This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

Queue Delegation Locking

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad

Abstract—The scalability of parallel programs is often bounded by the performance of synchronization mechanisms used to protect critical sections. The performance of these mechanisms is in turn determined by their sequential execution time, efficient use of hardware, and ability to avoid waiting. In this article, we describe *queue delegation (QD) locking*, a family of locks that both delegate critical sections and enable detaching execution. Threads *delegate* work to the thread currently holding the lock and are able to *detach*, i.e., immediately continue their execution until they need a result from a previously delegated critical section. We show how to use queue delegation to build synchronization algorithms with lower overhead and higher throughput than existing algorithms, even when critical sections need to communicate results back immediately. Experiments when using up to 64 threads to access a shared priority queue show that QD locking provides 10 times higher throughput than Pthreads mutex locks and outperforms leading delegation algorithms. Also, when mixing parallel reads with delegated write operations, QD locking outperforms competing algorithms with an advantage ranging from 9.5% up to 207% increased throughput. Last but not least, continuing execution instead of waiting for the execution of critical sections leads to increased parallelism and better scalability. As we will see, queue delegation locking uses simple building blocks whose overhead is low even in uncontended use. All these make the technique useful in a wide variety of applications.

Index Terms—Locking, Synchronization, Delegation, Detached execution, Multi-core, NUMA

1 Introduction

Lock-based synchronization is a simple way to ensure that shared data structures are always in a consistent state. Threads synchronize on a lock, and only the current holder can execute a critical section on the protected data. To be efficient, locking algorithms aim to minimize the time required to acquire and release locks when not contended and the lock handover time when locks are contended.

Queue-based locks, like MCS [28] and CLH [7], [27], try to minimize the handover time by reducing cache coherence traffic. However, these locks strictly order the waiting threads, which harms performance when thread preemption is common. Moreover, on NUMA systems MCS and CLH are outperformed by less fair locks that exploit the NUMA structure, e.g., the HBO lock [33], the hierarchical CLH lock [26] or more recently the *Cohort lock* [9]. These locks let threads on a particular NUMA node execute critical sections for longer periods of time without interference from threads on other nodes. This avoids expensive coherence traffic between NUMA nodes for the lock and the memory it protects, but not between the cores within a node.

In this article we focus on a different approach, which sends critical sections to the lock data structure instead of transferring the lock. This way, a single thread can execute multiple critical sections without transferring the data between caches of different cores or NUMA nodes. This locking approach is called *delegation*, and the thread performing other threads' critical sections is called *helper*. Put simply, delegation lets the operation come to the data while traditional locks let the data come to the operation. This can result in reduced cache misses since delegation makes it possible to let many operations on the shared data execute one after another on a single processor core. Besides these benefits, some algorithms employ *detached execution*, i.e., they allow threads to continue execution before the delegated critical section has been executed. In its original form [31], the detaching algorithm has some overhead and severe starvation issues for the helper thread. Newer approaches, like flat combining [17] or remote core locking [25], steered away from detached execution in favor of faster delegation and in order to provide a simpler semantics. As we show in this article, in contrast to these earlier approaches our locking mechanism allows efficient delegation while it also permits detaching execution without starving the helper thread.

Main Ideas. We introduce *Queue Delegation (QD) locking*, a new efficient delegation algorithm whose idea is simple. When a lock is contended, the threads do not wait for the lock to be released. Instead, they try to delegate their operation to the thread currently holding the lock (helper). If successful, the helper is responsible for eventually executing the operation. The threads may either wait for the operation to complete or alternatively continue their execution immediately, possibly delegating more operations.

Delegated operations are placed in a *delegation queue*. As the queue preserves FIFO order, the correct order of operations is maintained, and QD locking ensures linearizability. The linearization point is the successful enqueueing into the delegation queue. However, the enqueueing can fail when the lock holder is not accepting any more operations. This allows the helper to limit the amount of work it performs, providing starvation freedom for the helper, and ensures that no operations are accepted when the lock is about to be released. If delegation fails, a thread has to retry until it succeeds to either take the lock itself or delegate its operation to a new lock holder. The QD locking algorithm thus puts the burden of executing operations on the thread that succeeds in taking the lock. After performing its own operation, this thread must perform, in order, all operations it finds in the delegation queue. When it eventually finds no more operations in the delegation queue, it must make sure that no further enqueue call succeeds before the lock is released.

To communicate return values from a delegated operation

• *The authors are with the Department of Information Technology, Uppsala University, Sweden. Email: firstname.lastname@it.uu.se*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

to the thread that delegated it, the operation may pass an address to store the return value as well as the address of a flag. The helper then has to store the return value before setting the flag, on which the delegating thread is waiting. This also means that, optionally, threads that do not require a return value can continue their execution immediately after a successful delegation. We refer to the former option as *delegate and wait* and the latter option plainly as *delegate*.

All requirements for QD locking are met by assembling two simple components: (i) a *mutual exclusion lock* to determine which thread is executing operations, and (ii) a *queue* to delegate operations to the lock holder. By using a *reader indicator* as an optional third component it is also possible to allow multiple readers to efficiently execute in parallel.

Contributions. The main contribution of this article is a detailed description of a new delegation algorithm that we call QD locking. It is novel in that it efficiently delegates operations while also allowing to detach the delegation from the eventual execution. We discuss its prerequisites and properties in detail, introduce *multi-reader QD locks* which allow multiple parallel readers, and a *hierarchical QD locking* variant which targets NUMA systems. Last but not least, we quantify the performance and scalability aspects of QD locking by comparing them against state-of-the-art scalable synchronization mechanisms. As we will see, QD locking offers performance that is on par and often much better than that of existing synchronization algorithms.

Overview. The rest of the article is structured as follows. The next section reviews the necessary background in mutual exclusion algorithms and related work. The following four sections present the different QD locking variants (Section 3), their implementation (Section 4), their properties (Section 5), as well as how the algorithm is extended to support reader-writer locking semantics (Section 6). Subsequently, QD locking is compared with related synchronization mechanisms on the experimental level (Section 7), and the article ends with some concluding remarks.

2 Background and Related Work

This section discusses various algorithms that can be used to implement mutual exclusion. Starting with the most simple locking algorithms, we take a look at their abilities step by step. Furthermore, we showcase other algorithms that employ helper threads to improve performance, highlighting their respective features and limitations.

2.1 Locking

Locking (also called mutual exclusion), has been introduced by Dijkstra in 1965 [11]. Since then it has been a dominating synchronization mechanism for concurrent programming with shared memory. The most basic algorithm is the *test-and-set* (TAS) lock, which uses a simple flag to indicate the state of the lock and functions `lock()` and `unlock()` to modify this state as shown in Fig. 1. Atomic test-and-set is used to ensure that only one thread can take the lock at a time. Two additional functions are commonly also available. The first, `is_locked()`, returns true if the lock is currently taken and false otherwise. The second, `try_lock()`, takes the lock if it is available and then returns true, but does not wait (block) when it is unavailable; instead it simply returns false.

```
class tas_lock {
  std::atomic<bool> locked; /* or atomic_flag */
  void lock() { while(test_and_set(&this->locked)); }
  void unlock() { this->locked = false; }
  bool is_locked() { return this->locked; }
  bool try_lock() { return !test_and_set(&this->locked); }
};
```

```
class tatas_lock {
  std::atomic<bool> locked;
  void lock() { while(!this->try_lock()); }
  void unlock() { /* ... */ }
  void is_locked() { /* ... */ }
  bool try_lock() {
    if(this->locked) return false;
    return !test_and_set(&this->locked);
  }
};
```

Fig. 1. The test-and-set (TAS) and the test-and-test-and-set (TATAS) lock.

While the TAS lock is a functionally correct locking algorithm, its performance is lacking: When multiple threads try to take the lock at the same time, i.e., there is contention for the lock, each thread will repeatedly issue write-accesses to the *locked*-flag. This causes the flag to be removed from any other private caches it may currently be in, and thereby causes lots of cache coherence traffic. This traffic can be reduced by first checking the flag before writing to it. The resulting *test-and-test-and-set (TATAS)* algorithm, also shown in Fig. 1, avoids repeatedly writing when the lock is taken, but will still cause traffic by the waiting threads whenever the lock is unlocked. However, when locks are usually uncontended this algorithm provides decent performance [1].

2.2 Dealing with Oversubscription

One of the problems TATAS does not solve is that waiting threads can take away CPU cycles from the thread currently holding the lock. The probably easiest way to avoid this is by adding exponential backoff so that this happens less often [1]. Another option is to tell the operating system to

```
class futex_lock {
  enum status { free, taken, contended };
  std::atomic<status> locked;
  void lock() {
    int c = free;
    if(!this->locked.compare_exchange_strong(c, taken)) {
      if(c != contended) { c = this->locked.exchange(contended); }
      while(c != free) {
        sys_futex_wait(&this->locked, contended);
        c = this->locked.exchange(contended);
      }
    }
  }
  void unlock() {
    int old = this->locked.exchange(free);
    if(old == contended) { sys_futex_wake(&this->locked); }
  }
  bool is_locked() { return this->locked != free; }
  bool try_lock() {
    if(this->is_locked()) return false;
    int c = free;
    return this->locked.compare_exchange_strong(c, taken);
  }
};
```

Fig. 2. The *futex* lock.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

wake up the thread when the lock is released, which on Linux is used by glibc's implementation of the ubiquitous Pthreads mutex lock [12], [15], [16]. Fig. 2 shows how this can be implemented using the (Linux-specific) *futex* syscall. This *futex* lock uses three internal states: free, taken and contended. It only involves the operating system when there is contention, which causes the state to be set to contended. Otherwise it functions as the TAS and TATAS locks, using free and taken as the flag's values. As syscalls can be expensive, another option is to attempt to take the lock multiple times before going into the contended state. This allows higher performance if the waiting time is expected to be low at a limited computational cost [15].

2.3 Fairness

All locking algorithms described so far lack fairness: When a lock is released, all threads waiting for it have a chance of obtaining the lock. The probability of obtaining the lock is largely dependent on the hardware properties, but there is markedly no upper limit for the time a thread may have to wait until it obtains the lock.

A simple way of obtaining fairness is to issue tickets to threads by fetch-and-incrementing a field in the lock [28]. A thread has obtained the lock when the number it received matches a second field representing the next ticket to be called. On `unlock()`, threads simply increment this second field, allowing the next thread to proceed. However, this approach is memory intensive: All threads waiting need to fetch the next ticket field whenever it is updated, but only one of them will be allowed to proceed.

This issue is solved by queue-based locks that allow each thread to wait on a separate memory address instead. The earliest such locks are MCS [28] locks, which have a queue node per thread. The threads compare-and-swap their node's pointer with an initially null field when calling `lock()`. The first thread obtains null as the previous value and is allowed to proceed its critical section, while subsequent threads obtain the previous thread's node pointer instead. They then reset a flag in their own node and set a next-pointer in the previous thread's node. When unlocking, the thread tries to compare-and-swap the MCS lock field back to null, which fails if another thread arrived in the meantime. In that case, the thread follows its node's next-pointer to set the flag in the next thread's node, waking only that thread.

A further improvement of this scheme is the CLH [7], [27] lock, which does not require a swap operation for unlocking. (It takes a spare node from the lock which is not returned, but reused as the thread's node on the next `lock()` operation.)

Queue-based schemes improve performance over ticket locks due to the lower cache coherence traffic. However, the strict ordering of waiting threads can harm performance when thread preemption is common: The next thread to execute may have to wait for processing time first. In Fig. 3 the performance of CLH is compared with the ubiquitous Pthreads lock and the Cohort lock, which is described below. Note that, in all graphs, the x-axis (# threads) is in log scale.

