happen once per array element. Thus the search index will reach the end of the block after at most 2*BLOCK_SIZE iterations and would find a free array element if there is any left. That is, progress is made unless concurrent operations fill the block first. If the block is found to be full the next iteration will mark the block full (line 7) and continue in one of the cases below.

Second, consider the case where threadHeadBlock points to a block that is marked full and has a next pointer that isn't NULL. Finding out that the block is full takes at most BLOCK_SIZE iterations. After that *Enqueue* will read the full block's next pointer into block (line 8). Since block isn't NULL the *Enqueue* operation tests if the new block is marked as full (line 23). If it is full and isn't the last block (i.e., block.next is not NULL) the *Enqueue* operation moves to the block that globalHeadBlock points to. According to Lemma 7 this is either the last or second last block of the chain. If block is full and is the last block the next iteration will enter case three below.

Third, consider the case where threadHeadBlock points to a block that is full (oldBlock) and has a next pointer that is NULL. This case proceeds as the second case until the *Enqueue* reads the oldBlock.next pointer to be NULL at line 8. When it does that, it enters

the inner loop at line 11. To remain in the loop globalHeadBlock must not be equal to oldBlock and oldBlock must remain the last block in the chain. Further globalHeadBlock must point to the block before oldBlock at line 13 and not at line 14 since the *CAS* would succeed and exit the loop otherwise. With Lemma 7 in mind this can clearly only occur once since in the next iteration either globalHeadBlock is equal to oldBlock or, if globalHeadBlock has moved further, oldBlock.next is not NULL anymore. Past the inner loop the *Enqueue* tries to add a new block. Regardless of whether it succeeds or not the next iteration of the outer loop will be done on a new block. □

Lemma 12 (Lock-free II). *The operation Dequeue is lock-free.*

Proof. The *Dequeue* operation contains two nested loops. There are three cases to consider:

First consider the case where threadTailBlock points to a block that has not been marked emptied. According to Lemma 3 the value of the threadTail will be smaller or equal to the index of the first value not equal to NULL2 in the block when the *Dequeue* operation starts. At worst the operation has to search from the beginning of the block (i.e., threadTail was 0). Each array element in the block is read (line 31 in *Dequeue*) and depending on the value found at the current array element the operation either moves to the next array element if the value was NULL2 (line 33), returns NULL (line 36) if the value was NULL and then verified to be NULL by the *CAS* at line 34, or attempts to change the value of the location to NULL2 using a *CAS* (line 38). If the *CAS* succeeds the removed item is returned, otherwise the *Dequeue* operation will do another iteration in which it will move to the next array element in the block (since according to Lemma 1 the only possible reason for the *CAS* to fail is that a concurrent *Dequeue* operation changed the value to NULL2). In all at most 2*BLOCK_SIZE iterations of the outer loop is required to either find and successfully dequeue an item or find the block emptied. If the block is found to be emptied the next iteration will mark the block emptied (line 7) and continue in one of the cases below.

Second, consider the case where threadTailBlock points to a block that has been marked emptied and has a next pointer that is NULL. In this case the queue is empty and NULL is returned (line 10).

Third, consider the case where threadTailBlock points to a block that has been marked emptied and has a next pointer that is not NULL. In this case the current block is referenced by oldBlock (line 6) and its next pointer is read into block (line 8). There are two cases depending on whether oldBlock is marked deleted or not (line 12). If oldBlock is marked deleted the *Dequeue* operation checks if the next block is also marked deleted (line 24) in which case it moves directly to the block that globalTailBlock points to, which according to Lemma 8 is the first active block of the queue or the block immediately before it. Otherwise the *Dequeue* moves the next block (which at least was active at line 24). If oldBlock is not marked deleted the *Dequeue* will enter the inner loop (line 13). To remain in this loop, the variable globalTailBlock has to be different from oldBlock and oldBlock must not be marked deleted. From Lemma 8 we know that globalTailBlock points to the first active block or the block immediately before that. At the time the inner loop is entered oldBlock is the first active block so globalTailBlock is the block immediately before oldBlock or else the loop would not be entered (since

globalTailBlock would be equal to oldBlock). In this case the *CAS* at line 16 in this or a concurrent *Dequeue* can advance globalTailBlock to oldBlock and terminate the loop. Further, Lemma 8 shows that it is impossible to advance globalTailBlock past oldBlock without marking oldBlock deleted and thereby making sure the inner loop cannot continue. Once clear of the inner loop the *Dequeue* tests if the next block is marked deleted (line 24) and acts as described above, continuing with either the next block or the block pointed to by globalTailBlock. □

4.5 Concurrent FIFO queue

Theorem 1. *The algorithm implements a lock-free and linearizable FIFO queue data structure.*

Proof. The minimal set of operations[5] necessary for implementing a FIFO queue is consisting of the *Enqueue* and *Dequeue* operations. Correspondingly, given by Lemmas 11 and 12 our implementation is lock-free, and given by Lemmas 9 and 10 our implementation is linearizable.

5 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 new algorithm compares with other known lock-free queues under high contention and increasing concurrency. The benchmarks are the following:

1. Random 50%/50%. Each thread is randomly (the sequence is decided in forehand) executing either an *Enqueue* or a *Dequeue* operation.
2. Random 50%/50% Bias 1000. Performed as the previous benchmark, besides that the queue is initialized with 1000 items.
3. 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*.
4. N-1 Producers / 1 Consumer. Same as the previous benchmark, with the producer and consumer distributions interchanged.

