Forecasting Lock Contention Before Adopting Another Lock Algorithm

Xiaoyue Pan  David Klaftenegger  Bengt Jonsson  
Department of Information Technology  
Uppsala University  
{xiaoyue.pan,david.klaftenegger,bengt.jonsson}@it.uu.se

Abstract—Locks are often used in parallel applications to ensure correct behavior, but frequently induce performance bottlenecks caused by contention at frequently accessed locks. There are state-of-the-art locking algorithms, whose performance under high contention is superior to that of the most commonly used ones. However, some of these have a different interface, and adopting them incurs a reimplementation effort. It is therefore important to efficiently estimate the potential benefit of adopting another lock algorithm, prior to investing effort for reimplementation. We use the Queue Delegation Locking algorithm (QD locking) as an example, which to our knowledge has the best performance among such algorithms. For high-contention scenarios, delegation-based locking algorithms perform well due to their ability to delegate critical sections to a single thread, which requires using a different interface than the traditional lock()/unlock(). In the paper, we present a method that allows predicting the contention under the QD locking algorithm which does not require any reimplementation. In our method, one only needs to gather a set of machine-specific parameters and profile the target application in a single-threaded run to obtain the lock access patterns and holding times. Given these inputs, our method uses queueing networks to predict the contention at each lock, aiding programmers in judging the impact of adopting QD locks. We validate the model using generated benchmarks with multiple locks and showcase the accuracy on an in-memory database application, which is 24% for all 3500 configurations of our benchmark and 14% for critical sections sized no smaller than 1000 nanoseconds.

I. INTRODUCTION

Parallel applications often need synchronization to regulate accesses to shared data. Locks are a widely used synchronization method that provides mutually exclusive access to shared data in critical sections. While locks may be vital for guaranteeing correctness in parallel programs with shared data, they can also cause contention that slows down the execution. When multiple threads simultaneously attempt to access the same lock, only one of them can proceed and the other threads must wait, wasting valuable execution time. The time wasted in lock contention typically increases with the number of threads, thereby incurring scalability bottlenecks that severely limit the performance that can be achieved by parallel execution.

Scalability bottlenecks caused by lock contention can be reduced in several ways. Possible remedies include to use lock-free data structures, to change the ways in which locks are accessed, and to replace the lock implementation by another one. Algorithms for lock-free data structures is still a challenging research topic, and there is no standard recipe for developing efficient such algorithms; moreover it is a major implementation effort to replace critical sections protected by locks by lock-free data structures. Changing the ways locks are accessed requires restructuring the program, which is also a significant implementation effort, and is often not certain to actually remove scalability bottlenecks. Using another lock implementation requires understanding of the advantages and disadvantages of existing alternative lock implementations (e.g., Pthread mutex, spin locks, MCS locks [19], CLH queue locks [4] [17]) and choose the most suited one for the target application.

To illustrate the difference in performance between different lock implementation, Figure 1 shows the performance of different locks for a microbenchmark, taken from [12], in which a number of parallel threads (up to 64), each executing on a separate core, alternate between performing a small amount of local work and accessing a shared priority queue protected by a lock. When the number of threads increase beyond 16, all algorithms decrease in performance, as the system moves from using a single processor socket to multiple NUMA sockets. The lock algorithms that implement an interface with lock()/unlock() operations (the ubiquitous Pthread locks as well as the CLH spinlock) exhibit performance that drops below that of a single thread executing the benchmark. Cohort locks are specifically designed for NUMA systems, and can maintain their performance. Flat Combining [8], CC-Synch [7] and QD locks [12] are three delegation-based locking algorithms, and clearly outperform the lock()/unlock()
based algorithms, and can compete with a state-of-the-art lock-free implementation of priority queues [15].

Even though delegation-based locking algorithms clearly perform better than the more common lock() / unlock() ones, there are still obstacles for replacing an existing (typically Pthreads-based) lock implementation by a delegation-based one: (i) delegation-based locks have a different interface than lock() / unlock() locks, implying that it is a significant effort to port an application from lock() / unlock() locks to such an algorithms, and (ii) before investing the porting effort to replace the lock algorithm, it is not clear how large will be the reduction in lock contention, and whether this reduction warrants the porting effort. In this paper, we address this second obstacle.

