Brief Announcement: Queue Delegation Locking

David Klaftenegger
Dept. of Information Technology
Uppsala University, Sweden
david.klaftenegger@it.uu.se

Konstantinos Sagonas
Dept. of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Kjell Winblad
Dept. of Information Technology
Uppsala University, Sweden
kjell.winblad@it.uu.se

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 ability to use modern hardware efficiently and do useful work while or instead of waiting. This brief announcement sketches the idea and implementation of *queue delegation locking*, a synchronization mechanism that provides high throughput by allowing threads to efficiently delegate their critical sections to the thread currently holding the lock and by allowing threads that do not need a result from their critical section to continue executing immediately after delegating their work. Experiments show that queue delegation locking outperforms leading synchronization mechanisms due to the combination of its fast operation transfer with its ability to allow threads to continue doing useful work instead of waiting. Thanks to its simple building blocks, even its uncontended overhead is low, making queue delegation locking useful in a wide variety of applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

locking; multi-core; NUMA; synchronization

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 lock 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. In this work we focus on a locking approach that sends operations to the thread holding the lock instead of transferring the lock between threads. This locking approach is called *delegation*, and the thread executing other threads' critical sections is called the *helper*. The main reason why delegation algorithms perform well is improved locality as the helper thread only seldomly needs to wait for memory transfers between caches in different cores or even NUMA nodes. In addition, *detached execution* allows threads to continue execution before the delegated critical section has been executed. However, in its original form [6] the detached execution algorithm has some overhead and severe starvation issues for the helper thread. Newer approaches [1, 2, 4] require 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 approaches, our locking mechanism, called *Queue Delegation (QD) locking*, allows efficient delegation while also permitting detached execution without starving the helper thread.

Main Ideas. The main idea of QD locking 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. If successful, this thread becomes responsible for eventually executing the operation. The other 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 limiting the amount of work that the helper 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.

All requirements for queue delegation locking are met by assembling two simple components: a *mutual exclusion lock* to determine which thread is executing operations, and a *queue* to delegate operations to the lock holder. We will describe these components in the next section. In the full paper [3] we also describe how to extend the basic algorithm

SPAA'14, June 23–25, 2014, Prague, Czech Republic.
ACM 978-1-4503-2821-0/14/06.
http://dx.doi.org/10.1145/2612669.2612714.

```c
void delegate(QDLock* l, Operation op) {
  while (true) {
    if (try_lock(&l->lock)) {
      open(&l->queue);
      execute(op);
      flush(&l->queue);
      unlock(&l->lock);
      return;
    } else if (enqueue(&l->queue, op)) return;
    yield();
  }
}
```

Figure 1: The `delegate` function

to allow parallel access to multiple readers efficiently, and hierarchical variants for NUMA systems.

2. IMPLEMENTATION

As mentioned, queue delegation locks are built from two 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.

The second component, the delegation queue, is required to store delegated operations. Semantically, it is a *tantrum queue* [5]. 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.

2.1 Queue Delegation Lock Implementation

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; see Figure 1. 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 that 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*; i.e., 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 this value 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.

```c
void open(DelegationQueue* q) {
  q->counter = 0;
  q->closed = false;
}

bool enqueue(DelegationQueue* q, Operation op) {
  if (q->closed) return CLOSED;
  int index = fetch_and_add(&q->counter, 1);
  if (index < ARRAY_SIZE) {
    q->array[index] = op; /* atomic */
    return SUCCESS;
  } else return CLOSED;
}

void flush(DelegationQueue* q) {
  int todo = 0;
  bool open = true;
  while (open) {
    int done = todo;
    todo = q->counter;
    if (todo == done) { /* close queue */
      todo = swap(&q->counter, ARRAY_SIZE);
      open = false;
      q->closed = true;
    }
    if (todo >= ARRAY_SIZE) { /* queue closed */
      todo = ARRAY_SIZE;
      open = false;
      q->closed = true;
    }
    for (int index = done; index < todo; index++) {
      while (q->array[index].fun_ptr == NULL); /* spin */
      execute(q->array[index]);
      q->array[index].fun_ptr = NULL; /* reset */
    }
  }
}
```

Figure 2: The delegation queue implementation

2.2 Delegation Queue Implementation

The delegation queue can be efficiently implemented with a fixed-size buffer array and an index counter. The `enqueue` function tries to allocate a slot in the buffer array by incrementing the index counter with an atomic `fetch_and_add` operation. The queue is closed if the value of the index counter is greater than the index of the last slot in the array buffer. The `flush` function executes enqueued operations until the queue is closed and all operations have been processed. Pseudocode for the delegation queue is shown in Figure 2. Note that the `closed` flag in the pseudocode is just a performance optimization and is not needed for correctness.