For comparison we have also implemented the dynamic lock-free queues by Michael and Scott [8], ditto with elimination [9], the baskets queue [5], and the static cyclic array lock-free queue presented in [11]. All dynamic queues (including the new algorithm) have been implemented to support queue sizes only limited by the system's memory, i.e., using lock-free management schemes [7] or [2] and lock-free free-lists where appropriate. For the new implementation, the size of the array block (`BLOCK_SIZE`) 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

[5] If required, operations as *Peek* and *IsEmpty* can be derived straight-forwardly out of the *Dequeue* algorithm by omitting the update part of the *CAS* operation in line 38 combined with other minor changes.

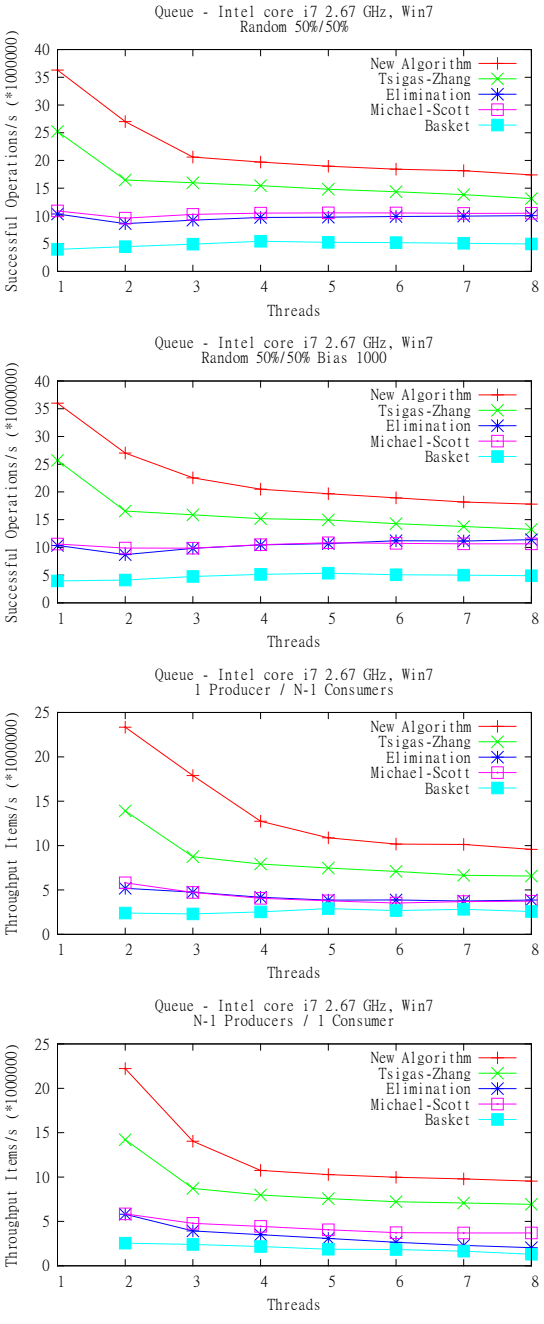

Fig. 2. Experiments on a 8-way Intel Core i7 processor system.

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 also performed just before each run.

The results from the experiments with up to 8 threads are shown in Figure 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 (failed *Dequeue*s 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 explained to be mainly due to the costs of having dynamic allocation of nodes. Interestingly, the new dynamic implementation 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 arrays.

6 Conclusions

We have presented a new algorithm for implementing a lock-free queue data structure. To the best of our knowledge, this is the first lock-free queue algorithm with all of the following properties:

- Cache-aware algorithmic handling of shared pointers including lazy updates to decrease communication overhead.
- Linked-list of arrays as underlying structure for efficient dynamic algorithmic design.
- Exploitation of thread-local static storage for efficient communication.
- Fully dynamic in size via lock-free memory management.
- Lock-free design for supporting concurrency.
- Algorithmic support for weak memory consistency models, resulting in 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 new 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 both its high performance, its strong progress guarantees, and its support to weak memory consistency models. We are currently incorporating it into the NOBLE [10] library.

7 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.

References

1. Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). pp. 43–52. ACM, New York, NY, USA (2008)
2. Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P.: Efficient and reliable lock-free memory reclamation based on reference counting. IEEE Transactions on Parallel and Distributed Systems 20(8), 1173–1187 (2009)
3. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 11(1), 124–149 (Jan 1991)
4. Higham, L., Kawash, J.: Impact of instruction re-ordering on the correctness of shared-memory programs. In: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks. pp. 25–32. IEEE (Dec 2005)
5. Hoffman, M., Shalev, O., Shavit, N.: The baskets queue. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2007). Lecture Notes in Computer Science, vol. 4878, pp. 401–414. Springer (2007)
6. Lamport, L.: Specifying concurrent program modules. ACM Trans. Program. Lang. Syst. 5(2), 190–222 (1983)
7. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems 15(8) (Aug 2004)
8. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 267–275. ACM Press (1996)
9. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures (SPAA '05). pp. 253–262. ACM, New York, NY, USA (2005)
10. Sundell, H., Tsigas, P.: Noble: non-blocking programming support via lock-free shared abstract data types. SIGARCH Comput. Archit. News 36(5), 80–87 (2008)
11. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01). pp. 134–143. ACM press (2001)