Queue Delegation Locking

David Klaftenegger Konstantinos Sagonas Kjell Winblad
Department of Information Technology, Uppsala University, Sweden

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 use of waiting time. In this paper, we introduce *queue delegation locking*, a family of locks that provides high throughput by allowing threads to efficiently delegate their work to the thread currently holding the lock. Additionally, it allows threads that do not need a result from their critical section to continue executing immediately after delegating the critical section, with the possibility to delegate even more work. 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 back results. Experiments on a shared priority queue show that queue delegation outperforms leading synchronization algorithms by up to 115% when only half of the critical sections can be fully detached. Also, queue delegation locking gives a performance advantage of up to 202% when mixing parallel reads with delegated write operations. Last but not least, queue delegation offers better scalability than existing algorithms because it enables threads to continue executing instead of waiting for the execution of critical sections. Thanks to its simple building blocks, even its uncontended overhead is low, making queue delegation locking useful in a wide variety of applications.

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 [13] and CLH [2, 12], 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 [16], the hierarchical CLH lock [11] or more recently the *Cohort lock* [4]. 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 paper we focus on a different approach, which sends operations to the lock data structure instead of transferring the lock. This way, a single thread can execute operations sequentially without transferring memory between NUMA nodes or between a multicore's caches. This locking approach is called *delegation*, and the thread performing other threads' critical sections is called *helper*. There are two reasons for existing delegation algorithms to perform well. *Detached execution* allows threads to continue execution before the delegated critical section has been executed, but in its original form [15] the algorithm has some overhead and severe starvation issues for the helper thread. Therefore newer approaches, like flat combining [8] or remote core locking [10], require the threads to wait until their delegated sections are performed. By making the delegation itself faster they aim to further reduce the communication overhead. In comparison 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 can immediately continue their execution, possibly delegating more operations.

Delegated operations are placed in a *delegation queue*. As the queue preserves FIFO order, the correct order of operations is ensured. 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, and ensures that no operations are accepted when the lock is about to be released. If delegation fails the 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 no further enqueue call succeeds before the lock is released.

To communicate return values from a delegated operation to the thread that delegated it, the thread may have to wait for these values to be made available at designated memory addresses.

All requirements for queue delegation locking are met by assembling two simple components. These are a *mutual exclusion lock* to determine which thread is executing operations, and a *queue* to delegate operations to the lock holder. Using a *reader indicator* as a third component allows parallel access to multiple readers efficiently.

Contributions Our main contribution is 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, and show how to protect data structures with QD locks instead of mutexes. We also 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 algorithms.

Overview The rest of the paper is structured as follows. The next three sections present the different QD locking variants (Section 2), their implementation (Section 3), and their properties (Section 4). Subsequently, QD locking is compared with related synchronization mechanisms both on the algorithmic (Section 5) and the experimental level (Section 6). We end with some concluding remarks.

2. Queue Delegation Locks

This section describes the different queue delegation (QD) locking variants. We start with the necessary components, and then use them to assemble a basic QD lock. We then extend it to a multi-reader QD lock variant that allows multiple read-only operations to execute in parallel. Last, we sketch how QD locking can also provide the functionality of traditional mutual exclusion locks.

2.1 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. Its minimal interface consists of only two functions. The first is `try_lock`, which takes the lock if it is free and returns whether the lock has been taken. The second is `unlock`, which releases the lock. Other common locking functionality (e.g., taking the lock unconditionally) is not required but may be useful in certain situations; e.g., when translating legacy code to use QD locking. We also include an `is_locked` function in the interface of the mutual exclusion lock because we will use it for the 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 [14]. 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 straightforward: `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.

2.2 Queue Delegation Lock

We use the building blocks outlined above to assemble a QD lock as follows: The mutual exclusion lock determines in which way operations are accepted by the QD lock. When the mutual exclusion lock is free, it is taken, the delegation queue is opened, the operation is executed, the queue is flushed and finally the mutual exclusion lock is unlocked. However, when the mutual exclusion lock is already taken, the delegation queue is used to accept additional operations. 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.