2.4 NUMA-aware Algorithms

On Non-Uniform Memory Access (NUMA) systems, the strict FIFO ordering of threads using CLH locks becomes a performance bottleneck. The locking algorithm needs to synchronize the lock's state between the thread unlocking the lock and the one subsequently locking it. Such synchronization slows down when information needs to be communicated to another processor chip. It thus follows that hierarchical approaches are required for faster lock handovers in NUMA systems, at the cost of a potentially lower degree of fairness. Several NUMA-aware locking algorithms that exploit the hierarchical structure of NUMA systems exist by now [8], [26], [33]. Recently Dice *et al.* have proposed *lock cohorting* as a general mechanism to create NUMA-aware hierarchical locks [9]. A *Cohort lock* has one lock for every NUMA node in the system and an additional global lock. The holder of the global lock and a local lock has the right to execute a critical section and hand over the local lock to a waiting thread, if there is one, provided that the hand-over limit has not been reached. This reduces the number of expensive memory transfers that have to be done between NUMA nodes. The performance of the Cohort lock is also shown in Fig. 3. Chabbi *et al.* recently generalized Cohort locks for deep NUMA hierarchies [5]. The resulting hierarchical MCS (HMCS) lock allows to take advantage of locality within each different NUMA level individually.

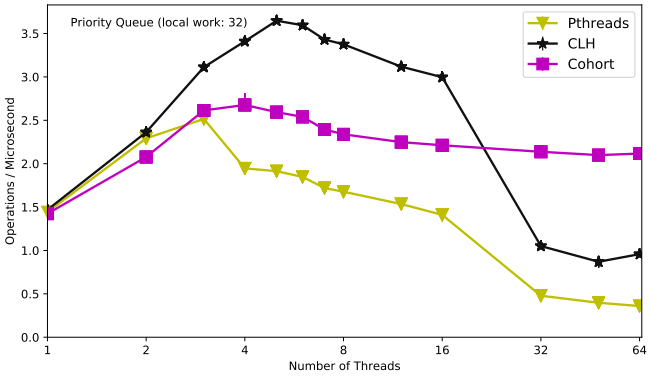

Fig. 3. Throughput for a concurrent priority queue implemented using different locks. The benchmark runs a loop in each worker thread that executes a critical section followed by a number of thread-local computations. The first 16 threads are pinned to a single processor socket. Additional threads run on additional sockets and thus use a NUMA system. (Refer to Section 7 for more details about the benchmark and the platform used.) CLH performs best on non-NUMA, while Cohort locks maintain their performance when using more sockets.

NUMA-aware approaches are much more efficient on NUMA machines under high contention than traditional locks. However, their performance is inferior to other approaches, such as those based on delegation, when operating within only one NUMA node (cf. Fig. 4 in Section 2.6).

2.5 Detached Execution

The poor cache locality due to data transfers between private caches was identified by Oyama *et al.* [31] back in 1999 as a major bottleneck for locking algorithms. To improve this, they suggested that threads should not execute critical sections themselves, but they should be *detached* from their execution. While the threads lose control over the execution of their critical section, the scheme allows a single thread to execute critical sections efficiently due to the ability to exploit data locality. The algorithm for detached execution by Oyama *et al.* stores critical sections in a LIFO queue-based on a linked

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

list. A thread that successfully executes a compare-and-swap (CAS) instruction on a lock word that also functions as head of the LIFO queue becomes the *helper*.

Not discussed by Oyama *et al.* is that their detached critical sections cannot directly return data to the thread detaching them. If threads require knowledge about the success of a critical section or need to read data in a critical section to guarantee consistency, the scheme requires additional synchronization. For example, a communication variable could be defined, which can then be used store the desired information in. Read-spinning on and writing to this communication variable must still ensure consistency.

Another problem in the algorithm by Oyama *et al.* is that the helper continues executing requests that are put by other threads as long as the LIFO queue is not empty. Therefore, it is possible for the helper to starve while executing requests for other threads. Furthermore, threads delegate operations by performing a CAS operation on the pointer to the LIFO queue. As noted by other researchers [14], [17], this pointer can become a contended hot spot which limits scalability. The original algorithm also suffers from the cost of memory management of LIFO queue nodes, which can be mitigated by pre-allocation of nodes. The resulting variant of the algorithm of Oyama *et al.* is called *DetachExec* in this article.

Despite these known problems, DetachExec can achieve some performance benefits through the ability to *detach* worker threads from the execution of their critical sections: If the programmer does not explicitly synchronize with the end of a critical section, threads will continue with their execution immediately after storing their operation in the LIFO queue.

2.6 Delegation Algorithms

The idea of a single thread executing many critical sections without any need for synchronization in between has been used by a number of approaches, which we call delegation algorithms. For consistent terminology, we say that worker threads *delegate* their critical sections to a *helper* thread. The following algorithms all differ from DetachExec in that they ensure that threads continue execution only when the delegated critical section has been executed.

Flat Combining. One approach to coalesce operations on a shared data structure into a single thread is *flat combining* (FC) [17]. Flat combining uses a lock and a list of request nodes L. Each thread has a single request node that can be put into L. To perform an operation on the shared data structure, the operation is first published on the thread's request node. Subsequently, the thread spins in a loop that switches between checking whether the response value has been written back to its request node and trying to take the lock. A thread that successfully acquires the lock becomes the helper (in FC also called *combiner*), traverses the list of requests (a number of times), and performs the requests it finds there. The FC algorithm also has a way of removing nodes that have not been used for a long time from the list of requests. A thread that is waiting for a response has to occasionally check that its node is still in the list and put it back with a CAS operation if it has been removed. Flat combining's delegation mechanism has been shown to perform better than the original algorithm of Oyama *et al.* for contended workloads [14], [17], but these comparisons do not consider an optimized variant of the algorithm, like DetachExec, that is better suited for modern machines.

Synch Algorithms. CC-Synch, DSM-Synch, and H-Synch are queue-based delegation algorithms developed by Fatourou and Kallimanis [14]. In all three algorithms, a thread T announces an operation by inserting its queue node, which contains the operation, at the tail of the request queue. T then needs to wait on a flag in the queue node until the flag is unset. If T then sees that the operation is completed it can continue normal execution, otherwise T becomes the *helper*. A helper thread first performs its own operation and then traverses the queue performing all requests until it reaches the end of the queue or a limit is reached.

The queue in the CC-Synch algorithm is based on the CLH lock [7], [27], while the queue in DSM-Synch is based on the MCS lock [28]. CC-Synch is slightly more efficient than DSM-Synch, but DSM-Synch is expected to also work well on systems without cache coherence. H-Synch is in spirit similar to lock cohorting, and has one CC-Synch data structure on every NUMA node and an additional global lock. Threads put their queue node in the Synch data structure located at their local NUMA node and a helper needs to take the global lock before starting to execute operations.

The Synch algorithms have been shown to perform better than flat combining for implementing queues and stacks [14].

Dedicated Core Locking. Locking mechanisms where cores are dedicated to only execute critical sections for specified locks have been studied both from a hardware and a software angle. Suleman *et al.* have proposed hardware support for the execution of critical sections [35]. Their suggested hardware has an asymmetric multi-core architecture where fat cores are dedicated to critical sections and have special instructions to execute them. *Remote core locking* [25] is a software locking scheme where one processor core is dedicated to execution of critical sections. The dedicated core spins in a loop checking a request array for new requests. All seen requests are executed and the response value or acknowledgment is written back in a provided memory location. Compared to delegation mechanisms, dedicated core locking has the disadvantage that the programmer has to decide which locks shall have a dedicated core and the cores that should be used for this. Furthermore, dedicated core locking is not well-suited for applications that have different phases where the lock is sometimes contended and sometimes not. Finally, remote core locking suffers from the same kind of overhead that flat combining has in that it needs to scan request nodes even when they are empty.

Hardware Message Passing. Recently, several algorithms using hardware message passing to optimize cache coherence interactions have been proposed [32]. The algorithms include an MCS lock variant called HybLock and two delegation algorithms, called mp-server and HybComb. While mp-server uses dedicated cores for execution of critical sections, HybComb promotes an existing thread to execute critical sections, similar to FC and CC-Synch. All three algorithms rely on a hardware message queue to avoid negatively affecting performance through the cache coherence protocol.

Delaying Updates. In update-heavy scenarios it can be beneficial to not update the global state until there is a read that needs to see the updates. OpLog [3] is such an approach. It uses core cycle timestamps provided by the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

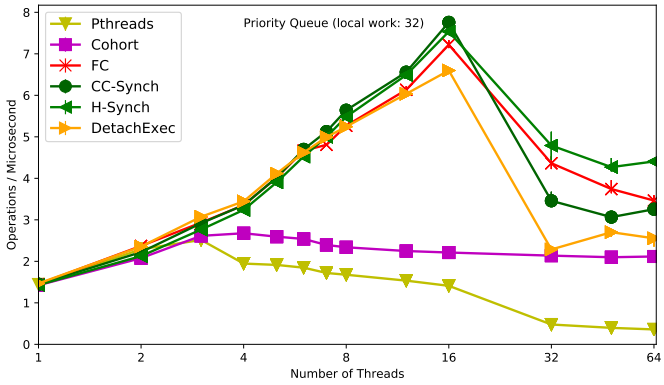

Fig. 4. Throughput for the same priority queue benchmark as in Fig. 3 with various delegation algorithms. Note that the y-axis scale has changed.

CPU to establish a total order of operations, which is shown to perform well in several Linux kernel use cases.

The performance of some delegation algorithms discussed above is shown in Fig. 4. For comparison, the Pthreads and Cohort locks are also shown. It is clear that all delegation algorithms perform better than traditional locking algorithms that do not use a *helper* thread to execute critical sections.

3 Queue Delegation Locks

This section describes the different QD locking variants. We start with the necessary components, and then use them to assemble a basic QD lock. We also show that QD locks can be used in a way that gives programmers more flexibility and allows them to achieve better performance than the one using only basic QD locks. Then we extend the algorithm to a multi-reader QD locking variant that allows multiple read-only operations to execute in parallel. Last, we sketch how QD locks can also provide the functionality of traditional mutual exclusion locks.

3.1 Building Blocks

Queue delegation locks are built from two main components: a *mutual exclusion lock* and a *delegation queue*.

The mutual exclusion lock is used to determine whether the lock is free or taken. QD locks can use most locking algorithms, as long as they provide a `try_lock` function in addition to the standard `lock`/`unlock` mechanisms.[1] We also use the `is_locked` function of the mutual exclusion lock when we later build a multi-reader QD lock.

The other building block, the delegation queue, is required to store delegated operations. Semantically, it is a *tantrum queue* as described by Morrison and Afek [30]. Calls to its enqueue operation are not guaranteed to succeed, but can return a *closed* value instead. This allows the QD lock to stop accepting more operations. The required interface for the delegation queue consists of only three functions: `open`, `enqueue` and `flush`. The first two are simple: `open` resets the queue from closed state to empty, and `enqueue` adds an element to the queue. The `flush` function is used instead of a dequeue operation: it dequeues all elements (performing their operation) and changes the queue's state to closed.

1. In fact, the `lock` mechanism for taking the lock unconditionally is not strictly required. Still, `lock` is useful in order to provide stronger guarantees and makes it easier to adapt legacy code to use QD locking.

3.2 Queue Delegation Lock

We use the building blocks outlined above to assemble a QD lock. A thread attempting to execute an operation under the QD lock first attempts to lock (using `try_lock`) the mutual exclusion lock. If this succeeds, the thread will first open the delegation queue (which then accepts other threads' operations). Then the operation will be executed, followed by a `flush` operation on the delegation queue. Finally, the mutual exclusion lock is unlocked again. However, if the `try_lock` fails, the thread instead attempts to `enqueue` its operation into the delegation queue. When this succeeds, the thread has detached the critical section. At that point, the thread can either wait for the critical section to complete or continue its execution (and possibly wait for the result of the critical section at a later point). The resulting QD lock therefore accepts operations even when the mutual exclusion lock is locked; threads only need to retry if the mutual exclusion lock is locked and the queue is closed.

