Delegation Locking Libraries for Improved Performance of Multithreaded Programs*

David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad

Department of Information Technology, Uppsala University, Sweden

Abstract. While standard locking libraries are common and easy to use, delegation algorithms that offload work to a single thread can achieve better performance in multithreaded applications, but are hard to use without adequate library support. This paper presents an interface for delegation locks together with libraries for C and C++ that make it easy to use *queue delegation locking*, a versatile high-performance delegation algorithm. We show examples of using these libraries, discuss the porting effort needed to take full advantage of delegation locking in applications designed with standard locking in mind, and the improved performance that this achieves.

1 Introduction

In many programming languages, locking is still the dominant synchronization mechanism that multithreaded programs use to protect their critical sections. Especially in systems programming languages, lock-based multicore programming is supported by libraries which are readily available and easy to use. Recently, particularly in programs manipulating shared data structures protected by a single lock, researchers have investigated ways of offloading critical sections to a single processor core and letting them be executed by the same thread [3, 4, 7, 9–11]. Such *delegation algorithms*, which effectively bring the operations of critical sections to the data instead of bringing data to where the operations execute, have significant performance advantages on modern multicores and are often superior to implementing concurrent data structures either by fine-grained locking or in lock-free ways [3, 4]. Alas, this style of multicore programming is very hard to employ without adequate library support.

This paper presents an interface for delegation locks and two libraries for C and C++ that make a versatile high-performance delegation algorithm, called *queue delegation locking* (Sect. 2), easy to use. More specifically, we describe the most important aspects of the programming interfaces of our libraries (Sect. 3 and 4), and our experiences in porting code of significant size ($\approx$ 16 300 LOC) to using this library both in effort required as well as in performance improvements that this brought (Sect. 5). To the best of our knowledge, our libraries[1] are the first to provide portable support for delegation. By portable we mean that they only require the presence of a compiler that adheres to the C11/C++11 standards. In addition, we discuss issues that are involved when one wants to modify existing multithreaded programs that have been written with standard locking in mind to take full advantage of delegation locking (Sect. 7).

* Research supported in part by EU project RELEASE (IST-2011-287510) and by UPMARC.

[1] `http://www.it.uu.se/research/group/languages/software/qd_lock_lib` (Publicly available)

2 Queue Delegation Locking

The delegation algorithm we use, called *Queue Delegation (QD) locking* [6], is a new efficient delegation algorithm whose idea is simple. When, e.g., a shared data structure protected by a single 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 (called the *helper*). If the operation does not read from the shared structure, successful delegation allows a thread to immediately continue execution, possibly delegating more operations or doing other work. The helper thread is responsible for eventually executing delegated operations in the order they arrived to ensure linearizability. This implies that reading from the shared data structure needs to wait for all prior operations so that no outdated data is ever read. Most kinds of critical sections can be delegated with this scheme, and waiting is only needed when effects of a critical section need to be visible.

QD locking is realized by placing delegated operations in a *delegation queue*. The operations are stored in order of their arrival, and thus their 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 happens when the queue is already full or when the helper already executed all operations it found and is about to release the lock. 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 delegation queue is implemented as a fixed size buffer which stores operations together with arbitrarily sized parameters, e.g. some data to insert. This avoids the allocation of additional memory to pass data for each operation. An atomically incremented offset is used to access the buffer and automatically close the queue when it is full. If the required data exceeds the remaining space in the queue, the queue still closes, but the delegation has to be retried. A standard mutual exclusion (mutex) lock is used only to determine the helper thread to which operations are delegated.[2] Our libraries also come with a lock which allows multiple readers in parallel. We base our multiple readers QD (MRQD) lock on the mutex lock agnostic write-preference readers-writer lock algorithm of Calciu *et al.* [2]. As a *reader indicator* we use reader groups [5], which essentially is a counter that is distributed over many cache lines to avoid false cache invalidations.

The QD lock is accessed using a `delegate` function. When needed, e.g. for reading data, *futures* can be used to wait until the data is available. Futures can be as simple as having the delegated operation setting a flag upon completion or writing data to a predetermined location. MRQD locks come with an additional trade-off: For consistency, readers have to wait immediately for previously delegated operations before they can all access in parallel. By the time the read operation is performed it is guaranteed that any previously delegated operation on the same QD lock has finished. Read-only critical sections using the `rlock` and `runlock` functions work exactly as in traditional readers-writer locks. To ease porting of existing code we also provide `lock` and `unlock` on QD and MRQD locks, which are simply forwarded to the underlying mutex lock. For more information on the queue delegation algorithm, e.g. hierarchical NUMA variants or susceptibility to starvation, see a companion paper [6] describing QD locking in detail.