The QD lock interface only consists of a `delegate` function which takes an operation as an argument. It is guaranteed that the operation will be executed before any operations from subsequent calls to `delegate` are executed. The operation 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 returning values from operations, the QD lock uses the semantics of *futures*. Namely, the value is not returned immediately, but the operation can promise to provide the value at a specific location upon its execution. When the calling thread needs to read the return value, it has to wait until it is available. This can either be exposed to the application programmer, or hidden by using a wrapper that immediately waits for the return value and returns the result.

```
1  bool try_lock(TATASLock* lock) {
2    if(lock->locked) return false;
3    return !test_and_set(&lock->locked);
4  }
5  void unlock(TATASLock* lock) {
6    lock->locked = false;
7  }
8  bool is_locked(TATASLock* lock) {
9    return lock->locked;
10 }
```

Pseudocode 1: The test-and-test-and-set (TATAS) lock

2.3 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 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 need is a `query` function [5] 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.

Using a reader indicator it is simple to extend the QD lock to a multi-reader QD lock which allows many concurrent readers. Its interface contains the additional functions `rlock` and `runlock` that work as in traditional readers-writer locks.

2.4 Queue Delegation Locks with a Wider Interface

We can easily extend the interface of QD and multi-reader 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 another order 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.

3. 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 paper. In fact, the components we use can be replaced by others 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, already provide good performance and scalability.

3.1 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. Thus, for our implementation, we chose one of the simplest locking algorithms available, the *test-and-test-and-set (TATAS)* lock, whose implementation is shown in Pseudocode 1. When the lock interface needs to be extended to also provide exclusive access (Section 2.4), the lock needs to also provide a `lock` function and, for performance, the underlying lock should be chosen more carefully. A more careful choice may also be required in order to provide stronger guarantees

in the resulting algorithm. For example, starvation freedom or (limited) fairness can only be achieved if the underlying lock provides such guarantees. More on this in Section 4.

3.2 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 additional operations.

Our delegation queue implementation, shown in Pseudocode 2, 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 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[1] (line 7), which gives each delegated operation 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 20). Note that the `closed` flag is used in lines 6, 22 and 27 only to reduce write contention on the counter and is not needed for correctness.[2] The `flush` function repeatedly reads the counter and dequeues operations until the queue has been put to closed state by an `enqueue` function (line 24 detects this) or because the counter has not been updated since the last check (line 19).

Special care is needed when writing and reading the delegated operation; i.e., in lines 9 and 30. 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 (`fun_ptr`) and two pointer-sized parameters for the operation. To ensure that no partially-written operations are used, we decided that the function pointer is to be written last and read first. Setting the function pointer to the null pointer (line 32), immediately after the delegated operation has been executed, allows to wait in the next iteration of the `for` loop for it to be set to a valid value. As the function pointer is written last, the entire operation can be read safely when the function pointer has a non-null value.

3.3 Queue Delegation Lock Implementation

With the mutual exclusion lock and the tantrum queue available, only the actual `delegate` function has to be provided to build a QD lock. As can be seen in Pseudocode 3 this function alternates between trying to acquire the lock and trying to delegate the operation until one of them succeeds. 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.

3.4 Multi-Reader Queue Delegation Lock Implementation

As described in Section 2.3, we need a reader indicator to extend QD locking to allow multiple parallel read operations. We use a simple reader indicator algorithm, shown in Pseudocode 4. 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

```
1  void open(DelegationQueue* q) {
2    q->counter = 0;
3    q->closed = false;
4  }
5  bool enqueue(DelegationQueue* q, Operation op) {
6    if(q->closed) return CLOSED;
7    int index = fetch_and_add(&q->counter, 1);
8    if(index < ARRAY_SIZE) {
9      q->array[index] = op; /* atomic */
10     return SUCCESS;
11   } else return CLOSED;
12 }
13 void flush(DelegationQueue* q) {
14   int todo = 0;
15   bool open = true;
16   while(open) {
17     int done = todo;
18     todo = q->counter;
19     if(todo == done) { /* close queue */
20       todo = swap(&q->counter, ARRAY_SIZE);
21       open = false;
22       q->closed = true;
23     }
24     if(todo >= ARRAY_SIZE) { /* queue closed */
25       todo = ARRAY_SIZE;
26       open = false;
27       q->closed = true;
28     }
29     for(int index = done; index < todo; index++) {
30       while(q->array[index].fun_ptr == NULL); /* spin */
31       execute(q->array[index]);
32       q->array[index].fun_ptr = NULL; /* reset */
33     }
34   }
35 }
```

Pseudocode 2: The delegation queue implementation

```
1  void delegate(QDLock* l, Operation op) {
2    while(true) {
3      if(try_lock(&l->lock)) {
4        open(&l->queue);
5        execute(op);
6        flush(&l->queue);
7        unlock(&l->lock);
8        return;
9      } else if(enqueue(&l->queue, op)) return;
10     yield();
11   }
12 }
```

Pseudocode 3: The `delegate` function

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.* [1]. The third available algorithm, SNZI [5], was not used mainly due to its complexity. For sufficiently large systems it may be a better choice.