2.3 Extensions

The basic QD locking algorithm can be extended in various ways, which we present in the full paper [3]. In particular, we describe a hierarchical variant (HQD), which targets NUMA systems as well as multi-reader QD (MR-QD) locks, which allow parallel access for readers as in traditional reader-writer locks. We also discuss how to adapt the algorithm to guarantee starvation freedom for all threads, and how to extend the interface to ease the porting process.

3. PERFORMANCE

Here we give a summary of a performance evaluation comparing QD locking with related synchronization algorithms. Refer to the full paper [3] for graphs and for more details.

3.1 Benchmark Descriptions

We measured the performance of different locking algorithms with a varying amount of threads and contention on a machine 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), using three benchmarks. The first measures the throughput of random operations on a shared priority queue (`insert` and `extract_min`) implemented by protecting a sequential priority queue with the synchronization algorithms of Oyama *et al.* [6], with flat combining [2], with CC-Synch and H-Synch [1], and with QD lock and HQD lock. The second benchmark compares MR-QD locks with state of the art readers-writer locks. Finally, we applied MR-QD locks on the code of the Kyoto Cabinet in-memory database (version 1.2.76) to evaluate their performance on a code base that uses traditional locks.

3.2 Summary of the Results

The QD and HQD variants perform better than or similar to the best of the other synchronization mechanisms in all scenarios we tested. As clarified by experiments with variants of QD and HQD in the full paper [3], there are several reasons for this. First, QD locking allows a thread that performs a write only operation (`insert`) to continue execution directly after delegating the operation. Except for the QD variants, the Oyama *et al.* algorithm is the only locking scheme where threads can continue without waiting for delegated operations. However, that algorithm performs poorly compared to the QD variants because of the overheads it contains and its inability to take advantage of modern hardware. We also noticed that the helper thread is often starved in the scheme of Oyama *et al.* The positive effect of being able to continue without waiting for the issued operation becomes more apparent when the threads have more local work to do between the priority queue operations. Since continuing execution means the next priority operation is issued faster, there can be more helped operations per lock acquisition even under low contention. The high number of helped operations per lock acquisition is also the reason why QD locking performs better than HQD and H-Synch (the NUMA-aware variant of CC-Synch) with low contention levels even when the threads are running on different NUMA nodes. The node level contention is not large enough to fill the delegation queue while contention on a system level is high enough to fill it. Secondly, the helper thread can read and execute delegated operations extremely efficiently in the QD variants. Since a continuous array is used to store operations in the delegation queue, several operations can be read with only one cache miss which is not the case in the other delegation algorithms. Finally, we note that the atomic `fetch_and_add` instruction used to enqueue operations in the delegation queue is, unsurprisingly, better than simulating the `fetch_and_add` with a CAS loop, even though the CAS loop variant still performs well.

The results of the second benchmark show that traditional readers-writer locks can be outperformed by a QD lock with the MR extension on some workloads. Being able to delegate write operations without waiting for their actual execution plays very well with parallel read-only operations. Threads can continue and issue read operations directly after issuing a write operation so that read-only operations can bulk up and execute in parallel. Similar results are observed in the Kyoto Cabinet benchmark, which also shows that performance is dependent on both fast delegation and not having to wait for the execution of critical sections.

4. CONCLUDING REMARKS

We have sketched the idea and implementation of a novel synchronization mechanism called queue delegation locking, showing the essential building blocks only. A key advantage of QD locking is its ability to delegate operations without waiting for response, its simplicity and its small communication cost. Experiments show that QD locking can outperform current state-of-the-art synchronization algorithms such as that of Oyama *et al.*, flat combining, CC-Synch and H-Synch. Our results also suggest that multi-reader QD locks can be a more performant alternative to readers-writer locks for some use cases. For a more in-depth explanation of QD locking and its variants and for an extensive comparison with related work refer to the full paper [3].

Acknowledgments

This work has been supported in part by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software" and the Uppsala Programming for Multicore Architectures Research Center.

5. REFERENCES

[1] 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.

[2] 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.

[3] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue delegation locking, 2014. Preprint available from `http://www.it.uu.se/research/group/languages/software/qd_lock_lib`.

[4] 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.

[5] 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.

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