[2] Our libraries use a test-and-test-and-set lock for this; for the reasoning of this choice refer to [6].

3 C Library

We will present the C API of our QD locking library through examples. Let us first consider the implementation of a shared integer (ShInt) that can be accessed concurrently by several threads. All operations on ShInts are supposed to be atomic. A ShInt can be represented with the following C structure which employs an MRQD lock to coordinate concurrent accesses to it.

```c
typedef struct { MRQDLock lock; int value; } ShInt;
```

Now consider a `mult` operation that multiplies a ShInt with an integer value and stores the result back on the ShInt. Since MRQD locks support the traditional locking operations `lock` and `unlock`, the `mult` function could be implemented like this:

```c
void mult1(ShInt* v1, int v2) {
  LL_lock(&v1->lock); v1->value = v1->value * v2; LL_unlock(&v1->lock);
}
```

This implementation is easy to understand, but it may not be very efficient when a ShInt is contented on a multicore machine. One reason is that every thread has to wait for the thread currently holding the lock to release it before its execution can proceed. With thread preemption, the OS can force the lock holding thread to be suspended for an arbitrary amount of time. Since `mult` does not return any value, we could instead delegate the responsibility of executing the critical section to another thread that has already acquired the lock. This is easy to do using the `LL_delegate` function from the QD locking library:

```c
1 typedef struct { ShInt* v1; int v2; } MultMsg;
2 void mult_cs(unsigned int sz, void* msgP) {
3   MultMsg* msg = (MultMsg*)msgP;
4   msg->v1->value = msg->v1->value * msg->v2;
5 }
6 void mult2(ShInt* v1, int v2) {
7   MultMsg msg = {.v1 = v1, .v2 = v2};
8   LL_delegate(&v1->lock, mult_cs, sizeof(msg), &msg);
9 }
```

The `LL_delegate` call (line 8) either executes the `mult_cs` function in the current thread while holding the lock or delegates the responsibility to execute it to the current lock holder. As long as all accesses to the ShInt are done by threads that have acquired the lock, `mult1` and `mult2` are equivalent in the sense that it is impossible to detect which one is used. The parameters to `LL_delegate` are the lock, the function with the code of the critical section, the message size and a pointer to the message data. The message data and the message size will be passed to the delegated function when it is executed. The programmer cannot rely on that the `mult_cs` function is executed by the current thread or that the message data is not copied to another location. Many threads can delegate critical sections to the same helper, which has the performance advantage that the manipulated data can stay in the same private cache while it is being manipulated.

Extending the example, suppose we now also want the result of the multiplication as a return value. We can create the function `mult_res` for that purpose:

```
1  typedef struct { ShInt* v1; int v2; int* retval; } MultResMsg;
2  void mult_res_cs(unsigned int sz, void* msgP) {
3    MultResMsg* msg = (MultResMsg*)msgP;
4    msg->v1->value = msg->v1->value * msg->v2;
5    *msg->retval = msg->v1->value;
6  }
7  int mult_res(ShInt* v1, int v2) {
8    int res;
9    MultResMsg msg = {.v1 = v1, .v2 = v2, .retval = &res};
10   LL_delegate_wait(&v1->lock, mult_res_cs, sizeof(msg), &msg);
11   return res;
12 }
```

The delegated function writes back the result to the location pointed to by `retval` in `MultResMsg` (line 5). Here we are using a variant of `delegate` called `LL_delegate_wait` (line 10), which waits for the delegated function's execution before it returns. Unsurprisingly, `LL_delegate_wait` also guarantees that the lock is held by the thread that executes the delegated function. Instead of returning the actual value from `mult_res` one could return at this point a future that will contain the result when it is ready. However, in a low-level language such as C, constructing this future is something that needs to be done by the program itself. In the next section we will see how the situation changes in C++.