An operation that can be submitted to QD locks is semantically a self-contained *function object*, which means it needs to store all required parameters from the local scope when delegated, similar to a *closure*. For implementation details of this mechanism, see Fig. 6 and 7.

The basic QD lock interface only consists of a `delegate_and_wait` function which takes an operation as an argument. It also creates a flag (initially set to false) and augments the operation with a final step that sets the flag to true. The call to `delegate_and_wait` then waits until this flag is true. This is semantically equivalent to the way the delegation algorithms in Section 2.6 operate.

However, the `delegate_and_wait` function can also be separated into `delegate` and `wait` functions, which can be called individually. It is then still guaranteed that the delegated operation will be executed before any operations from subsequent calls to `delegate` are executed, but the execution is detached until `wait` is called. Additional performance can be leveraged by a thread that delegates multiple operations before waiting for them. Finally, if the operation has no return value that needs to be obtained, the `wait` call can be omitted entirely. In this case it is also not required to create and set a flag that can be waited on, which further reduces the overhead incurred by the algorithm.

This separation of `delegate` and `wait` can be explained using the semantics of *futures* [2]. Namely, the value is not returned immediately, but the `delegate` call can promise to provide the value at a specific location upon the execution of the operation, i.e., it returns a future. When the calling thread needs to read the return value, it has to wait until it is available. The waiting can be done by the programmer by calling `wait`, or by hiding the call in a wrapper that does it automatically when the value is read.

3.3 Multi-Reader Queue Delegation Lock

Readers-writer locks can be built from mutual exclusion locks in a generic way. This is applicable to QD locking as well. For Multi-Reader QD (MR-QD) locks, which is our variant of readers-writer locks, a third building block is required: an indicator that shows whether there are any threads currently holding the lock in reading mode. Note that it is not necessary to count the readers. Instead, all we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

need is a `query` function [13] that returns true when there are readers and false when there are none. Readers use the functions `arrive` and `depart` to indicate when they start and stop reading. This is needed so that delegated operations can wait until it is safe to write.

A reader indicator extends QD locks to MR-QD locks, which allow many concurrent readers that efficiently execute in parallel when there is no writer or when the writer releases the lock. The interface of an MR-QD lock contains the additional functions `rlock` and `runlock` that work as in traditional readers-writer locks.

3.4 Queue Delegation Locks with a Wider Interface

We can easily extend the interface of QD and MR-QD locks to allow critical sections that are not delegated or to offer other functionality provided by mutual exclusion locks. To do so, we only need to expose the functions from the mutual exclusion lock component. This interface may be required for critical sections that need to lock multiple locks and release them in an order other than last in, first out, or because a critical section needs to run in a specific thread. We do not further discuss or evaluate this kind of extended QD locks, as their performance depends mostly on the performance of the mutual exclusion lock.

4 Implementation

We will now describe how QD locks can be realized by presenting our implementation. However, we note that the range of possible implementations of QD locking is not restricted to the ones we describe in this article. In fact, the components we use can be replaced by other ones providing the required interface. The ones we chose to base our implementation on may not be the "best" (whatever that means), but are easy to understand and, as we will see, provide good performance and scalability.

4.1 Mutual Exclusion Lock

The mutual exclusion lock component is not as important for QD locking as one might expect. As delegating operations is preferred over waiting for the lock, there is virtually no write contention on the lock. The basic algorithm only uses `try_lock`, while locking algorithms mainly differ in their way to wait for taking the lock. Our implementation works very well with the *TATAS* lock, whose implementation was shown in Fig. 1. However, as we will show in Section 5, the guarantees of the lock used here can be used to extend similar guarantees to the entire QD lock. Thus, for our implementation, we chose the *MCS-futex* lock, whose implementation is shown in Fig. 5. It is a simple extension of MCS locks that use futex syscalls to wait for their turn instead of spin-waiting. Additionally, there is a `try_lock` function (equivalent to the one in the futex lock), which does not provide fairness. There is also a function `try_lock_or_wait`, which acts similar to `try_lock` but before returning false also makes threads sleep until the helper finishes all operations delegated to it and calls `unlock`. This allows spin-waiting for a limited period before sleep-waiting, which is a common optimization.

This MCS-futex lock provides a `lock` function, which is used in Section 5 to provide starvation freedom. It can also be used in the QD lock interface to deal with situations where delegation is not possible, like unlocking in an order other than last in, first out as mentioned in Section 3.4.

```
1  class mcsfutex_lock {
2    thread_local static std::map<mcs_futex_lock*, mcs_node>
3      mcs_node_store; // each thread gets one node per lock
4    std::atomic<mcs_node*> locked;
5    void lock() {
6      mcs_node* mynode = &mcs_node_store[this];
7      mynode->next = NULL;
8      mcs_node* c = this->locked.exchange(mynode);
9      if(c != NULL) {
10       mynode->is_locked = true;
11       c->next = mynode;
12       while(mynode->is_locked) {
13         sys_futex_wait(&mynode->is_locked, true);
14       }
15     }
16   }
17   void unlock() {
18     mcs_node* mynode = &mcs_node_store[this];
19     mcs_node* c = mynode;
20     if(mynode->next == NULL) {
21       if(this->locked.compare_exchange_strong(c, NULL)) {
22         if(this->sleep) {
23           this->sleep = false;
24           sys_futex_wake(&this->sleep);
25         }
26         return;
27       }
28     }
29     while(mynode->next == NULL) { /* wait for nextpointer */ }
30     mynode->next->is_locked = false;
31     sys_futex_wake(&mynode->next->is_locked);
32   }
33   bool is_locked() { return this->locked != NULL; }
34   bool try_lock() { return try_lock_may_wait(false); }
35   bool try_lock_or_wait() { return try_lock_may_wait(true); }
36   bool try_lock_may_wait(bool wait) {
37     if(this->is_locked()) return false;
38     mcs_node* mynode = &mcs_node_store[this];
39     mcs_node* c = NULL;
40     if(!this->locked.compare_exchange_strong(c, mynode)) {
41       if(wait) {
42         if(!this->sleep) { this->sleep = true; }
43         sys_futex_wait(&this->sleep, true);
44       }
45       return false;
46     }
47     return true;
48   }
49 };
```

Fig. 5. The MCS-futex lock.

4.2 Delegation Queue

On the other hand, the delegation queue component is important for QD locking since it is used by all contending threads. It therefore must be fast when enqueueing operations.

Our delegation queue implementation, shown in Fig. 6, uses a fixed-size buffer array to store operations. A counter is used to keep track of how many elements are already in the queue. The queue is defined to be closed when the counter is greater than or equal to the size of the array. Initially, the counter is set to a value greater than the size of the array and thus the queue is in closed state. The `enqueue` function increases the counter using an atomic *fetch_and_add* instruction[2] (line 10), which gives each delegated operation

2. As not all platforms have a *fetch_and_add (FAA)* instruction, we've also done experiments where *FAA* is simulated with a *CAS* loop (Sect. 7).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems


```
1  class delegation_queue {
2    std::atomic<long> counter;
3    std::atomic<Function> entries[ENTRIES];
4    char array[ENTRIES * ENTRY_SIZE];
5    void open() { counter = 0; }
6    bool enqueue(Function f, Parameters p) {
7      int size = sizeof(int) + sizeof(p);
8      int req = (size+ENTRY_SIZE-1)/ENTRY_SIZE; /* ceil */
9      if(counter > (ENTRIES - req)) return CLOSED;
10     int index = counter.fetch_and_add(req);
11     if(index <= (ENTRIES - req)) {
12       int os = index*ENTRY_SIZE;
13       memcpy(&array[os], &req, sizeof(int)); /* non-atomic */
14       os = os + sizeof(int);
15       memcpy(&array[os], &p, sizeof(p)); /* non-atomic */
16       entries[index].store(f); /* atomic */
17       return SUCCESS;
18     } else {
19       if(index < ENTRIES) {
20         int os = index*ENTRY_SIZE;
21         int r = ENTRIES-index;
22         memcpy(&array[os], &r, sizeof(int)); /* non-atomic */
23         entries[index].store(no_op); /* atomic */
24       }
25       return CLOSED;
26     }
27   }
28   void flush() {
29     long todo = 0;
30     bool open = true;
31     while(open) {
32       long done = todo;
33       todo = counter;
34       if(todo == done) { /* close queue */
35         todo = counter.exchange(ENTRIES);
36         open = false;
37       }
38       if(todo >= ENTRIES) { /* queue closed */
39         todo = ENTRIES;
40         open = false;
41       }
42       for(int index = done; index < todo; ) {
43         Function f = entries[index];
44         while(!f) {
45           Function f = entries[index];
46         }
47         int req = *((int*)&array[index]);
48         int offset = index + sizeof(int);
49         Parameters* p = &array[offset];
50         (*f)(*p);
51         index += req;
52       }
53     }
54     std::fill(&entries[0], &entries[todo], 0);
55   }
56 };
```

Fig. 6. The delegation queue implementation.

its index in the buffer array. This way, the queue automatically closes when the buffer fills up, and can also be closed by atomically changing the counter field with a *CAS* or a *swap* instruction (as in line 35). The `flush` function repeatedly reads the counter and dequeues operations until the queue has been put to closed state by an `enqueue` function (line 38 detects this) or because the counter has not been updated since the last check (line 34).

Special care is needed when writing and reading the delegated operation; i.e., in lines 12–16 and 43–49. First of all, the operation needs to be self-contained: besides the operation, all parameters needed to execute it must be provided. In our implementation, we use a function pointer in the `entries` array, a size field and a variable-sized parameter field for each operation. Each function pointer entry is associated with a fixed amount of buffer space in the parameter array. If more space is needed, multiple entries can be used for a single function. The amount of required entries is stored in the size field, which always comes at the beginning of the associated buffer space. To ensure that no partially-written operations are used, the function pointer is written last and read first. The entries array content is reset to all zero (line 54) after the delegated operations have been executed. This allows to wait in the next iteration of the `for` loop until the function pointer entry contains a valid value. As the function pointer entry is written last, the entire operation can be read safely when the function pointer entry has a non-zero value.

The observant reader may wonder whether the counter field in the delegation queue could also be used for deciding which thread becomes the helper. While this is possible and gives high performance, it is then not possible to use the additional properties (fairness and sleeping while waiting) provided by an MCS-futex lock. When such properties are desired, a separation of lock and delegation queue allows simplified reasoning about them, as we will see in Section 5.

4.3 Queue Delegation Lock Implementation

With the mutual exclusion lock and the tantrum queue available, only the actual delegation functions have to be provided to build a QD lock. The `delegate_and_wait` function provides an interface that enforces the stronger semantics of most other locking algorithms. It extends the operation with a flag that will be set after the operation has been executed. This flag will be waited on before `delegate_and_wait` returns. The extended operation is handled by the `delegate` function, which can also be used by the programmer directly. As can be seen in Fig. 7, this function initially checks whether the lock is contended (line 5), so that it can avoid overheads if there is no contention. It then alternates between trying to delegate the operation (line 11) and trying to acquire the lock (lines 12–18) until one of them succeeds or the maximum number of tries is reached. If the `enqueue` function call succeeds, it is guaranteed that the operation will be executed and the `delegate` function can return. As described earlier, a `delegate` caller that does not need a return value from the operation can just continue execution at this point. An operation that requires a return value needs to write this value to a location that the caller of `delegate` can wait on. If the `try_lock` call succeeds, the thread opens the queue, executes its own operation and all enqueued operations until the queue is closed. Finally it unlocks the mutual exclusion lock. However, if the maximum number of tries is reached (in line 10) then the algorithm reverts to using the mutual exclusion lock directly (line 27), which will achieve progress if a starvation-free lock is used.