Pseudocode 5 shows the implementation of multi-reader QD locks. It is derived from the *writer-preference readers-writer lock* algorithm [1]. In fact, functions `rlock` and `runlock` are unchanged; we include them just for completeness and refer the reader to that paper [1] for their explanation. The queue delegation algorithm itself requires only two changes to allow multiple parallel readers. The

[1] As not all platforms have a *fetch_and_add* instruction, we have also done experiments where *fetch_and_add* is simulated with a *CAS* loop; see Sect. 6.

[2] Overflowing is practically not a problem if one uses 64 bits for the counter.

```
1  bool query(ReaderIndicator* indicator) {
2    for(counter : indicator->counters)
3      if(counter > 0) return true;
4    return false;
5  }
6  void arrive(ReaderIndicator* indicator) {
7    indicator->counters[ThreadID] += 1; /* atomic */
8  }
9  void depart(ReaderIndicator* indicator) {
10   indicator->counters[ThreadID] -= 1; /* atomic */
11 }
```

Pseudocode 4: The reader indicator

first is on line 2 where execution can be blocked on a write barrier to avoid starvation of readers. The second change is on line 6 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 5 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, note that this is not a problem because the delegated operations can never be executed while there are active readers; the while loop in line 6 ensures this. 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.

4. Properties

Having described the basis for QD locking's implementation, we now discuss some of its properties; most notably starvation freedom and linearizability. We also show how to extend QD locking to be better suited for Non-Uniform Memory Access (NUMA) systems.

4.1 Starvation Freedom

The `delegate` function implementations shown in Pseudocodes 3 and 5 are not starvation free, meaning that a thread can get starved while executing these functions and never succeed with a `try_lock` or `enqueue` call. 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, so we have decided, for presentation purposes, to only show simple versions of these functions.

However, we can easily make the algorithms starvation free by putting a limit on how many times their `while(true)` loop is executed. If the limit is reached, the mutual exclusion lock can be acquired unconditionally to execute the operation. If the mutual exclusion lock is starvation free, it is easy to see that the whole `delegation` function must be starvation free because, in the worst case, it only does a fixed amount of work before it acquires the starvation free mutual exclusion lock unconditionally.

For multi-reader QD locks, there is the additional problem of readers starving delegated operations or vice versa. Our algorithm uses write preference which readers can overturn to avoid starvation, thus multi-reader QD locks are starvation free whenever the QD lock component is starvation free.

4.2 Linearizability

Linearizability [9] 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 delegation 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.

```
1  void delegate(MRQDLock* l, Operation op) {
2    while(l->writeBarrier > 0) yield();
3    while(true) {
4      if(try_lock(&l->lock)) {
5        open(&l->queue);
6        while(query(&l->indicator)) yield();
7        execute(op);
8        flush(&l->queue);
9        unlock(&l->lock);
10       return;
11     } else if(enqueue(&l->queue, op)) return;
12     yield();
13   }
14 }
15 void rlock(MRQDLock* l) {
16   bool bRaised = false;
17   int readPatience = 0;
18 start:
19   arrive(&l->indicator);
20   if(is_locked(&l->lock)) {
21     depart(&l->indicator);
22     while(is_locked(&l->lock)) {
23       yield();
24       if((readPatience == READ_PATIENCE_LIMIT) && !bRaised) {
25         fetch_and_add(&lock->writeBarrier, 1);
26         bRaised = true;
27       }
28       readPatience += 1;
29     }
30     goto start;
31   }
32   if(bRaised) fetch_and_sub(&lock->writeBarrier, 1);
33 }
34 void runlock(MRQDLock* l) {
35   depart(&l->indicator);
36 }
```