The QD locking library also supports read-only critical sections:

```
int read(ShInt* v) {
  int res;
  LL_rlock(&v->lock);   res = v->value;   LL_runlock(&v->lock);
  return res;
}
```

Using read-only critical sections is a powerful way to support multiple parallel read operations. Mixing read-only critical sections with delegated critical sections can give excellent performance. Since threads can delegate critical sections without waiting for the actual execution they can continue and issue a read-only critical section. Read-critical sections are thus more likely to bulk up so that more can execute in parallel than if a readers-writer lock without delegation was used.

We have almost gone through all functions in the locking library for C. The only ones left are the `LL_delegate_or_lock` family of functions. These provide the same functionality as `LL_delegate` but avoid the overhead that `LL_delegate` has in creating a separate buffer for copying the message for the delegated critical section to the current lock holder. Creating this buffer can be particularly expensive when the size of the buffer is not known at compile time and needs to be allocated dynamically. Dynamic memory allocation is expensive and can be a scalability problem on multicores. The code at the top of the next page shows how this family of functions is used. The `enqueue` function is for a concurrent queue. The function `enq` is not shown for lack of space but we can assume that it is a non-thread-safe enqueue function which copies the data into the queue data structure. The `LL_delegate_or_lock` function (line 5) attempts to get a message buffer of the specified size for a critical section. If `LL_delegate_or_lock` succeeds a buffer address will be returned, otherwise NULL is returned and the lock is acquired.

```
 1 void enqueue_cs(unsigned int sz, void* m) {
 2   enq(*((Queue**)m), sz - sizeof(Queue*), &((char*)m)[sizeof(Queue*)]);
 3 }
 4 void enqueue(Queue* q, int dSize, void* d) {
 5   void* buff = LL_delegate_or_lock(q->lock, dSize + sizeof(Queue*));
 6   if (buff == NULL) {
 7     enq(q, dSize, d);
 8     LL_delegate_unlock(lock);
 9   } else {
10     memcpy(buff, &q, sizeof(Queue*));
11     memcpy(&((char*)buff)[sizeof(Queue*)], d, dSize);
12     LL_close_delegate_buffer(q->lock, buff, enqueue_cs);
13   }
14 }
```

The function `LL_delegate_unlock` (line 8) is used to unlock the lock when no message buffer was acquired. `LL_close_delegate_buffer` (line 12) will indicate to the lock holder that the message buffer is fully written. After the `LL_close_delegate_buffer` call, the delegated function (provided as parameter) can be executed with the message buffer given as parameter.

4 C++ Library

While the C library performs very well, its interface restrictions mean that a C++ programmer has to write many function wrappers so that operations could be delegated. For this reason, we also developed a C++11 implementation of QD locking, which makes delegation in C++ code easier. It provides the same variants of QD locks as the C library, namely `qdlock` and `mrqdlock`. For the user, it provides two delegation interfaces: `delegate_n`, which detaches the execution entirely, and `delegate_f`, which returns a future that is bound to the critical section. The initial example in C could look similar to this in C++:

```
1 mrqdlock lock;
2 void mult_cs(int* a, int b) { *a = *a * b; }
3 void mult(int* v1, int v2) {
4     lock.lock(); mult_cs(v1, v2); lock.unlock();
5 }
```

From the parameters to the critical section, it is clear what data needs to be transferred. As the type of each object is known at compile time, the compiler can produce code to forward the data to the critical section.

```
3 void mult(int* v1, int v2) {
4     lock.delegate_n(&mult_cs, v1, v2);
5 }
```

Furthermore, with C++11 lambda functions it is not even necessary to have a separate named function, like `mult_cs` above, for the critical section.

```
void mult(int* v1, int v2) {
    lock.delegate_n([](int* a, int b) { *a = *a * b; }, v1, v2);
}
```

Even with these simple examples one can see the flexibility of the library. Using functors – which lambda functions are – or normal function pointers, the library takes them together with their parameters and passes them to the QD lock, serializing each element as needed. However, this flexibility comes at a price. To determine the types of the parameters, a function is constructed by the compiler, which can deserialize the parameters and passes them to the function or functor specified by the user. This additional function pointer needs to be passed to the lock together with the parameters, resulting in a one word overhead when compared to the C version of the library. This can be avoided only for non-overloaded functions by passing the function address as a template parameter to the library code.

```
  lock.delegate_n<void (*)(int*, int), &mult_cs>(v1, v2);
```