In this work, we focus on Queue delegation locking (QD locks) [12], since it yields the best performance (see Figure 1). In addition, it is free from two performance issues of the traditional lock() / unlock() locks. First, on a multi-core system, moving critical sections between cores increases the execution time of critical sections. On multicore systems, threads often alternate executing their critical sections. This will most likely move data used in these critical sections from one core to another. Such a data movement can trigger cache coherence misses when one core modifies shared data that another core then accesses, which leads to increased execution time of critical sections. Second, the lock() / unlock() operations have a nonnegligible overhead, which also affects all threads that are waiting to access the lock, thereby contributing to lock contention. This effect can become very significant for small critical sections.

The QD lock, on the other hand, allows one thread to execute critical sections of other threads. This design makes the QD lock diminish the two performance problems of lock() / unlock() locks in situations where the lock is highly contended. First, QD locks suffer from fewer coherence misses compared to other locks. This is because when a helper thread (one that can execute other threads’ critical sections) executes the critical section of another thread after executing its own, there is no movement of shared data, since it has already been moved to the executing core’s private cache. This leads to critical section time staying almost constant as the number of threads increases. Second, once the helper thread pays the overhead of acquiring the lock, it does not need to re-acquire the lock when it executes the critical section of another thread.

Challenge If a parallel application, which uses the widely adopted lock() / unlock() lock interface, exhibits scalability problems due to lock contention, it would be natural to consider replacing the lock algorithm by the QD lock to reduce contention. A prerequisite for justifying the involved implementation effort is then to determine beforehand how large will be the reduction in lock contention. One way to find out is to measure the lock contention of the current lock implementation, reimplement the program with QD lock, measure the lock contention with QD lock, and then compare the two to determine how much QD locking would reduce contention. However, this is not always practical. First, the program source code may not be available. Second, even if the source code is available, the reimplementation may require significant effort, especially for programs with a big number of locks or complicated structures. In addition, such an effort may not be worthwhile if another lock implementation fails to reduce the lock contention. Hence, it would be desirable to have a simple way to forecast the lock contention with an alternative lock algorithm before adopting. This can not only shed lights on how a different lock implementation affects the lock contention, but also guide other program optimizations.

Contribution In this paper, we present a technique for predicting the performance influence of using delegation locks in programs originally implemented with non-delegation locks. Our technique provides a prediction without reimplementing the program with a delegation lock. It is based on profiling the application to extract each thread’s pattern of lock accesses and time needed to perform critical sections. The profiled data is then inserted into a queueing-network based model of the application, which also includes a performance model of the QD locking mechanism. By analyzing the model, we obtain a quick assessment how much a reimplementation with QD locks would reduce the lock contention for any number of threads.

In summary, in this paper we present the following contributions:

1) We provide a framework for measuring Pthread mutex contention. This framework can be easily extended to any other non-delegation-based locks. We show how this framework can be used to identify whether lock contention is a scalability bottleneck.
2) We present the first analytic model to estimate the lock contention for delegation locks given the parameters of the delegation locks and a program’s lock access pattern, which can be easily obtained from a single-threaded run.

The rest of the paper is organized as follows. Section II discusses the different types of lock implementations and related work. Section III discusses the QD lock mechanism in detail. Section IV presents the queueing networks model of the QD lock. Section VI evaluates the accuracy of the model in section IV. Section VII shows how we can use the model to predict the lock contention and guide optimization. Section VIII concludes the paper.