Pseudocode 5: The code for multi-reader QD locks

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 3 and 4 in Pseudocode 3). This is true, because `try_lock` can only succeed when the lock is free, which implies 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. However, when queue delegation is used with operations that have side effects outside the date 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.

```
1  void delegate(HQDLock* h, Operation op) {
2    QDLock* l = h->localLocks[my_node];
3    while(true) {
4      if(try_lock(&l->lock)) {
5        lock(&h->globalLock);
6        open(&l->queue);
7        execute(op);
8        flush(&l->queue);
9        unlock(&h->globalLock);
10       unlock(&l->lock);
11       return;
12     } else if(enqueue(&l->queue, op)) return;
13     yield();
14   }
15 }
```

Pseudocode 6: The `delegate` function in the HQD lock

4.3 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 [4] are constructed from traditional mutual exclusion locks and to how the H-Synch algorithm is constructed from CC-Synch [6]. An HQD lock uses one mutual exclusion lock and one delegation queue per NUMA node. Additionally, the lock contains a *global* mutual exclusion lock which is used to determine which NUMA node is allowed to execute operations. We have chosen a CLH lock [2, 12] as the global lock, which is a fair queue-based lock. This guarantees that all NUMA nodes will be able to execute operations in a reasonably fair order.

The implementation for the HQD lock algorithm is shown in Pseudocode 6. As with other hierarchical locking approaches each thread needs to know which NUMA node it is running on. 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. 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; see Section 6. 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 on NUMA systems.

5. Related Work

Detached Execution Oyama *et al.* developed the first algorithm that used delegation to deal with synchronization bottlenecks [15]. Similar to our QD locking, their algorithm allows for operations to be delegated without any need to wait for their actual execution. Instead, it stores delegated operations in a linked list based LIFO queue. A thread that successfully executes a CAS instruction on a lock word that also functions as head of the LIFO queue becomes the *helper*. The helper continues executing requests that are put by other threads as long as the LIFO queue is not empty. Therefore, starvation is possible in the algorithm of Oyama *et al.*: the helper can starve while executing requests for other threads. This is not the case for QD locking which uses a tantrum queue with limited size. In the algorithm of Oyama *et al.*, threads delegate operations by performing a CAS operation on the pointer to the LIFO queue. As noted by others [6, 8], this pointer can become a contended hot spot which limits scalability. Although the queue in our QD locking algorithm can potentially have a contended hot spot as well, as we will show in Section 6, it still performs well because we use a fetch-and-add instruction instead of a CAS loop to synchronize between threads. A final difference is that in order to make the lock of Oyama *et al.* linearizable, the LIFO queue needs to be reversed, which also imposes extra cost compared to QD locking.

Flat Combining Another approach to coalesce operations on a shared data structure into a single thread is *flat combining* (FC) [8]. 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 *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 algorithm of Oyama *et al.* for contended workloads [6, 8], but these comparisons do not consider detached execution and instead always immediately wait for critical sections. Compared to flat combining, our approach is offloading work to the lock's holder and we do not need to wait for the critical section to be executed. Also unlike FC, the "combiner" 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 not very high.

NUMA-aware Locks Several NUMA aware locking algorithms that exploit the hierarchical structure of NUMA systems exist by now [3, 11, 16]. Most recently Dice *et al.* have proposed *lock cohorting* as a general mechanism to create NUMA-aware hierarchical locks [4]. A cohort lock has one lock for every NUMA node in the system and an additional global lock. A lock 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. This approach is much more efficient on NUMA machines under high contention than traditional locks. However, without additional delegation mechanisms, NUMA-aware traditional locks suffer from frequently transferring the data structure protected by the lock between the private caches in one processor chip.

Synch Algorithms CC-Synch, DSM-Synch, and H-Synch are queue-based delegation algorithms developed by Fatourou and Kallimanis [6]. 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 [2, 12], while the queue in DSM-Synch is based on the MCS lock [13]. 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 [6]. Compared to the Synch algorithms, 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, threads only need to wait if and when a return value is needed. The reason why waiting is required in CC-Synch, DSM-Synch and H-Synch is that there is no guarantee that the operation would be executed if a thread would continue without waiting for acknowledgment. Second, our implementations using a queue implemented with an array buffer are more efficient than the Synch based algorithms because they require less cache misses (external loads fed from other cores). 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.

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 [17]. 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* is a software locking scheme where one processor core is dedicated to execution of critical sections [10]. The dedicated core spins in a loop checking an array of request slots 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 slots even when they are empty.