Due to language restrictions, we need to specify the function signature as well, so that the compiler can type-check it later. For convenience, we provide a macro which inserts the function signature automatically.

```
  lock.DELEGATE_N(&mult_cs, v1, v2);
```

To return values from critical sections or wait for their actual execution, we use futures. This allows the programmer to execute an arbitrary amount of code between issuing a critical section and the point where the return value or side effect of it needs to be visible to the system.

```
int read(int* shared) {
  auto future = lock.delegate_f([=shared]() -> int { return *shared; });
  return future.get();
}
void add_two_and_wait_for_first(int* shared) {
  auto future = lock.delegate_f([=shared]() -> void { *shared += 1; });
  lock.delegate_n([=shared]() -> void { *shared += 1; });
  future.wait();
}
void add_two_and_block_both(int* shared) {
  auto future1 = lock.delegate_f([=shared]() -> void { *shared += 1; });
  auto future2 = lock.delegate_f([=shared]() -> void { *shared += 1; });
  future2.wait();
}
```

In the last example the waiting for the second future is sufficient, as the order of delegations is preserved. If a future is destructed, the behaviour depends on the implementation used, which is currently `std::future`. To avoid ambiguity it is best to always store the return value of `delegate_f` and explicitly decide when the future needs to be invoked.

For a more complicated example, we will use `std::map` in a thread-safe way. While the example does not actually use concurrency, the code could be plugged into a concurrent program. It first inserts two values, then reads one back using a delegation, and finally reads the other value back using `rlock`. Remember that `rlock` has to wait for the completion of prior delegated sections, which means the `future.get()` call at the end will always immediately return the value.

```
#include<map>
#include "qd.hpp"
int main() {
  std::map<int, double> map;
  mrqdlock lock;
  lock.delegate_n([&map]() { map.insert(std::make_pair(21, 2.56)); });
  lock.delegate_n([&map]() { map.insert(std::make_pair(42, 3.14)); });
  auto future = lock.delegate_f([&map](int key) {return map[key];}, 42);
  lock.rlock(); double r2 = map[21]; lock.runlock();
  double r1 = future.get();
}
```

5 Queue Delegation Locking for the Erlang Term Storage

In this section we will discuss the applicability of delegation locking to protect a shared key-value store, namely the Erlang Term Storage (ETS). ETS is a heavily used part of the Erlang programming language and is implemented in C for efficiency reasons. Erlang programs can use ETS as shared memory between threads (Erlang processes). Being shared memory, ETS has become a scalability concern on multicore machines [5]. Because of the complexity of ETS, it is difficult to apply efficient fine grained locking or lock-free techniques. This is especially true for the ordered_set ETS table type that is implemented as an AVL tree. Such ETS tables are currently protected by a single readers-writer lock. We will describe the steps we went through when porting the ETS code to use an MRQD lock instead of a readers-writer lock and the performance we got.

5.1 Porting

We have chosen to focus on the ETS operations `insert`, `delete` and `lookup` when porting. The first two operations are interesting since they do not have any return value and can thus be delegated to the current lock holder without any need to wait for their execution. The `lookup` operation is interesting because it can help us show how well QD locking works together with the multiple-readers extension. We divided the porting work into three steps of increasing difficulty, where each step produced working code that we could benchmark to measure the resulting performance. We started from an ETS code base of eight files with a total of 16 277 lines of code.

Step 1, delegate and wait: In this step we just delegate the original critical section and wait for its actual execution with the `LL_delegate_wait` function from the C library. In most situations this works without any semantic change of the original code. However, if thread-local variables are accessed inside the critical section, as was the case in ETS, care must be taken so the right thread-local variable is accessed. In ETS, the thread-local variable access was subtle since it was done in the read-unlock call of a readers-writer lock. To fix this issue we simply moved the read-unlock call to after the issuing of the critical section. Another way to deal with this problem would have been to pass a reference to the thread-local variable to the delegated function. In total this step required changing about 400 lines of code (60 of which were changes and 340 were additions, many to integrate with the existing locking structure).