II. RELATED WORK

Locks and categories: There are many algorithms for mutual exclusion available, and it is often hard to choose the right one for any given application. To get a rough overview this section looks at a number of different algorithms and highlights their main differences. They can be categorized according to the way critical sections are executed. In lock() / unlock() style locks, each thread acquires a lock, executes a critical section, and releases the lock. Contrary to this, delegation locks allow threads to hand over their critical section to a different thread. As a single thread can execute critical sections faster by reusing hardware caches and avoiding synchronization overheads, this leads to higher performance for contended locks. Optimization of delegation locks focuses on the performance of delegating sections, executing them, and signalling the issuing thread. Examples for algorithms
that can be directly applied to implement delegation locks are the method of Oyama et al. [21], flat combining [8], CC-Synch and H-Synch [7], and queue delegation locking [13]. Other techniques aiming to exploit performance in a similar fashion include remote core locking [16], thread migration [23] and acceleration on asymmetric multi-cores [24]. With lock()/unlock() locks, the contended threads need to wait until the lock is released and try to take the lock again. Here, the different algorithms generally focus on the time it takes to acquire and release a lock, often including specific mechanisms to hand over the lock to another thread waiting for it. Queue based algorithms like MCS [20] and CLH [3], [18] avoid waking all waiting threads to re-attempt taking the lock by enforcing a FIFO order when they need to wait for another thread that currently holds the lock. This is advantageous as only the next thread in the FIFO queue needs to be woken when the lock is released, avoiding a lot of contention on lock handover. However, the strict FIFO ordering is also the problem of these algorithms: Modern multi-core computers often cause different access latencies depending on which core currently holds some data that is requested. If the data is nearby, the lookup can succeed significantly faster than when it is currently owned by a core on another physical processor chip. To exploit this hierarchical nature of computer systems, cohort locks [5] prefer to hand over the lock to a nearby core, which again speeds up lock handover.

**Measuring lock contention:** There are two kinds of tools to measure the lock contention: instrumentation based and performance counter based. While the instrumentation based tools [2] [10] are more general and flexibility, they usually introduce a bigger overhead and may cause cache pollution while keeping the data structures used for the bookkeeping. The performance counter based tools are fast and more accurate, yet usually requires hardware support. There are various methods to speedup the instrumentation based tools and minimize the cache pollution. For example, Tallent et al [25] measure the lock contention using sampling. They also suggest three different strategies for giving insight to the programmer into which parts of the program should be “blamed” for observed lock contention, and should hence be modified to improve performance. Bryant and Hawkes [2] propose Lockmeter to instrument spin locks in a Linux kernel. It is designed to minimize cache disruptions during the instrumentation. Huang et al [10] use a hardware memory tracing tool to reduce the memory interference of the measuring tool on the target program.

**Analytic models for lock contention:** Eyerman and Eekhout [6] present a probabilistic model for the critical sections for parallel programs with simplified assumptions about the program, e.g., all threads execute the same code and the locks are accessed uniformly. A main insight is that the impact of critical sections of a program can be included into Amdahl’s law by splitting into a sequential fraction and parallel fraction. The sequential fraction is proportional to the probability of entering the critical section multiplied by the contention probability. This insight is aimed to give guidelines to future micro-architectural multi-core design. Our previous work [22] proposes an analytic model predicting the lock contention of Posix thread mutex on a multi-core system. The model takes the same parameters as our proposed model in this paper, i.e., the lock access pattern and lock holding times, obtained from a single-threaded profile run. The proposed model can be used to estimate how much lock contention another lock with the lock()/unlock() interface would have for the same program. However, it can not be easily extended to delegation locks.

### III. UNDERSTANDING THE QUEUE DELEGATION LOCK MECHANISM

The delegation locking algorithm chosen for comparison here is queue delegation locking [12], [13]. While multiple algorithms are suitable for building delegation locks, queue delegation locking (QD lock) is readily available in library form [13]. Additionally, it has been shown to provide the best performance among the available algorithms [12].

The main idea behind the QD lock implementation is that threads that fail to acquire a lock 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. QD locks are built from two main components: Firstly, there is a normal mutex lock, which supports a tryLock operation. The tryLock operation does not block, but takes the lock and returns success if the lock was previously not taken. Secondly, there is a delegation queue, in which operations can be stored by threads while the mutex lock is taken.

In this work, we assume threads wait for their delegated operation before delegating another one. As two threads that try to delegate an operation concurrently do not have any order between them, the order in which they end up in the delegation queue is not important. 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.