6. Evaluation

Having compared queue delegation locking with other synchronization mechanisms on the algorithmic level, we now compare its performance experimentally. For this, we use two synthetic benchmark sets and a program from the Kyoto Cabinet, an application of considerable size and code complexity. 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 (FC), CC-Synch and H-Synch.[3] We will also compare against two variants of the algorithm by Oyama *et al.* Both variants are implemented after the pseudocode provided in the paper [15]. The first variant (called Oyama) uses standard malloc and free calls for allocation and deallocation of queue nodes. In the second variant (called OyamaOpt), we preallocate 4096 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 CLH and Cohort locks. The implementation of the CLH lock is taken from the Synch repository; the Cohort lock one is written by us.

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 off to another thread if less than `rep_threshold` operations have been performed. We also added a condition to FC that will hand off the lock to another thread if `help_limit` or more operations have been helped during the help session. The `help_limit` was set to 4096 in all experiments that we present graphs for in this paper. For FC, `num_rep` was set to 4096 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 (i.e., a total of 64 hardware threads running on 32 cores). The machine ran Debian Linux 3.10-0.bpo.2-amd64 and had 128GB of RAM. All lock implementations and the two synthetic benchmarks are written in C, while Kyoto Cabinet is written in C++. All code was compiled using GCC version 4.7.2 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.

Thread Pinning We pin threads to hardware 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 hardware threads on one NUMA node, then on two NUMA nodes, and so on. The pinning policy on a NUMA node level first pins a thread to a core without any previously pinned thread and then fills the hardware threads without any previously pinned thread. Measurements with up to eight threads will therefore run on eight cores on a single chip on our machine. Up to 16 threads will run on a single chip with all hardware threads occupied, up to 32 threads will run on two chips, and so on.

6.1 Data Structure Benchmark

Benchmark Description This benchmark 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 [6, 8]. 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* [7]. 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 [8]. Also, a priority queue is well-suited 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 for both. The seed for the random number generator is thread-local to avoid false sharing between threads. The thread-local work is the same as in the benchmark in Section 6.2, so we omit its description here for brevity.

Results Figure 1 shows the results of the benchmark for different thread counts and different amount of work between the operations.

[3] Available at: `http://github.com/mit-carbon/Flat-Combining` and `https://code.google.com/p/sim-universal-construction/`.