5 Properties

Let us now discuss some of the properties of QD locking; most notably starvation freedom and linearizability. We also show how to extend QD locking to be better suited for NUMA systems, and discuss issues related to how QD locks can be used for practical programming.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems


```
1  class qd_lock {
2    mcsfutex_lock lock;
3    delegation_queue queue;
4    void delegate(Function f, Parameters p) {
5      if(lock.try_lock()) { /* check for contention */
6        f(p);
7        lock.unlock();
8        return;
9      }
10     for(int i = 1; i <= MAX_TRIES; i++) {
11       if(queue.enqueue(f,p)) return;
12       bool lock_acquired;
13       if(i % (TRIES_WITHOUT_WAIT + 1) != 0) {
14         lock_acquired = lock.try_lock();
15       } else {
16         lock_acquired = lock.try_lock_or_wait();
17       }
18       if(lock_acquired) {
19         queue.open();
20         f(p);
21         queue.flush();
22         lock.unlock();
23         return;
24       }
25     }
26     /* maximum retries reached, revert to internal lock */
27     lock.lock();
28     f(p);
29     lock.unlock();
30   }
31   Returntype delegate_and_wait(Function f, Parameters p) {
32     std::atomic<bool> flag = false;
33     Returntype r;
34     Function waiting_op = [&r,&flag](Parameters p) {
35       r = f(p);
36       flag = true;
37     };
38     delegate(waiting_op, p);
39     while(!flag) { /* wait */ }
40     return r;
41   }
42 };
```

Fig. 7. The `delegate` function and its waiting counterpart.

5.1 Starvation Freedom

The `delegate` function implementations shown in Fig. 7 and 12 are starvation free, meaning that a thread cannot get starved while executing them. When a thread fails too many times in both `try_lock` and `enqueue` calls, the algorithm uses `lock` as a fair fallback. This can happen if a thread always executes the `enqueue` function when the queue is closed and the `try_lock` function when the mutual exclusion lock is locked. According to our experience this does not seem to be a problem in practice, and experiments (in Section 7) show that the additional code does not harm performance severely.

However, one can easily replace the internal locking algorithm and remove the limit for retrying `try_lock` and `enqueue` to gain a little bit higher performance. As long as this limit exists and the mutual exclusion lock is starvation free, it is easy to see that the whole `delegate` function is also starvation free. This is because, in the worst case, the `delegate` function only does a fixed amount of work in the retry loop before it acquires the starvation free mutual exclusion lock unconditionally.

5.2 Linearizability

Linearizability [18] is a correctness criterion for concurrent data structures. Methods on a linearizable concurrent object appear as if they happen atomically at a linearization point during the methods' execution. We discuss linearization of QD locking algorithms here, dealing with the problem that detached execution allows continuing to work even before critical sections have been executed. Still, if all accesses to a data structure are protected using a lock of the QD lock family, the resulting data structure is linearizable as we argue below.

Up front we note that the delegation queue is linearizable. The `enqueue` function, if successful, linearizes delegated operations exactly in the order in which they appear in the queue. When `delegate` enqueues successfully, linearization of operations is therefore given by the linearization of the delegation queue. When `try_lock` is successful, the linearization point is just before the opening of the queue (the point between lines 18 and 19 in Fig. 7). This is true because `try_lock` can only succeed when the lock is free, which implies that any previous holder must have executed all previously delegated operations. Likewise, concurrent `delegate` calls cannot succeed before the queue is opened, thus their operations must have a linearization point later on.

For completeness, we note that `delegate` returns a future of the operation's result, not the actual result. This allows the linearization point of reading the return value to be distinct from the operation's linearization point. This linearization point is after the actual execution of the operation, right after the return value has been written successfully. It should be noted that the returned value is still consistent with the linearization of the operations and does not reorder them. This linearization model has been independently described as *strong future linearizability* [22], albeit in that paper it is restricted to data structures and does not support general critical sections. However, when queue delegation is used with operations that have side effects outside the data structure protected by the lock, linearizability is not guaranteed. In such code, the program may have to wait for the result of the future before it is safe to depend on the actual execution of the operation.

5.3 NUMA Awareness

In this section, we present a NUMA-aware hierarchical queue delegation lock called *hierarchical QD lock* (or *HQD lock* for short). The HQD lock is derived from the QD lock in a way that is in spirit similar to how Cohort locks [9] are constructed from traditional mutual exclusion locks and to how the H-Synch algorithm is constructed from CC-Synch [14]. An HQD lock uses one mutual exclusion lock and one delegation queue per NUMA node. Additionally, the lock contains a *global* mutual exclusion lock that is used to determine which NUMA node is allowed to execute operations. We have chosen an MCS lock [28] as both the global lock and the per-node mutual exclusion lock. MCS is a fair queue-based lock that provides starvation freedom. These choices guarantee that all NUMA nodes will be able to execute operations in a reasonably fair order at high performance. (We have not used futex syscalls to deal with threads that spin-wait, as this hierarchical algorithm is highly sensitive to the timing between handovers and operating system calls are too expensive in this scenario.)

The implementation for the HQD lock algorithm is shown in Fig. 8. As with other hierarchical locking approaches each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems


```
1  class hqd_lock {
2    qd_lock local_locks[NUMA_NODES]; /* MCS based */
3    mcs_lock lock;
4    void delegate(Function f, Parameters p) {
5      qd_lock* local_lock = &local_locks[my_numa_node];
6      if(local_lock->try_lock()) { /* no contention? */
7        lock.lock();
8        f(p);
9        lock.unlock();
10       local_lock->unlock();
11       return;
12     }
13     for(int i = 1; i <= MAX_TRIES; i++) {
14       if(queue.enqueue(f,p)) return;
15       bool lock_acquired;
16       if(i % (TRIES_WITHOUT_WAIT + 1) != 0) {
17         lock_acquired = local_lock->try_lock();
18       } else {
19         lock_acquired = local_lock->try_lock_or_wait();
20       }
21       if(lock_acquired) {
22         lock.lock();
23         queue.open();
24         f(p);
25         queue.flush();
26         lock.unlock();
27         local_lock->unlock();
28         return;
29       }
30     }
31     /* maximum retries reached, revert to internal lock */
32     local_lock->lock();
33     lock.lock();
34     f(p);
35     lock.unlock();
36     local_lock->unlock();
37   }
38 };
```

Fig. 8. The `delegate` function in the HQD lock.

thread needs to know which NUMA node it is running on (see line 5). The `delegate` function is using the lock and delegation queue of the local NUMA node to perform the QD locking algorithm. Additionally, it needs to take the global lock before opening its delegation queue in line 23. The same lock must also be acquired in all code paths that perform only a single critical section (lines 7 and 33). Constructed this way, the amount of expensive communication between the NUMA nodes can be significantly reduced. This allows HQD locks to perform better under high contention as we will see in Section 7. On the other hand, when the contention is low, a QD lock can perform better than an HQD lock on NUMA systems. QD locks can achieve higher parallelism at a higher communication cost compared to HQD locks. Threads on other NUMA nodes have to wait instead of delegating and continuing with their local work, which itself can limit performance. Also, it means there are less workers supplying the lock holder with additional work, which can mean the lock is released and taken again instead of the lock holder helping other threads. It is therefore not the case that HQD is always a better choice than QD on NUMA systems.

5.4 More Threads than Hardware Supports

In high-performance applications, programs often spawn exactly as many worker threads as the hardware can efficiently run at the same time. However, a general purpose mutual exclusion algorithm should also consider the case

```
1  MyStruct data; // global shared
2  futex_lock l;
3  void foo() {
4    Value v = calculate_value();
5    l.lock()
6
7    data.insert(v);
8    l.unlock();
9  }
```

```
1  MyStruct data; // global shared
2  qd_lock q;
3  void foo() {
4    Value v = calculate_value();
5    q.delegate_and_wait(
6      [](Param p) {
7        data.insert(p);
8    }, v);
9  }
```

Fig. 9. Code tranformation of a regular lock to a QD lock.

```
1  futex_lock l;
2  void bar() {
3    Key k = calculate_key();
4
5    Value v1;
6    l.lock()
7
8
9    v1 = data.lookup(k);
10   l.unlock();
11
12   Value v2 = calculate_value();
13
14
15   if(v1 > v2) do_something();
16 }
```

```
1  qd_lock q;
2  void bar() {
3    Key k = calculate_key();
4    std::atomic<bool> f(false);
5    Value v1;
6    q.delegate(
7      [&f, &v1](Param p) {
8        v1 = data.lookup(p);
9        f = true;
10   }, k);
11   // may run parallel to lookup
12   Value v2 = calculate_value();
13   // wait for detached operation
14   while(!f); // cf. Fig.7, l.39
15   if(v1 > v2) do_something();
16 }
```

Fig. 10. Code transformation with detached execution using a QD lock.

where there are more threads contending for the lock than hardware supports. In this case, locking algorithms need to cooperate with the operating system by putting themselves to sleep while waiting, in order to avoid that threads waiting to access the lock prevent execution of the thread holding it. In QD locking, this is not an issue as long as delegations succeed. However, once the delegation queue is filled up, the other threads can waste resources by repeated attempts to delegate, which are bound to fail until the helper finishes with its current workload. To limit the extent of spinning, the implementation of QD locks can use `try_lock_or_wait` to make threads sleep until the helper finishes, and there is thus a chance of progress again. This has been implemented for the Linux operating system using `futex` syscalls within the mutex lock's `try_lock` call. Measurements have shown that the performance impact of this extension is reasonably small, but can increase performance significantly when there are more threads than cores, especially when threads are not pinned to cores. However, this ability does not extend to the spinning done by threads waiting for return values, as waking them up individually causes significant overheads. Fortunately, each of them spins on its own flag which can be cached and therefore interferes little with the rest of the system as long as there is only a limited number of threads spin-waiting for a return value.

5.5 Usage Examples

The use of QD locks is straightforward. A comparison with `lock`/`unlock`-based critical sections is shown in Fig. 9. The critical section is turned into a lambda function, so that it can be delegated to a QD lock. By using `delegate_and_wait` the code guarantees that the critical section has completed. To also benefit from detached execution, the example in Fig. 10 uses `delegate` in conjunction with a boolean flag `f` to signal availability of the return value `v1`. The lambda function

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

captures both `f` and `v1` by reference so that it can write to them. This allows the lookup in the critical section lambda (line 8) and an expensive computation (line 12) to be executed in parallel when the operation is delegated to another thread. Waiting on the flag (line 14) immediately before the value is needed guarantees that `v1` has been written to by the helper thread. If the current thread happens to become a helper, the execution would still be serial, but this will only affect one thread at a time, benefitting all other worker threads.

5.6 Structured Locking and Practical Considerations

In many situations one may need to use structured locking with multiple nested locks. For QD locking, the case of delegating to one lock while executing a critical section under another lock is not problematic. If there is already a helper for the inner lock, the critical section will be delegated. When both locks are required for an operation, the outer lock's helper thread issues the inner lock's operation and then just needs to wait for its execution, thus ensuring no other operation is executed under the outer lock. However, this does not cause any additional delays over classical locking algorithms, as with them the thread would stall until it acquires the inner lock as well. In the case where there is no helper yet, the thread issuing the operation will become a helper for the inner lock as well, helping any additionally delegated calls before resuming the help session on the outer lock. This can be seen as starving the helper thread, as the maximum amount of work during the outermost help session increases exponentially with the locking depth. If this becomes a problem, mitigation is possible by spawning additional threads to become helpers. Alternatively, a flag could be passed to `delegate_and_wait` and `delegate` to indicate that the call should not become a helper. This would allow benefitting from existing helpers, but fall back to `lock/unlock` if there is no helper available. However, with this flag the performance of a call that never helps becomes harder to predict and having too many such calls can cause the scheme to degrade in performance.