Figure 2 shows a pseudo code description of the queue delegation lock mechanism. Threads either succeed in a tryLock operation on the internal mutex lock (line 3), or insert their operation into the delegation queue (line 13). A thread succeeding to take the lock becomes the helper thread. It executes its own operation, then executes all operations in the delegation queue and finally unlocks the lock at the end (line 4 through 11). Non-helper threads (lines 13 through 15) try to insert their operations into the delegation queue. This will succeed if the helper has not finished executing critical sections yet and there is still space in the delegation queue. After this, it waits for its critical section to be executed by the helper thread. Note that only the helper thread locks and unlocks the lock.

In lock()/unlock() locks, there is overhead associated with handing the lock over from one thread to another. This overhead is largely dictated by hardware, as taking/releasing a lock requires at least a write to memory. QD locks only incur this lock overhead (LO) in the helper thread. Other threads instead have two separate overheads for
inserting an operation into the delegation queue and waiting for the operation’s execution. The former consists of creating an appropriate structure for the operation (parallel) and the delegation overhead (DO) at the insertion itself (sequential). As the insertion can be implemented using fetch-and-add, it can proceed even when there is contention. Additionally, there is a chance of repeated failure to either take the lock or insert the operation, which was not observed in practice, but could be averted using a slightly more complicated algorithm.

```
1: method DELEGATE(operation)
2: while true do
3:   if TRYLOCK(lock) == success then
4:     execute operation
5:   while delegation queue not empty do
6:     take operation, from delegation queue
7:     execute operation,
8:     set result flag,
9:   end while
10:  UNLOCK(lock)
11:  return
12: else
13:   if TRYINSERT(operation) == success then
14:     wait for result flag;
15:   return
16: end if
17: end while
18: end method
```

Fig. 2: DELEGATE method of a queue delegation lock. TRYLOCK locks an internal lock if it is available, but does not block. UNLOCK unlocks the internal lock. TRYINSERT tries to insert operation into the delegation queue, which only succeeds if the internal lock is still locked.

Figure 3 shows a scenario where three threads $thread_0$, $thread_1$ and $thread_2$ try to access the shared lock. $thread_0$ arrives before $thread_1$ and $thread_2$. After succeeding in a tryLock operation and paying a lock overhead (denoted by LO), $thread_0$ becomes the helper. $thread_1$ and $thread_2$ arrive when $thread_0$ is still holding the lock and thus become non-helper threads. After they each pay a delegation overhead, their operations are available to the helper thread, which can execute them back-to-back without additional synchronization. Once the operations have been executed, there is a short synchronization delay before each non-helper thread will observe that their operation is completed and finish. Finally, the helper will also finish once all operations have been executed.

For the helper thread, the time it spends at the lock includes the lock overhead (LO in Figure 3), its critical section size ($cs_0$) and the time to execute all other non-helpers’ critical sections ($cs_1$ and $cs_2$). For the non-helper threads, the time at the lock consists of the delegation overhead (DO), its critical section size ($cs_1$ or $cs_2$) and the waiting time for other threads’ critical section. The delegation overhead (DO) increases as the number of non-helper threads waiting at the lock increases. We denote the overhead with DO($k$) where $k$ is the number of non-helper threads waiting at the queue. Measuring this overhead is discussed in detail in Section IV-F.

IV. Predicting Reduction in Contention

A. Queueing networks

Queueing networks are often used to model and analyze the performance of systems in which jobs access shared resources and thereby cause congestion. Evaluating whether lock contention could be a performance bottleneck in a parallel application is essentially about analyzing whether certain locks are points of congestion in the whole program. From this perspective, queueing networks is a suitable model to analyze lock contention. Queueing networks come with structures that are suitable for capturing programs with mutually exclusive locks: locks can be represented by single-server nodes in a queueing network, and local computation that proceed in parallel can be represented by infinite-server nodes.