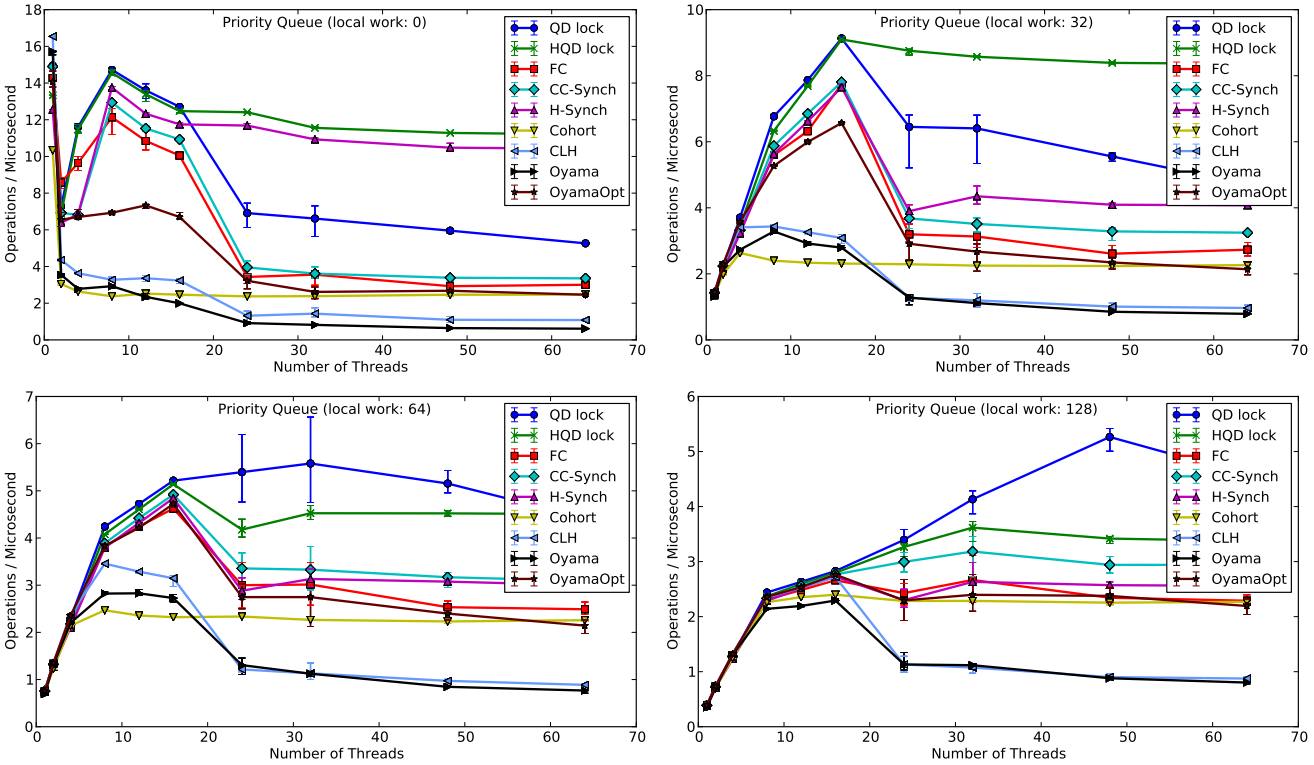

Figure 1. Throughput for a priority queue benchmark with 50% `insert` and 50% `extract_min` operations varying the amount of local work.

The graph at the top left corner shows the case with no local work. All algorithms have a big performance drop when going from the sequential case to two threads. For all delegation algorithms except Oyama and OyamaOpt 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. The Oyama implementation clearly suffers from the cost of memory management via system calls. OyamaOpt performs relatively bad compared to the other delegation algorithms in the case when there is no local work. This can be explained by the CAS loop that Oyama *et al.* use in their algorithm to delegate work and by the overhead of reversing the LIFO queue.

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 number of required 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 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, Cohort is not able to perform well compared to delegation algorithms on this workload.

The graph in the top right corner of Figure 1 shows results for 32 units of local work between the operations. In this scenario, 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 a `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 Figure 2. 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 Oyama(Opt) 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