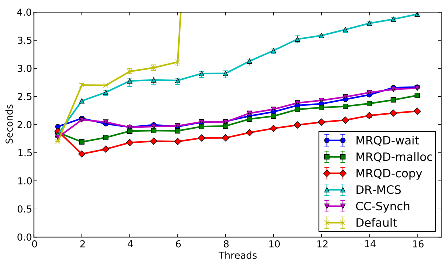
Step 2, delegate without wait: To delegate without waiting for the actual execution of the critical section required more changes. The original code did some checking of parameters inside the critical section that could result in a return value indicating an error. These checks did not need to be done inside the critical section and could simply be lifted out. The parameters to the `insert` and `delete` functions are allocated on the heap of the issuing thread (process in Erlang terminology) and can be deallocated as soon as the functions have returned. Therefore it is not safe to send references to these values to delegated critical sections. Instead, we changed the original code to allocate a clone of the value and send a reference to the clone. For the `insert` case, the clone is in a form that can be inserted directly into the table data structure. The effort of allocating a clone is therefore not wasted since a clone would need to be created anyway to store the object. Furthermore, since the cloning is done outside the critical section, this modification can also decrease the length of the critical section. However, if the object being inserted is replacing an existing larger object, the original code had less memory management cost because it would just overwrite the existing object. For the `delete` operation, both the allocation of the cloned key and its subsequent freeing incurs an overhead compared to the original code. This step required changes in about 400 more lines of code (760 if one starts counting the differences from the original code).

Step 3, delegate and copy directly into the QD queue: In this step we got rid of the need to do more memory management than the original code by copying all parameters needed in the critical section directly into the queue buffer of the QD lock. This also had the benefit of improving the cache locality for the helper thread that is executing the critical sections. Because all needed data for the critical sections is stored in a continuous array, data for several operations can potentially be read with a single cache miss. The only additional porting effort required in this step was the serialization of the key and the object to a form that can be stored directly into the QD queue. This step required changing only about 100 more lines of code.

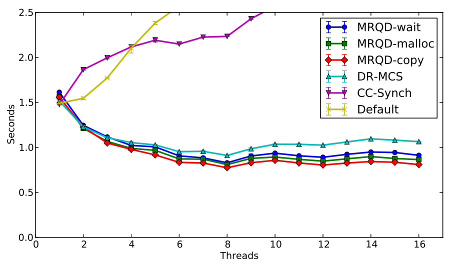
5.2 Performance Evaluation

We used ets_bench, a benchmark from BenchErl [1] to evaluate the performance and scalability of ETS tables of type ordered_set after applying each porting step described in the previous section. A detailed description of ets_bench can be found in a paper about the scalability of ETS [5], but basically it measures the performance of ETS under variable contention levels and distributions of operations. We ran the benchmark on an Intel(R) Xeon(R) E5-4650 (2.70GHz) chip with eight cores and two hardware threads per core (i.e., a total of 16 hardware threads running on eight cores). The machine ran Debian Linux 3.10-0.bpo.2-amd64 and had 128GB of RAM. All code was compiled using GCC version 4.7.2 with `-O3`. We pinned the software threads to logical processors so that the first eight software threads in the graphs were pinned to separate cores. Each configuration was run three times and we report the average run time. The minimum and maximum are shown as bars when varying enough to be visible.

The update only scenario presented in Fig. 1(a) shows the run time of N threads (Erlang processes) performing $2^{22}/N$ operations each. The inserted objects are Erlang tuples with an integer key randomly selected from the range $[0, 2^{16}]$. The operations are

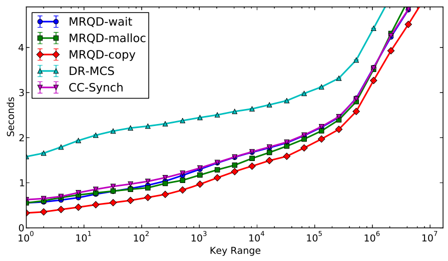


(a) Only Updates

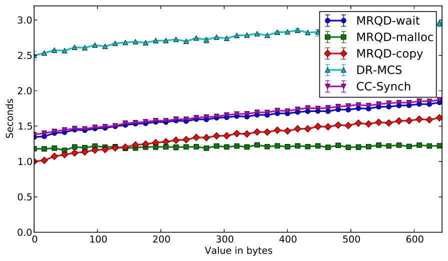


(b) 80% Reads, 20% Updates

Fig. 1. Scalability Benchmarks for ETS. Dataset size is 2^{16}.