More discussion on porting existing applications to use QD locks and programmability aspects of QD locking can be found in a paper describing our QD locking libraries [21].

5.7 Comparison with Other Delegation Algorithms

QD locking provides a broader set of functionality than other delegation algorithms, and there is a number of differences regarding the limitations of and choices by these algorithms.

When DetachExec by Oyama *et al.* executes critical sections, there is no upper limit for how many critical sections can be executed by the thread, causing starvation of the helper. This is not the case for QD locking which uses a tantrum queue with limited size. DetachExec has a potential hot spot where critical sections are delegated, and QD locking has a similar potential hot spot. However, it is less of an issue as we use a fetch-and-add instruction instead of a CAS loop to synchronize between threads, which reduces the already small impact of the hot spot. A final difference between DetachExec and QD locking is that in order to make DetachExec linearizable, the LIFO queue of operations needs to be reversed, which also imposes extra cost.

Compared to flat combining, the QD locking approach is offloading work to the lock's holder and does not need to wait for the critical section's execution. FC cannot easily detach execution because of happens-before relations: Another thread could synchronize with the detaching thread and issue an operation to FC, which then might be executed before the detached operation. Also, unlike FC, the helper in QD locking does not need to traverse empty request nodes which potentially can become a performance problem for FC when the number of threads is big but contention is low.

Compared to CC-Synch, DSM-Synch and H-Synch, QD locking has two advantages. First, it does not require threads to wait until an operation has been applied to the data structure that is protected by the lock. Instead, the programmer can specify if and when a return value is needed, allowing for the same strict or more relaxed semantics. If the Synch algorithms were to be extended to support detached execution, significant changes in their implementation would be required to maintain starvation freedom. Second, our implementations are using a queue that is based on an array buffer. This approach causes fewer cache misses (external loads fed from other cores) than the Synch based algorithms. Since the operations are stored continuously in the array buffer, several operations can be loaded per cache miss compared to the Synch algorithms that require one cache miss per loaded operation.

6 Multi-Reader Queue Delegation Locks

All locking algorithms reduce the amount of available parallelism, which can become a bottleneck even if the locking algorithm itself is very efficient. The most common way to mitigate this problem is to allow limited parallelism even when accessing shared data. By using readers-writer locks, programmers can allow multiple read-only critical sections to execute in parallel. This section describes how QD locks can be extended to also support parallel read-only critical sections. But let us start with some background.

The extension of mutex locks to readers-writer locks was first proposed by Courtois *et al.* in 1971 [6]. Since then, the idea has been used numerous times to extend various algorithms [4], [10], [23], [29], [34]; in particular, it has been employed on top of plain queue locks as well as other locking algorithms mentioned in Section 2. A readers-writer (RW) lock is an extension to the mutual exclusion lock that offers two levels of locking: 1) The *read* lock level is usually used for critical sections that do not modify shared data. Several *read* critical sections protected by the same lock can execute concurrently. 2) The *write* lock level is usually used for critical sections that modify shared data. If a thread is inside a *write* critical section, all other accesses (read and write) must wait for it to complete before they can proceed. Since *read* critical sections can execute concurrently, an RW lock can offer better performance than a mutual exclusion lock for applications that frequently execute read-only critical sections.

Calciu *et al.* described a general method to extend mutual exclusion locks to RW locks [4]. This method can be adapted to QD locks as well by applying the construction to `delegate`. The chosen write-preference policy extends the interface of QD locks by `rlock/runlock` functions, which neither delegate nor detach read-only critical sections. Instead, read-only

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems


```
1  class reader_indicator {
2    std::atomic<long> counters[MAX_THREADS]; /* padded */
3    bool query() {
4      for(counter : counters)
5        if(counter > 0) return true;
6      return false;
7    }
8    void arrive() { counters[ThreadID] += 1; /* atomic */ }
9    void depart() { counters[ThreadID] -= 1; /* atomic */ }
10 };
```

Fig. 11. The reader indicator.

critical sections are executed in the thread requesting them, potentially in parallel. Write-preference (up to a limit) aims to execute delegated operations (i.e., write critical sections) first, so that there is more time for read-only critical sections to arrive and execute in parallel once the mutual exclusion lock is released. The preference is limited to avoid starvation.

There are two transitions that need to be explained in some detail: (i) going from executing delegated operations to parallel reading, and (ii) going from parallel reading to a thread becoming a helper. We use the internal mutex lock's status to determine whether there is a helper thread currently executing delegated operations. If there is, all reads will defer to delegated operations unless their patience limit is reached. When a thread reaches the patience limit, it will set a barrier that prevents further `delegate` calls from proceeding, which means that only delegated operations that have been delegated before will be executed. Therefore, delegated operations are preferred over read-only critical sections, but there is no starvation of readers. For the opposite transition of parallel read-only critical sections being executed and a thread calling `delegate`, we use a reader indicator (as mentioned in Section 3.3) to wait until all read-only critical sections have been completed before executing the delegated operation. The internal mutex lock will still be locked before waiting, which guarantees that further `rlock` calls will wait until this `delegate` call has completed. Furthermore we can open the delegation queue before waiting, as this guarantee also ensures correct order of operations, even if a thread delegates an operation and then calls `rlock`.

We use a simple reader indicator algorithm; see Fig. 11. Its rationale is to not have a single counter for the readers, which would be a bottleneck, but to split that counter into several cache lines. This reduces write contention on the counter significantly, and is easy to implement. By having at least as many counters as threads, the counters become simple flags, and there is no contention at all. Checking this reader indicator requires iterating over all counters to check that they are all zero, which is relatively expensive. That cost notwithstanding, we chose this algorithm because it performed better in our experiments than the ingress-egress counter used by Calciu *et al.* [4]. The third available algorithm, SNZI [13], was not used due to its complexity. For sufficiently large systems it may be a better choice.

The implementation of multi-reader QD locks (Fig. 12) is derived from the *writer-preference readers-writer lock* algorithm [4]. In fact, functions `rlock` and `runlock` are unchanged; we include them just for completeness and refer the reader to that paper [4] for their explanation. The queue delegation algorithm itself requires only two changes to allow

```
1  class mrqd_lock {
2    std::atomic<long> writeBarrier;
3    reader_indicator indicator;
4    mcsfutex_lock lock;
5    delegation_queue queue;
6    void delegate(Function f, Parameters p) {
7      while(writeBarrier > 0) yield();
8      while(true) {
9        if(lock.try_lock()) {
10         queue.open();
11         while(indicator.query()) yield();
12         f(p);
13         queue.flush();
14         lock.unlock();
15         return;
16       } else if(queue.enqueue(f,p)) return;
17       yield();
18     }
19   }
20   void rlock() {
21     bool bRaised = false;
22     int readPatience = 0;
23   start:
24     indicator.arrive();
25     if(lock.is_locked()) {
26       indicator.depart();
27       while(lock.is_locked()) {
28         yield();
29         if(readPatience == READ_PATIENCE_LIMIT && !bRaised) {
30           writeBarrier.fetch_and_add(1);
31           bRaised = true;
32         }
33         readPatience += 1;
34       }
35       goto start;
36     }
37     if(bRaised) writeBarrier.fetch_and_sub(1);
38   }
39   void runlock() { indicator.depart(); }
40 };
```

Fig. 12. The code for multi-reader QD locks.

multiple parallel readers; refer to Fig. 12. The first is on line 7 where execution can be blocked on a write barrier to avoid starvation of readers. The second change is on line 11 where the code waits for all readers to leave their critical section.

One might be wondering why the delegation queue can be opened on line 10 while readers still can be active. With this code a read critical section can execute during the same time that one or more `delegate` calls are issued. However, this is not a problem because the delegated operations can never be executed while there are active readers; the loop in line 11 ensures this. Likewise, the lock used by the helper will prevent any readers from becoming active before all successfully delegated critical sections have been executed.

The main advantage of multi-reader QD locks compared to traditional readers-writer locks is that a writer does not need to wait for the readers. This makes it possible for writers to continue doing useful work after delegating their critical section whereas they would have to wait if a traditional readers-writer lock was used. This can also have the effect of making more read critical sections bulk up, which can increase parallelism even further.

7 Evaluation

Having compared queue delegation locking with other synchronization mechanisms on the algorithmic level, we now compare its performance experimentally. For this, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

use two synthetic benchmark sets and a benchmark program from the Kyoto Cabinet, which is a library of routines for managing a database and an application of considerable size and code complexity. The program we use is the `kccachetest` benchmark, which exercises an in-memory database that is designed to be used as a cache (cf. Section 7.3). We will compare QD lock variants mostly against the latest implementations of three state-of-the-art delegation algorithms as provided by their authors: flat combining, CC-Synch and H-Synch.[3] We will also compare against our own implementation of DetachExec, based on the pseudocode by Oyama *et al.* [31]. The main difference between the original code and DetachExec is that the latter preallocates $4{,}096$ thread local queue nodes. The queue nodes have a free flag that is changed before a queue node is delegated and after it has been executed so that they can be reused. In order to put performance numbers into perspective, we also include measurements for Pthreads, CLH and Cohort locks. The implementation of the CLH lock is taken from the Synch repository; the Cohort lock one is written by us. In the data structure benchmark we also use a recently proposed lock-free implementation of the priority queue data structure [24].

Parameters. All delegation algorithms except that by Oyama *et al.* have a `help_limit` constant that limits the number of operations that can be performed by a helper during a help session. We use the term *help session* to refer to the period of time from when a thread starts executing delegated operations to the time when it hands over this responsibility. The original flat combining (FC) implementation has two additional parameters: `num_rep` and `rep_threshold`. The `num_rep` constant decides the maximum number of traversals of the request list per help session. After every request list traversal the lock is handed over to another thread if less than `rep_threshold` operations have been performed. We also added a condition to FC that will hand over the lock to another thread if `help_limit` or more operations have been helped during the help session. The `help_limit` was set to $4{,}096$ in all experiments that we present graphs for in this article. For FC, `num_rep` was set to $4{,}096$ and `rep_threshold` was set to 1. We found these parameters to work well with all algorithms; increasing the value further gave only a very small increase in throughput.

Benchmark Environment. All benchmarks were run on a Dell server with four Intel(R) Xeon(R) E5-4650 CPUs (2.70GHz), eight cores each, with simultaneous multithreading (SMT) and TurboBoost (up to 3.30GHz) enabled. I.e., the system has a total of 64 SMT-threads running on 32 cores. The machine ran Debian Linux 3.16.0-4-amd64 and had 128GB of RAM. The two synthetic benchmarks are written in C, Kyoto Cabinet in C++, and the lock implementations in C and C++. All code was compiled using GCC version 4.9.2-10 with -O3. The duration of runs for all synthetic benchmarks was two seconds and we took measurements five times. The graphs show the average of these five runs, and bars for the minimum and maximum values we observed.

Thread Pinning. We pin threads to SMT-threads for two reasons. First, in order to avoid arbitrary thread migrations as a source of unreliability in the measurements. Second, pinning allows us to both show the performance on a single processor chip and a NUMA system in the same graph. Our pinning policy first fills all SMT-threads on one NUMA node, then on two NUMA nodes, and so on. The pinning policy on a NUMA node level aims to achieve the best performance that our machine can give when running a low number of threads: It first pins a thread to a core without any previously pinned thread (i.e., it first uses all physical cores of the NUMA node) and then fills the SMT-threads without any previously pinned thread (i.e., it then uses the hyperthreads). Measurements with up to eight threads will therefore run on the eight physical cores on a single chip on our machine. Up to 16 threads will run on a single chip with all SMT-threads occupied, up to 32 threads will run on two chips, and so on.