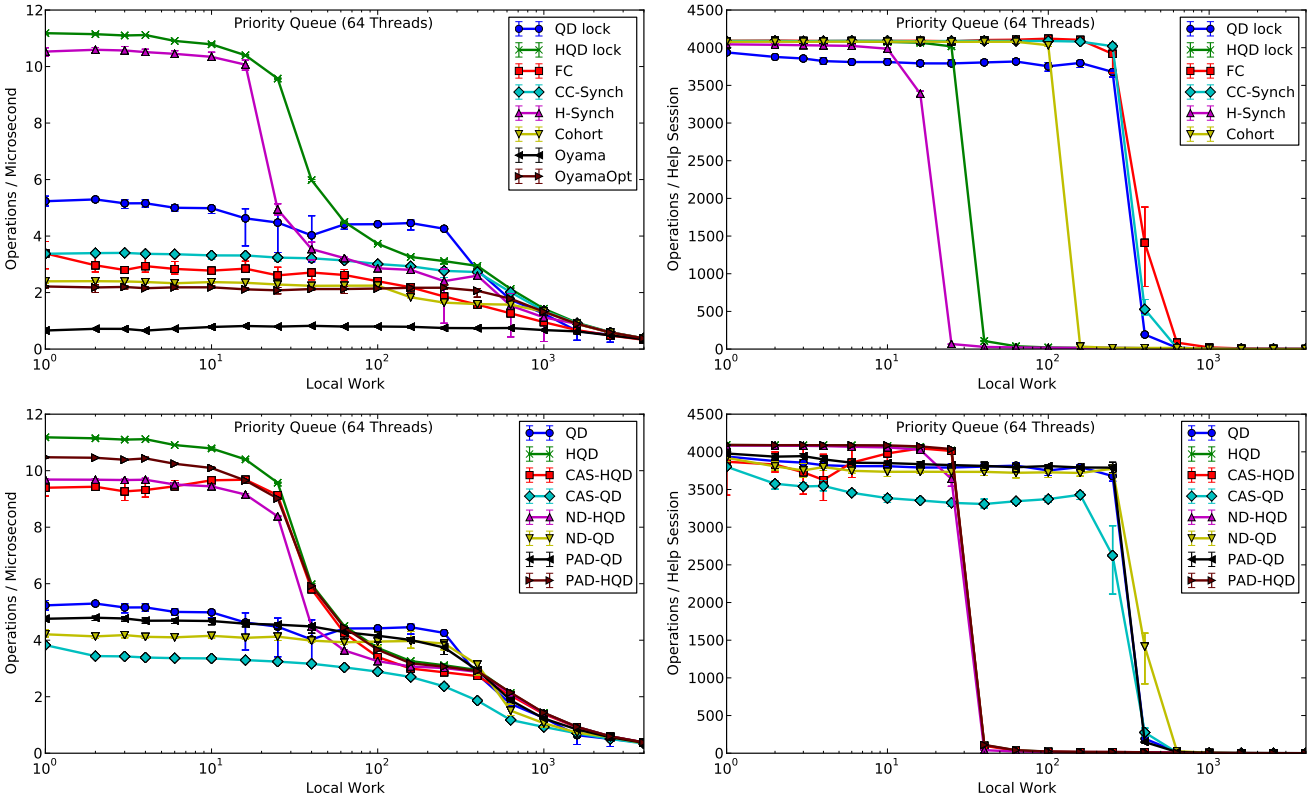

Figure 2. The two graphs on the left show operations per microsecond under different contention (amount of local work between operations). The two graphs on the right show the average number of helped operations for all help sessions. The graphs at the top compare QD locking with other algorithms. The graphs at the bottom try to investigate why QD locking performs well. Lines prefixed with PAD have the queue entries padded to a full cache line. Lines prefixed with ND wait for confirmation that the `inserts` have been done although this is unnecessary. Lines prefixed with CAS have the *fetch_and_add* used in the delegation queue implemented with a CAS loop instead of an instruction.

explained with the increased traffic between the nodes. The two graphs at the bottom part of Figure 1 can also be explained with a similar reasoning: More work between the operations is more beneficial for the QD, HQD and Oyama(Opt) algorithms compared to the other locking algorithms because of the ability to delegate `insert` operations without needing to wait for response. One reason why flat combining performs worse than other algorithms with low contention may be that few request nodes have time to get filled between traversals, resulting in wasted work.

A Deeper Look into QD Locking's Performance The two graphs at the bottom part of Figure 2 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. In the lines identified as CAS-QD and CAS-HQD, the `fetch_and_add` call in the `enqueue` function (line 7 of Pseudocode 2) is simulated with a CAS loop. We did this to find out how QD locking would perform in processors without a *fetch_and_add* instruction. It is clear that *fetch_and_add* is beneficial for QD locking's implementation. It is also clear that even without using a *fetch_and_add* instruction, QD and HQD perform similar or better than the other algorithms we compare against. The implementations labeled ND-QD and ND-HQD 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 delegate operations without waiting for response when not needed. Finally, the PAD-QD and PAD-HQD implementations, which have every entry in the delegation queue padded to one cache line, investigate how being able to read several operations per cache miss affects performance. By comparing PAD-HQD to HQD, it is easy to see the benefit of being able to read several operations with one cache miss when all threads that are writing to the array run on the same NUMA node. The difference between PAD-QD and QD is not that large. However, QD has higher variance than PAD-QD. The reason for this variance is not yet fully understood, but it is possibly caused by threads on different NUMA nodes that write to the same cache line when enqueueing operations.

6.2 Readers-Writer Benchmark

Benchmark Description To evaluate our multi-reader QD lock implementation and compare it to other RW 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.* [1]. RWBench is similar to our data structure benchmark in that it measures the 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.