(a) Key-value store structure size



(b) Stored value size

Fig. 2. Benchmarks varying the size of data. Using eight threads for all measurements.

`insert` or `delete` with equal probability. The line labeled Default represents the default readers-writer lock used by ETS. It is optimized for frequent reads and the uncontended case. Therefore, it does not scale well with parallel writers. We also include the state-of-the-art readers-writer lock DR-MCS presented by Calciu *et al.* [2]. The MCS lock [8] that DR-MCS uses to synchronize writers is good at minimizing cache coherence traffic in the lock hand over. However since MCS is a queue based lock, the thread that executes critical sections is likely to alternate between the cores. This is causing a lot of expensive cache coherence traffic inside the critical section which is one reason why its performance is worse than all delegation based locks. CC-Synch [3] is included in our comparison to show an alternative delegation locking mechanism that can be used with the same interface as QD locking. MRQD-copy, which is corresponding to porting step 3 in the previous section, performs best in all contended cases. MRQD-copy is closely followed by MRQD-wait, CC-Synch (both step 1) and MRQD-malloc (step 2). MRQD-wait, CC-Synch and MRQD-malloc are almost indistinguishable except for the case with two threads. MRQD-malloc performs better in this case because of its ability to continue directly after delegating work.

Figure 2(a) shows how the performance varies with the set size (key range) while the thread count is fixed to eight. The figure shows that the performance advantage of delegation based locks compared to DR-MCS is larger with smaller set sizes. This is expected since with smaller set sizes, the length of the critical sections is smaller and the helper thread can keep most data in the private cache while executing operations. The larger slowdown is caused by running out of shared cache and hitting memory.

Figure 2(b) shows performance with varied value size and thread count fixed to eight. MRQD-malloc is least affected by the size of the value passed together with the key to the insert operation. The reason is that MRQD-malloc clones the key and the value before the critical section so that the size of the critical section stays largely unaffected by the value size. MRQD-copy, which is the best performing variant for value sizes up to a few cache lines, has to read the whole value from the queue inside the critical section. In Fig. 2(b) we use the key range $[0, 2^{10}]$ to make the effect of the value size more visible.

In Fig. 1(b) we show the performance when 80% of the operations are `lookups` and the rest are `inserts` and `deletes` with equal probability. Unsurprisingly, since they use the same read synchronization algorithm, the order between MRQD-copy, MRQD-malloc, MRQD-wait and DR-MCS is the same as in the update only scenario. With 99% `lookup` operations the difference between MRQD-copy, MRQD-malloc, MRQD-wait and DR-MCS is very small, but the performance advantage of the four delegation based algorithms gets larger and larger with more updates. Due to space limitations we only show measurements for 80% `lookups` in this paper.

6 Related Synchronization Algorithms

QD locking comes from the same line as other delegation locking techniques such as the detached contexts algorithm by Oyama *et al.* [9], the flat combining algorithm by Hendler *et al.* [4], and the Synch algorithms by Fatourou and Kallimanis [3]. The common property of all these techniques is that a helper thread can execute critical sections for other threads. Under contention this means that the same processor core can execute many critical sections, one after the other, and thereby keep the protected data in the same fast private cache. In the uncontended case the delegation locking algorithms behave like traditional locking algorithms, although most of them have a small overhead for opening the delegation data structure.

The algorithm of Oyama *et al.* shares with QD locking the ability to continue a thread's execution immediately after delegation. However, unlike QD locking, under high contention it will starve the helper thread. Furthermore, its need to allocate contexts for critical sections can easily become another bottleneck. While flat combining has a very efficient handover of critical sections to the helper thread, it does not execute critical sections in arrival order, which means that threads have to block until their delegated section is executed in order to maintain linearizability. The Synch algorithms execute the critical sections in arrival order and have efficient handover of critical sections. To avoid the problem of starving helper threads, they allow the helper to stop helping. In this case the thread owning the next delegated section becomes the new helper, which means that all threads must wait until their section is executed as well.

In short, QD locking offers delegation without waiting, and does not starve helper threads, which we have not found possible in other algorithms. This is achieved through a preallocated queue buffer which enables both a fast handover and limiting the amount of work a helper thread has to do. Experiments reported elsewhere [6] show that QD locking performs better than other delegation algorithms at various contention levels.