3. Available at https://github.com/mit-carbon/Flat-Combining and https://github.com/nkallima/sim-universal-construction.

7.1 Data Structure Benchmark

This benchmark, also used in other papers [14], [17], is used to evaluate QD locking as a way of constructing concurrent data structures. Both FC and the Synch algorithms have been shown to perform very well for this kind of task [14], [17]. Delegation is especially beneficial for data structures such as queues, stacks, and priority queues, whose operations can not be parallelized easily. The data structure we have chosen is a priority queue implemented using a *pairing heap*. This is a high-performance implementation of a priority queue that, when using flat combining, has been shown to outperform the best previously proposed concurrent implementations as well as the then fastest lock-free implementation [17]. We verify this by comparing against a more recent lock-free priority queue implementation [24], using the implementation provided by its authors. A priority queue is well-suited for comparing synchronization algorithms for a concurrent data structure because it has a natural mix of operations that do not return a value (`insert`) and operations that return one (`extract_min`). However, both are write operations.

The benchmark measures throughput: the number of operations that N threads can perform during t seconds. All N threads start at the same time and execute a loop until t seconds have passed. The loop body consists of some amount of thread-local work and a global operation (either `insert` or `extract_min`) which is selected randomly with equal probability. The seed for the random number generator is thread-local to avoid false sharing between threads. The thread-local work uses a thread-local array L with 64 integer entries. Each local work unit consists of randomly selecting two of the 64 entries of L, an additional random integer I, adding I to the value stored in the first entry, and subtracting I from the value of the second entry.

Figures 13 through 17 show the results of the benchmark for different thread counts and different amount of work between the operations. We use four scenarios in total: the case with no local work, and the cases with 32, 64 and 128 units of local work between each two accesses to the lock.

In Fig. 13, QD locks are compared with DetachExec, FC, and CC-Synch, as well as the standard CLH lock, the Pthreads mutex lock and the lock-free priority queue implementation. The graph at the top left corner shows the case with no local work. All algorithms except the lock-free one show a big performance drop when going from the sequential case to two threads. For all delegation

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

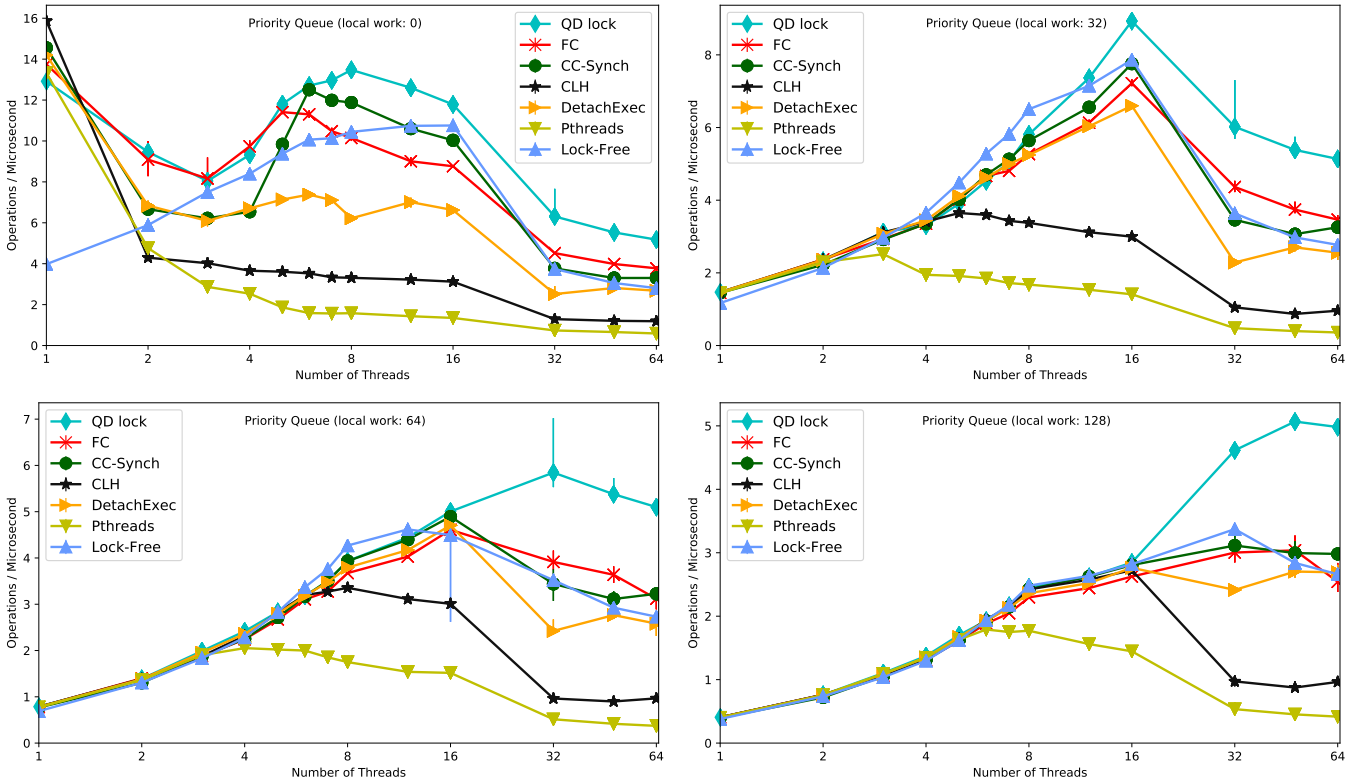

Fig. 13. Throughput for a priority queue benchmark with 50% `insert` and 50% `extract_min` operations varying the amount of local work.

algorithms except DetachExec the performance is similar to the sequential case again when running on eight cores. To investigate the reason for the performance drop with two threads, we ran experiments that collect statistics about the number of operations performed per help session. We found that with only two threads the contention is not high enough to keep the same helper for long periods of time. The data structure will therefore get transferred frequently between the two cores while the transfer happens much less frequently as the number of threads increases. DetachExec performs relatively badly compared to the other delegation algorithms in the case when there is no local work. This can be explained by the CAS loop that DetachExec uses to delegate work and by the overhead of reversing the LIFO queue. Another performance drop is apparent when going beyond 16 threads, which coincides with using more than one NUMA node in this setting. CC-Synch performs better than FC on a single NUMA node, which is reversed with more threads, but neither is designed for NUMA systems.

The graph in the top right corner of Fig. 13 shows results for 32 units of local work between the operations. In this scenario, there is some amount of thread-local work between critical sections which benefits from parallelization. Thus, the single-threaded case is no longer the optimal choice. The lock-free algorithm scales best until eight cores are used, but does not benefit from additional SMT-threads as much as delegation algorithms. Again, performance drops significantly when using multiple NUMA nodes and again the algorithms are affected roughly the same by this.

The two graphs at the bottom of Fig. 13 show a different picture. Performance between different algorithms up to 16 threads becomes more similar, and in the bottom right graph even CLH locks perform well up to 16 threads. In the bottom left graph, algorithms other than QD locks still somewhat drop in performance when using more than 16 threads while QD locks maintain their performance. Even more striking, no delegation algorithm drops in performance in the bottom right graph, but only QD locking continues to scale with more threads.

Figure 14 shows how hierarchical algorithms deal with NUMA effects. Here we see that Cohort locks maintain their performance when going beyond 16 threads, albeit at a lower level than delegation algorithms. The hierarchical delegation algorithms, HQD lock and H-Synch, outperform the other algorithms when using more than one NUMA node in the zero local work case. This is due to their ability to reduce the amount of memory transfers between NUMA nodes.

The reason why the HQD lock performs better than H-Synch and the QD lock performs better than FC and CC-Synch is twofold: First, the QD and HQD locks can delegate their `insert` operations without having to wait for them to be applied to the underlying data structure. Second, the QD and HQD locks can have fewer cache misses because approximately 50% of the operations do not need a value written back and the helper can read several operations from the delegation queue with one cache miss. This is because the operations are stored one after the other in an array buffer so that several operations can fit in a single cache line. Both flat combining and the Synch algorithms require at least one cache miss for the helper thread to read an operation and one cache miss for the thread issuing the operation to read the response value or an acknowledgment. Due to the cost of transferring the data structure between the cores on the same NUMA node, the Cohort lock is not able to perform

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

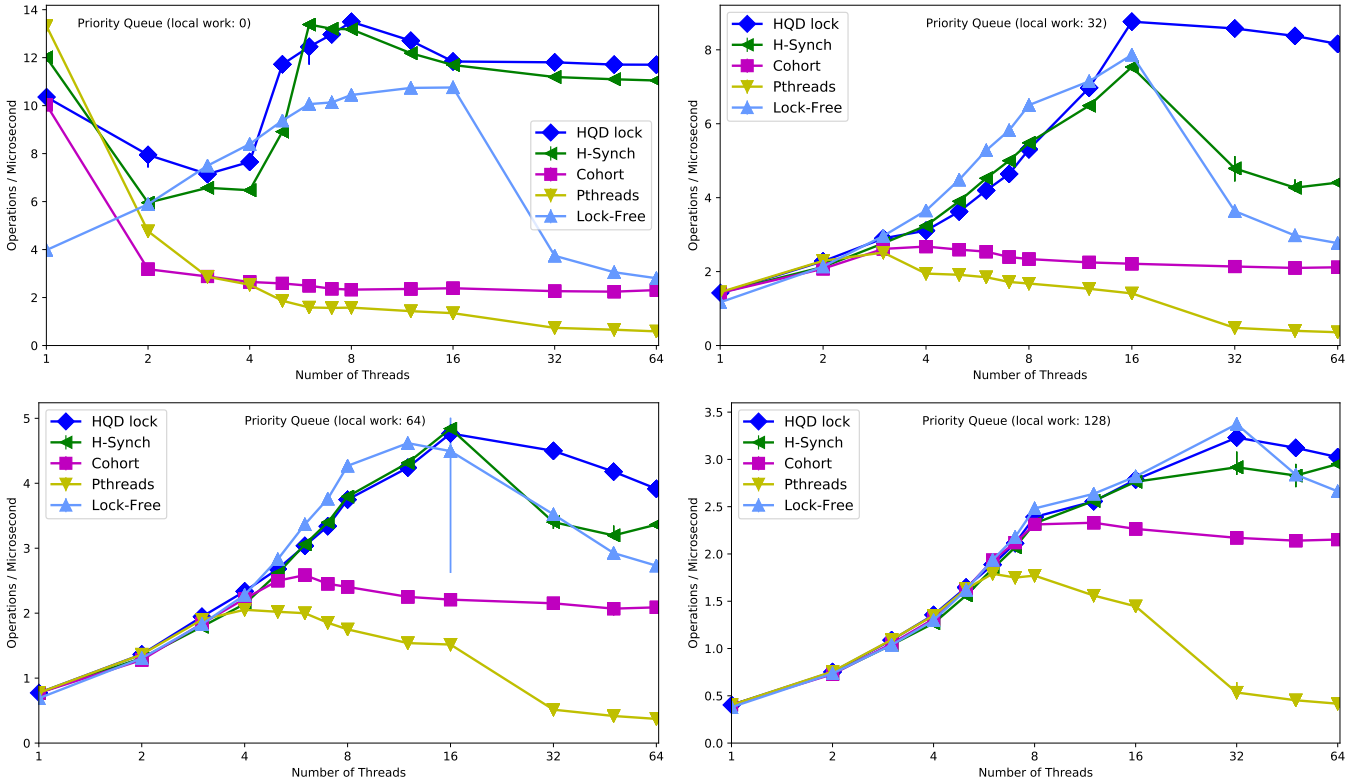

Fig. 14. Same benchmark as in Fig. 13, but showing hierarchical algorithms.

well compared to delegation algorithms on this workload. This downside of Cohort locks could potentially be mitigated by HMCS locks [5], which also aim to reduce data transfers between cores.