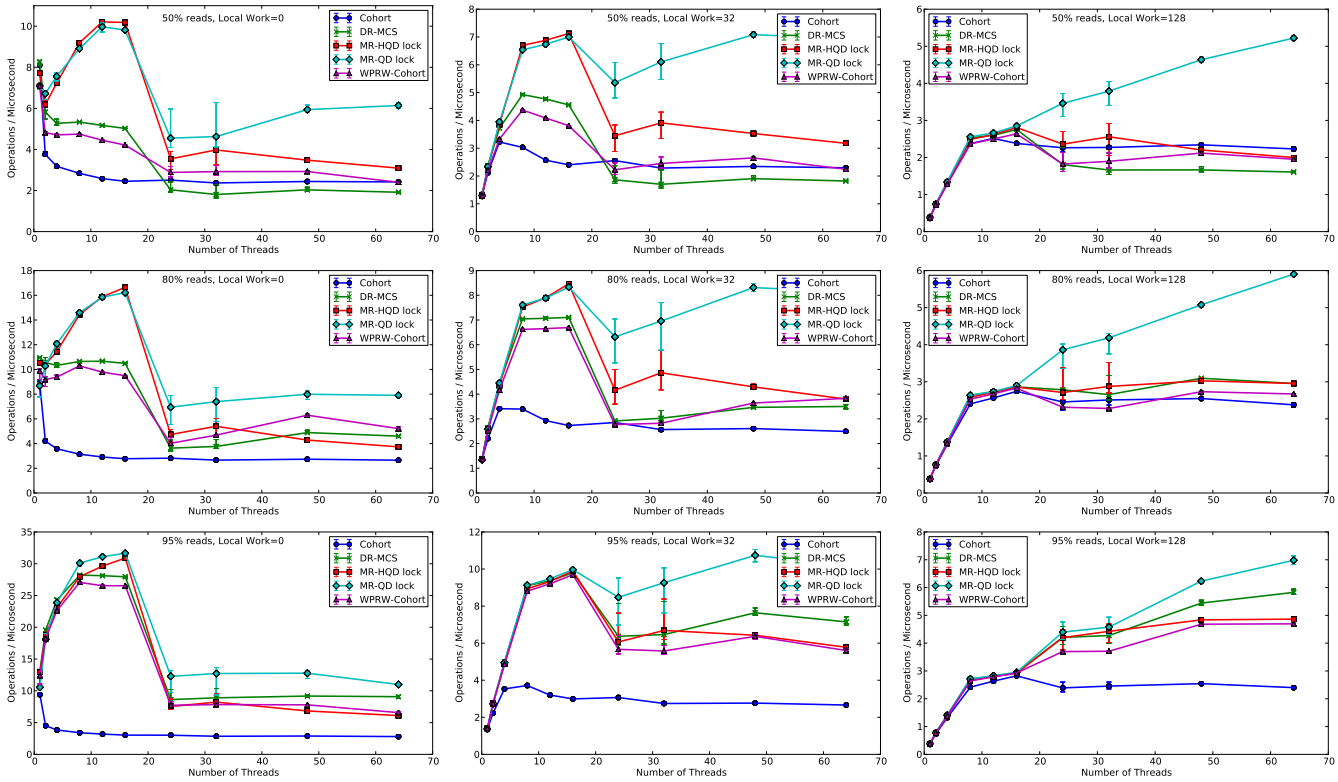

Figure 3. Results for a readers-writer benchmark varying amount of readers (vertical) and thread-local work (horizontal).

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. 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 6.1).

Results 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.* [1]. All locks are constructed using the same algorithm; see Section 3.4. 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 3 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, and 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 the lock therefore 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. Less readers are required for multi-reader QD locks to amortize their cost and be useful to 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.

6.3 Kyoto Cabinet Benchmark

We also tested our multi-reader QD locks on the `kccachetest` program from the Kyoto Cabinet (version 1.2.76) 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. Anyhow, a comparison of different algorithms running with the same number of workers is possible.

`CacheDB` uses Pthreads RW locks to protect its data, which can be replaced by other readers-writer locks. However, as multi-reader QD locks do not strictly conform to the interface of readers-writer

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

Table 1. Times to run `kccachetest wicked -th $threads 100000`

locks, some porting was required. We changed the code manually, but note that there exist techniques to perform these kinds of transformations automatically [10]. The porting was done with some glue code to make the multi-reader QD lock usable from C++. 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 signalling 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 results marked with (p) some error-checking has been delayed 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 current readers-writer locks. Differing from the results in Section 6.2, MR-QD performs only slightly better than MR-HQD. However, as results still need to be read for error checking, this benchmark benefits less from detached execution. While there is no immediate memory transfer for synchronization, there is one later 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.

7. Concluding Remarks

We have proposed a novel synchronization mechanism called queue delegation locking and variations to support multiple readers as well as NUMA. Our experiments show that QD locking can outperform current state-of-the-art synchronization algorithms such as Oyama *et al.*, flat combining, CC-Synch and H-Synch. A key advantage of QD locking is its ability to delegate operations without waiting for response, its simplicity and its small communication cost. Our results also suggest that multi-reader QD locks can be a more performant alternative to readers-writer locks for some use cases. It remains as future work to investigate how QD locking can be used together with combining. Another interesting area 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, Jan. 1986.

[8] 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.

[9] 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.

[10] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 65–76, Berkeley, CA, USA, 2012. USENIX Association.

[11] 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-Verlag.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.