7 Discussion

Many delegation algorithms have been shown to outperform non-delegating locking approaches, but compared to traditional locks the support for programming with delegation has been somewhat limited so far. Code for using flat combining[3] and the Synch algorithms[4] is available online, but significant work is required to use them in programs other than the benchmarks these implementations were written for. Making use of delegation based locking thus requires a substantial investment from programmers. The libraries presented in this paper aim to make it easier to use delegation locking, lowering the entrance barrier and enabling more programs to take advantage of them. In fact the APIs of our libraries can be used for other delegation locking schemes as well.

As has been illustrated in this paper, many types of critical sections can be ported straightforwardly. However, some other types of critical sections cannot use delegation algorithms. As an example, hand-over-hand locking interleaves `lock` and `unlock` calls, resulting in partially overlapping critical sections, which cannot be delegated. This is not a problem for many applications, as the most common use of locks allows critical sections to be tranformed into subroutines.

For latency critical programs a complaint could be that the helper can get stuck executing critical sections for other threads. This is actually not a big problem as the maximum amount of help that a thread does can be limited using a parameter to the lock. Furthermore, if the critical section allows for delegating without waiting for the actual execution, threads can continue where they would have needed to wait with traditional locking. The ability to continue without waiting for the actual execution of a critical section could even make QD locking an attractive alternative if otherwise only non-blocking algorithms would be considered.

8 Future Work and Concluding Remarks

To ease the porting of large programs with many locks and critical sections, it would be useful to have a tool that detects critical sections that can be used with delegation locking. Such a tool would need to detect if the critical section can be factored into a function or if it uses thread local variables. Additionally, the tool could detect implicit return values from the critical sections and decide whether delegation without waiting for the actual execution can be used or not.

More options for performance tuning in the libraries will be added in the future. For example, a companion paper presents a hierarchical QD (HQD) lock variant for NUMA systems [6]. On such systems, it has faster delegation and execution at the cost of more often having to wait for delegate calls to succeed. This means that it is not always better to use this variant on NUMA systems, but rather depends on the workload. Another extension would be to make `delegate` wait passively instead of spinning. The libraries also could incorporate other locking and delegation algorithms to facilitate comparisons. For the C library it is already possible to use it to compare against a few other locking algorithms, including DR-MCS and CC-Synch as used in this paper.

[3] `http://github.com/mit-carbon/Flat-Combining`

[4] `https://code.google.com/p/sim-universal-construction/`

All in all, there is room to improve the libraries, but also potential to apply them. For writing new code, the idea to `delegate` operations is fairly intuitive, but when porting from `lock` and `unlock` some effort is required to achieve optimal performance. Even then, experiments in this and other papers show that delegation can improve performance twofold and sometimes more. For many applications this may be enough incentive to give QD locking a try, and the libraries presented here make it a lot easier to get started.

References

1. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.E.: A scalability benchmark suite for Erlang/OTP. In: Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop, pp. 33–42. ACM, New York (2012)
2. Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V.J., Shavit, N.: NUMA-aware reader-writer locks. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 157–166. ACM, New York (2013)
3. Fatourou, P., Kallimanis, N.D.: Revisiting the combining synchronization technique. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 257–266. ACM, New York (2012)
4. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 355–364. ACM, New York (2010)
5. Klaftenegger, D., Sagonas, K., Winblad, K.: On the scalability of the Erlang term storage. In: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, pp. 15–26. ACM, New York (2013)
6. Klaftenegger, D., Sagonas, K., Winblad, K.: Queue delegation locking (2014), `http://www.it.uu.se/research/group/languages/software/qd_lock_lib`
7. Lozi, J.-P., David, F., Thomas, G., Lawall, J., Muller, G.: Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In: Proceedings of the 2012 USENIX Annual Technical Conference, Berkeley, CA, USA, pp. 65–76. USENIX Association (2012)
8. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (1991)
9. Oyama, Y., Taura, K., Yonezawa, A.: Executing parallel programs with synchronization bottlenecks efficiently. In: Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications, pp. 182–204. World Scientific (1999)
10. Sridharan, S., Keck, B., Murphy, R., Chandra, S., Kogge, P.: Thread migration to improve synchronization performance. In: Workshop on Operating System Interference in High Performance Applications (2006)
11. Suleman, M.A., Mutlu, O., Qureshi, M.K., Patt, Y.N.: 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, pp. 253–264. ACM, New York (2009)