In the top right scenario in Fig. 14, the QD lock performs better than H-Synch even when using more than one NUMA node. The reason for this is again twofold: First, the QD lock's ability to delegate `insert` operations without waiting is more beneficial in this scenario. A thread that delegates an `insert` operation is guaranteed to be able to execute at least 32 units of work until it executes another global operation again (and possibly waits). In the zero local work scenario, there is 50% chance that a thread needs to wait for the result of an `extract_min` operation almost directly after completing an `insert` operation. Second, the drop in performance of H-Synch when using more than one NUMA node shows a problem with the hierarchical delegation approaches when contention is not high enough.

This problem is clearly illustrated in the graphs in Fig. 15. The graphs to the left of this figure show performance with varied amount of local work units. The graphs to the right show the average number of helped operations per help session instead of performance. Note that we have excluded the DetachExec lines from the graphs to the right because they show millions of helped operations per help session due to the lack of support for limiting the number of helped operations per help session in the algorithm. This means the helper thread is starved with the method by Oyama *et al.* It is clear that the drop in number of operations that are helped per help session drops earlier for the hierarchical variants than the non-hierarchical ones. The reason for this is easy to understand if one considers that the contention on a NUMA node level is lower than on the system level. The drop in operations that are helped per help session correlates well with the drop in performance for the hierarchical variants which can be explained with the increased traffic between the nodes. The two graphs at the bottom part of Fig. 14 can also be explained with a similar reasoning: More work between the operations is more beneficial for QD locks and HQD locks compared to the other locking algorithms because of the ability to delegate `insert` operations without needing to wait for a response. Because QD locks allow for more parallelism, they benefit from this effect more strongly than HQD locks. Therefore, when there is not enough contention, QD locks can outperform hierarchical algorithms.

A Deeper Look into QD Locking's Performance

The graphs in Fig. 16 and 17 investigate the performance effect of different implementation aspects of QD and HQD locks. Lines for QD and HQD locks are included in these graphs as reference points, while the other lines correspond to implementations with a twist. QD (MCS) refers to a QD lock that uses a standard MCS lock instead of an MCS-futex lock internally, while HQD (futex) refers to an HQD lock that uses an MCS-futex lock instead of an MCS lock. The QD (MCS) lock performs very similar to the standard QD lock, showing that in the scenarios measured there is no worrisome overhead for using MCS-futex locks. For the HQD (futex) lock this is different: Using an MCS-futex lock causes HQD locks to drop in performance drastically when more NUMA nodes are added. The cause here is that only one NUMA node is active at a time in HQD locks, which causes the MCS-futex locks on all other nodes to enter the sleep state. However, this means the eventual lock

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

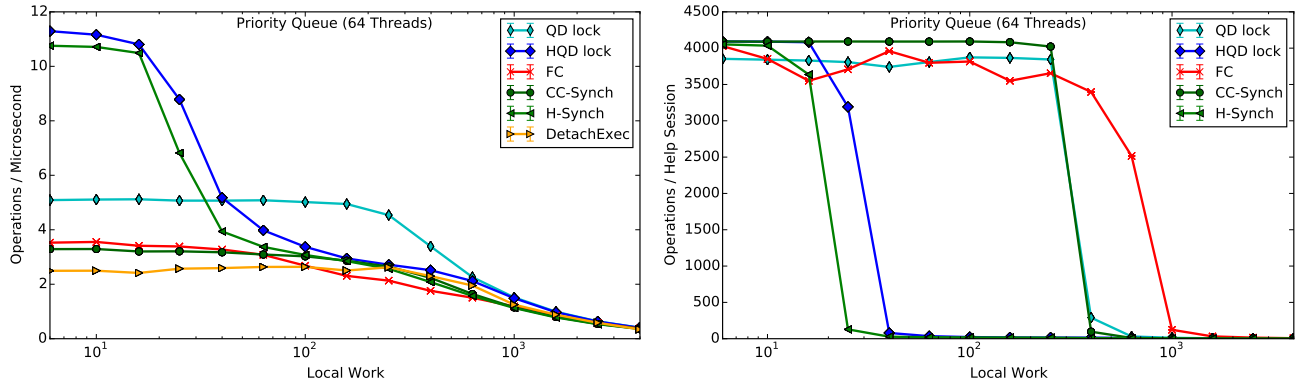

Fig. 15. The graph on the left shows operations per microsecond under different contention levels (amount of local work between operations). The graph on the right shows the average number of helped operations for all help sessions.

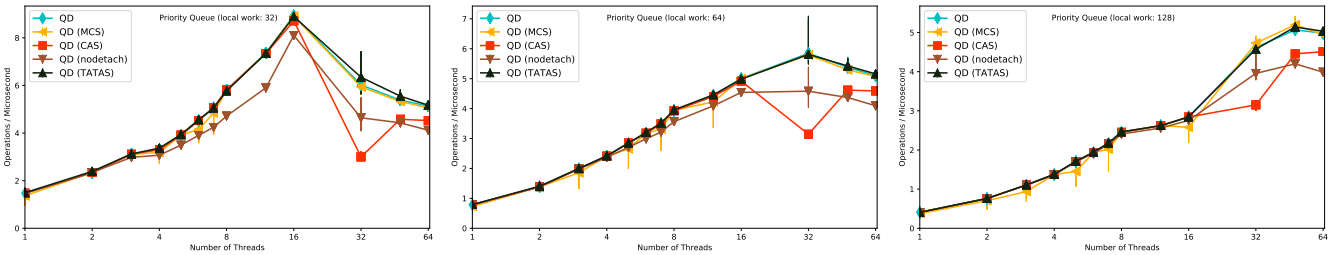

Fig. 16. Different variants of QD locks in the scenarios of Fig. 13.

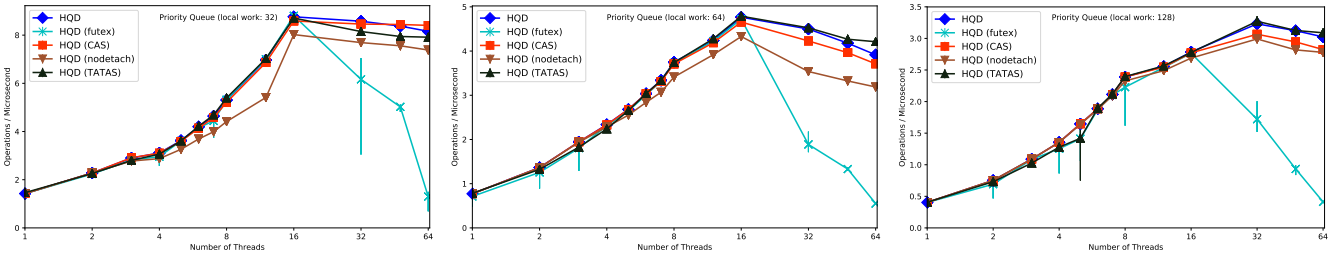

Fig. 17. Different variants of HQD locks in the scenarios of Fig. 14.

handover takes significantly longer, resulting in overall worse performance. For the lines identified as QD (CAS based) and HQD (CAS based), the `fetch_and_add` call in the `enqueue` function (line 10 of Fig. 6) is simulated with a CAS loop. We did this to find out how QD locking would perform in processors without a *fetch_and_add (FAA)* instruction. It is clear that *FAA* is beneficial for QD locking's implementation. It is also clear that even without using the *FAA* instruction, QD and HQD perform similar or better than the other algorithms we compare against. The implementations labeled QD (nodetach) and HQD (nodetach) have `insert` calls that wait for an acknowledgment from the delegated operation, even when the return value is not used. From these lines, it is clear that a large part of QD locking's performance advantage in this benchmark comes from the ability to do detached execution. However, QD locking still performs slightly better than other algorithms even without detaching critical sections. Finally, QD (TATAS) and HQD (TATAS) refer to implementations that use a simple TATAS lock internally and do not set a limit for retries to avoid starvation. This leads to at most a minuscule advantage in performance, which shows that the cost of avoiding starvation is not severe.

Further experiments with combinations of the above variants as well as with padding delegation queue entries to entire cachelines have not shown much deviation from the results shown here and are thus omitted from the figures.

7.2 Readers-Writer Benchmark

To evaluate our multi-reader QD lock implementations and compare them to other readers-writer locks we use a benchmark especially designed for RW locks. The benchmark is implemented from the description of RWBench that has been presented by Calciu *et al.* [4]. RWBench is similar to our data structure benchmark in that it measures throughput: the number of critical sections that N threads, which alternate between critical section work and thread-local work, can execute during t seconds. A shared array A with 64 integer entries is used for the protected shared memory. According to a specified probability for reading, it is determined randomly whether the critical section is a read or a write operation.

The read critical section work is placed inside a loop that iterates for four times. Inside this loop, the values of two random array slots from the shared array A are loaded.

The loop iteration count is also 4 for the write critical section. In the loop body, two of the 64 entries of A are updated in the following way: The two entries are randomly selected and an additional random integer I is generated.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

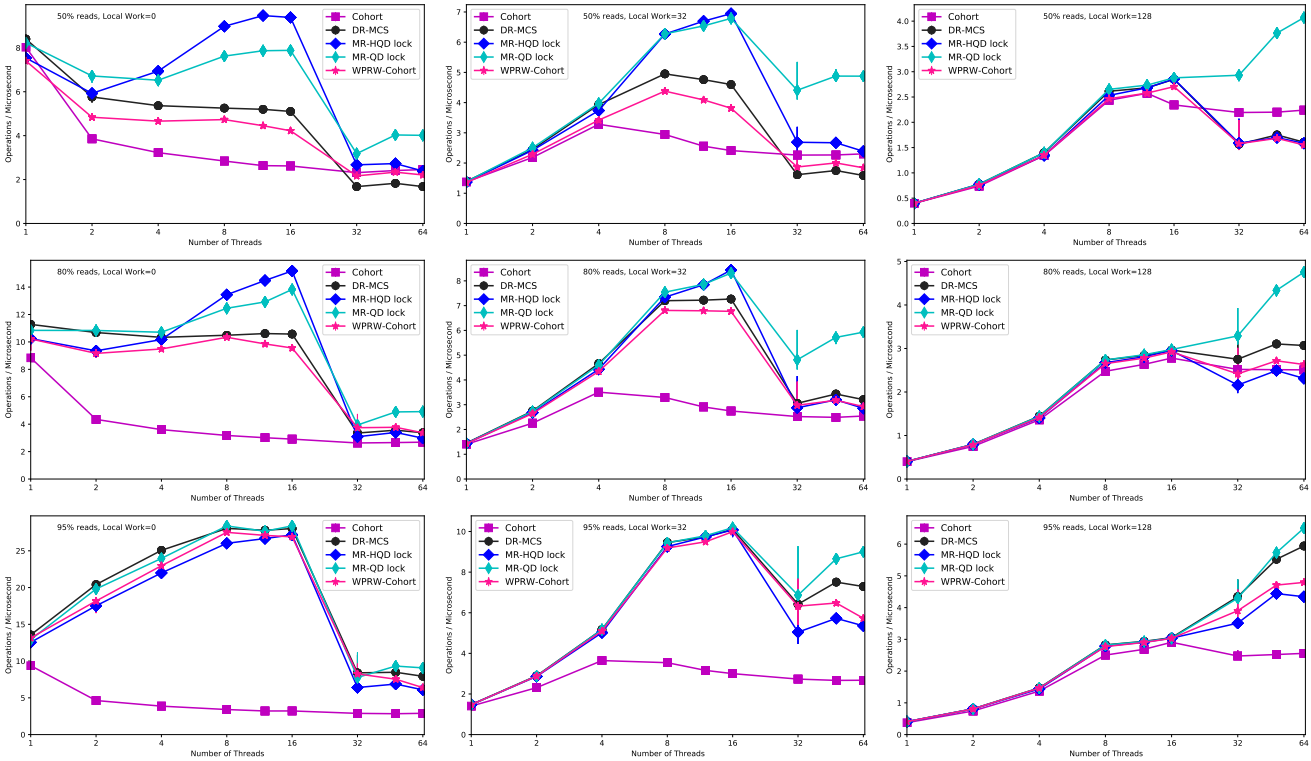

Fig. 18. Results for a readers-writer benchmark varying amount of readers (vertical) and thread-local work (horizontal).

Then I is added to the value stored in the first entry and subtracted from the value of the second entry. Thus, the sum of all array elements should be zero after the benchmark completes and can be used as a sanity check.

The thread-local work uses the same loop as the write operation, but writes in a thread-local array instead. One iteration in this loop is defined as one unit of thread-local work. This thread-local work is also used in the data structure benchmark (Section 7.1).

We compare our multi-reader QD lock (MR-QD) and its hierarchical variant (MR-HQD) with the DR-MCS and WPRW-Cohort algorithms of Calciu *et al.* [4]. All locks are constructed using the same algorithm; see Section 6. DR-MCS is a readers-writer variant of the MCS queue lock and WPRW-Cohort is based on a Cohort lock. For comparison we also show the performance of a mutual exclusion Cohort lock. The benchmark was run with different combinations of read probability and thread-local work. Figure 18 shows the results for 50%, 80% and 95% reads combined with 0, 32 and 128 thread-local work loop iterations.

The left column shows the somewhat unrealistic scenario of no thread-local work. Under such high contention, all algorithms perform best when operating on a single chip, but only the QD locking algorithms scale when there are many write operations. In the right column, with a high amount of thread-local work, it can be seen that with 50% and 80% readers only MR-QD continues to scale when running on multiple chips. Overall, it can be seen that MR-QD and MR-HQD outperform the other algorithms on a single processor chip when there is high contention or many write sections. Furthermore, MR-QD outperforms all other algorithms when running on multiple chips. MR-HQD, on the other hand, does not work as well on multiple chips. We reason that this is because the contention on the delegation queue drops too low, and therefore the lock is released frequently.

With only 50% read operations and 64 threads the mutual exclusion Cohort lock performs better than DR-MCS and comparably to WPRW-Cohort. This shows that our algorithms, which perform better when enough contention is maintained, can be used efficiently in scenarios with many writers. Established readers-writer locks have been limited to applications with very high amounts of readers to amortize the additional cost over mutual exclusion locks. Fewer readers are required for multi-reader QD locks to amortize their cost and be useful in applications traditionally not considered for readers-writer locking. But even with high amounts of readers, MR-QD consistently outperforms the other algorithms. In our experiments this is true even for 99% readers, albeit the difference becomes less pronounced. With only readers all four algorithms use only the read indicator, and therefore behave identically.

7.3 Kyoto Cabinet Benchmark

We also tested our multi-reader QD locks on the `kccachetest` program from the Kyoto Cabinet (version 1.2.76, compiled with `-O2`) to evaluate the feasibility of using it in existing software and how well it performs compared to other algorithms. The `kccachetest` uses `CacheDB`, an in-memory database designed for use as a cache. In particular, we run $100{,}000$ iterations of the `wicked` workload, which uses a user-defined amount of workers to perform operations on a `CacheDB`. As the workload is changed depending on the number of threads, it is not easily possible to compare performance with different numbers of workers. However, a comparison of different algorithms running with the same number of workers is possible.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2767046, IEEE Transactions on Parallel and Distributed Systems

TABLE 1
Times (secs) to run `kccachetest wicked -th $threads 100000`.

threads	1	2	4	8	12	16	24	32	48	64
Pthreads RW	.06	.13	.31	.65	1.12	1.45	4.66	6.78	13.99	21.64
DR-MCS	.06	.10	.18	.38	.69	1.05	3.14	5.17	11.33	17.43
WPRW-Cohort	.06	.11	.23	.53	.96	1.47	3.15	5.22	10.14	15.57
MR-QD	.06	.10	.18	.39	.73	1.08	3.18	4.67	9.46	13.86
MR-HQD	.06	.10	.19	.40	.71	1.06	3.21	4.65	9.43	14.50
MR-QD (p)	.06	.09	.18	.38	.68	1.03	2.92	4.07	7.74	12.31
MR-HQD (p)	.06	.09	.18	.39	.69	1.00	2.87	3.95	8.89	12.83

`CacheDB` uses Pthreads RW locks to protect its data, which can be replaced by other readers-writer locks. But since multi-reader QD locks do not strictly conform to the interface of readers-writer locks, some porting was required. We changed the code manually, but note that there exist techniques to perform these kinds of transformations automatically [25]. The porting was done with some glue code to make the multi-reader QD lock usable from the existing C++ code. This additional layer needs to store the parameters used by the critical sections in an accessible format. We store them in an `std::tuple`, which is then the pointer-sized parameter to the delegated operation. For return values we use a structure that also has a flag which signals when the value has been written to it. With these tools ready, the porting itself was straightforward. For using a thread-local variable signaling errors, a pointer to it had to be included in the parameter tuple, so that the error code arrives at the correct thread.

The results in Table 1 show runtime in seconds for varying numbers of worker threads. The row labeled Pthreads RW shows the performance of the original code of Kyoto Cabinet. The `kccachetest` is a kind of worst case scenario for our algorithms. It is designed to act as a benchmark but also to test the database. Therefore it always checks return values immediately to verify correctness. Besides that, outside the critical sections it only generates random numbers to decide which database operation to perform next. To make better use of delegation, we also patched the benchmark itself. For the two rows marked with (p) some error-checking has been postponed until at least 64 return values can be checked in bulk. This patch did not affect performance of the non-QD locking algorithms. Even without this patch, the results show that MR-QD and MR-HQD perform slightly better than other readers-writer locks. In contrast to Section 7.2, here MR-QD performs only slightly better than MR-HQD. This benchmark benefits less from detached execution as return values still need to be read for error checking and memory needs to be transferred to read the return value.

All in all, this shows that QD locks can be used in real applications for immediate benefit. Even better results are achieved when utilizing the time between a delegation and the use of return values.

All benchmark programs are available at http://www.it.uu.se/research/group/languages/software/qd_lock_lib.

7.4 Experience from Two Use Cases

We have also employed QD locking in two bigger systems. The first of them is in the implementation of the Erlang Term Storage (ETS). ETS is Erlang's in-memory key-value store and is the only shared memory between Erlang processes. Being shared memory, ETS has become a scalability concern on multicore machines [20]. As ETS tables are protected by locks, QD locks were used to improve performance. Using our C QD locking library to gradually transform the code [21], we first used MR-QD locks without detaching execution (MRQD-wait). Then, we passed all required parameters in an allocated struct to the algorithm so detaching execution was possible (MRQD-malloc). Finally, we wrote a version that passes the parameters directly into the MR-QD lock instead of allocating an object on the heap (MRQD-copy). It was shown [21] that all three versions significantly outperform the existing ETS implementation in contended scenarios. While MRQD-copy performed best under most circumstances, the cost of copying parameters made MRQD-malloc more efficient for large parameter sizes (above 150 bytes per critical section).

The second use case for QD locking is in the Argo Distributed Shared Memory System [19]. ArgoDSM provides a shared memory layer for running applications on multiple cluster computer nodes. There, traditional locking does not perform due to the enormous communication cost even for spin-waiting on values. Thus HQD locks were chosen as a provider for mutual exclusion, as they allow the single-node performance to be available in such systems.

For more information on these use cases, we refer the readers to these two publications [19], [21].

8 Concluding Remarks

We have presented the details of a novel synchronization mechanism called queue delegation locking and variations to support multiple readers as well as NUMA systems. Our experiments show that QD locking can outperform current state-of-the-art delegation algorithms such as flat combining, CC-Synch and H-Synch. A key advantage of QD locking is its ability to delegate operations without waiting for a response, its simplicity and its small communication cost. Our results also suggest that multi-reader QD locks can be a better performing alternative to readers-writer locks for some use cases, especially with workloads that frequently require a full exclusive lock but still can exploit some read-only parallelism. It remains as future work to look into is how QD locking can be used for data structures with fine-grained locking such as hash tables. Finally, an important practical issue to investigate is how tools can help programmers in migrating from traditional synchronization mechanisms and get the highest benefit possible from queue delegation locking.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[2] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[3] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical report, MIT CSAIL, 2014.

[4] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–166, New York, NY, USA, 2013. ACM.

[5] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–226, New York, NY, USA, 2015. ACM.

[6] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[7] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical report, Dept. of Computer Science and Engineering, University of Washington, Seattle, 1993.

[8] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 65–74, New York, NY, USA, 2011. ACM.

[9] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 247–256, New York, NY, USA, 2012. ACM.

[10] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 284–293, New York, NY, USA, 2010. ACM.

[11] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965.

[12] U. Drepper. Futexes are tricky, 2004.

[13] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable NonZero indicators. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, New York, NY, USA, 2007. ACM.

[14] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266, New York, NY, USA, 2012. ACM.

[15] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Summit*, pages 479–495, 2002.

[16] The GNU C library (glibc), low level lock implementation. Code available at https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/x86_64/lowlevellock.h;hb=beb0f59498c3e0337df298f9d7a3f8f77eb39842.

[17] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, New York, NY, USA, 2010. ACM.

[18] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[19] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 3–14, New York, NY, USA, 2015. ACM.

[20] D. Klaftenegger, K. Sagonas, and K. Winblad. On the scalability of the Erlang term storage. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, pages 15–26, New York, NY, USA, 2013. ACM.

[21] D. Klaftenegger, K. Sagonas, and K. Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 572–583. Springer, 2014.

[22] A. Kogan and M. Herlihy. The future(s) of shared data structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 30–39, New York, NY, USA, 2014. ACM.

[23] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the 21st annual Symposium on Parallelism in Algorithms and Architectures*, pages 101–110, New York, NY, USA, 2009. ACM.

[24] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*, volume 8304 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2013.

[25] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and portable locking for multicore architectures. *ACM Transactions on Computer Systems*, 33(4):13:1–13:62, Jan. 2016.

[26] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *Proceedings of the 12th International Conference on Parallel Processing*, pages 801–810, Berlin, Heidelberg, 2006. Springer.

[27] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994. IEEE Computer Society.

[28] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[29] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[30] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, New York, NY, USA, 2013. ACM.

[31] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 182–204. World Scientific, 1999.

[32] D. Petrović, T. Ropars, and A. Schiper. Leveraging hardware message passing for efficient thread synchronization. *ACM Transactions on Parallel Computing*, 2(4):24:1–24:26, Jan. 2016.

[33] Z. Radović and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages 241–252. IEEE Computer Society, 2003.

[34] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar. Design, verification and applications of a new read-write lock algorithm. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 48–57, New York, NY, USA, 2012. ACM.

[35] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, New York, NY, USA, 2009. ACM.

David Klaftenegger received his M.Sc. in Computer Science from the University of Edinburgh in 2011. He is currently a Ph.D. student at Uppsala University supported by UPMARC (Uppsala Programming for Multicore Architectures Research Center). His research focuses on synchronization algorithms and their application to multicore, NUMA, and cluster-scale shared memory systems.

Konstantinos Sagonas received his Ph.D. in Computer Science from Stony Brook University, U.S.A. in 1996. He is a faculty member at the School of Electrical and Computer Engineering of NTUA, Greece and at the Dept. of Information Technology of Uppsala University, Sweden. At Uppsala, he has been the director of ASTEC and ProFuN centers, and he is one of the main PIs of UPMARC, the center which funded this work. His research interests include programming languages and systems, concurrency and distribution, and techniques and tools for the effective analysis and testing of programs.

Kjell Winblad is currently pursuing his Ph.D. degree at Uppsala University, from which he also got his computer science M.Sc. degree in 2011. Besides locking algorithms, his research focuses on design and implementation of concurrent data structures. His research has been funded in part from the EU project RELEASE (IST-2011-287510) and from UPMARC.