A queueing network consists of a set of nodes (aka. service stations). Jobs (aka. customers) travel between nodes at which they receive “service”. A node can either serve one job at a time (single-server node) or some specified number of jobs concurrently. There are also infinite-server nodes where an unbounded number of jobs can be served simultaneously. When a job arrives at a node which is currently serving a maximal number of jobs, it must wait in the queue of the node until currently served jobs have left the node. Each node has a defined service time, which is the time needed to serve a job at that node. The service time is given as a probability distribution, typically it is exponentially distributed. The flow of jobs between nodes is defined by a routing matrix $R$, which for each pair of node $i$ and node $j$ specifies the probability that a job visits node $j$ after having been served at node $i$. Networks can be single-class, meaning that all jobs obey the same routing matrix and have the same service time for each given node, or multi-class, meaning that the jobs can be partitioned into classes, in which jobs follow the same pattern. Furthermore, networks can be either closed, meaning that they have a constant number of jobs that never leave or enter the network, or open, in which case jobs can enter and leave an environment.
B. Modeling the Locking Behavior of Programs

In this subsection, we introduce our basic model for predicting the lock contention. In this section, we ignore the overheads (i.e., lock overhead or delegation overhead) associated with lock accesses: these will be introduced in Section IV-D.

Our proposed model can handle programs with asymmetric lock behavior (different threads can access locks according to different patterns). To simplify the explanation of the model, we assume the target program has symmetric lock behavior. We model the locking structure of a program by a closed network with a constant number of jobs: this is because the number of threads is constant in our target programs. We describe the lock accesses of a parallel program as a closed queueing network where the number of jobs is the number of threads in the program. The structure of the network corresponds to the control structure of the program that is executed by each thread.

A program is composed of code segments. Each code segment $cseg_i$ can be either a local computation segment or a lock segment. A local computation segment can proceed in parallel with execution of other threads and a lock segment is a critical section. We represent each code segment $cseg_i$ as a node $node_i$ in the queueing network. Each local computation segment is represented by an infinite-server node whose service time is the execution time of the segment. Each lock segment is represented by a single-server node whose service time is the execution time of the corresponding critical section. We model the two different patterns of lock accesses, exhibited by helper threads and non-helper threads, by using a two-class network. We statically set the number of helper threads to one. In practice, there can be more than one helper thread in the queueing network (at most one at each lock at the same time). Since the queueing networks model does not allow jobs to switch classes or have a dynamic number of jobs of each class, we make the simplification that there is always one helper thread in the whole network. When there are more than one locks in the program, several helpers can be delayed at the same time to help other threads. Our simplification makes one helper thread count for the delays of all helper threads. It implies that when the only helper thread is at one lock, it can not block non-helper threads at another lock. This may cause an underestimation in the lock contention of the non-helper threads.

Since helper and non-helper threads have different access patterns at a lock, we cannot model a lock simply as a single-server node with two different service times. Such a representation cannot model the fact that the helper typically executes in the lock for a period of time which is proportional to the number of non-helper jobs that are served in the queue, which is also spent in parallel with the execution of a non-helper’s critical section. Instead, we approximate a lock by extending the single-server node representing the lock with an infinite-server node, which is visited only by helper threads, and has a service time that is the average time needed to execute the non-helper’s critical sections. In Figure 4 we show the structure of these two locks. The node with servers labeled $1 \rightarrow \infty$ is an infinite-server node. For a lock corresponding to $node_i$, let $t_i^h$ and $t_i^{nh}$ be the time for executing the critical section by a helper thread and a non-helper thread. Then service times at the single-server node are $t_i^h$ and $t_i^{nh}$ for the helper and non-helper thread, as expected. Non-helper threads go directly to the next node, corresponding to a segment of local computation, after exiting the node. Helper threads instead go to the extra helper node, denoted $node_i^h$, which is an infinite-server node with service time $t_i^{help}$. Since the time $t_i^{help}$ depends on the number of jobs in the queue, it cannot be determined statically, but will be determined by a fixpoint procedure for analyzing the network together with finding the corresponding values for $t_i^{help}$.

In summary, our queueing network model for a program with QD locks is obtained as a closed queueing network with 2 classes of jobs (corresponding to helpers and non-helpers), with one single-server node for each lock, and one infinite-server node for each segment of local computation. The routing matrix for non-helper threads specifies for each pair of nodes $(node_i, node_j)$ the probability that a job visits node $node_j$ immediately after visiting node $node_i$. It is the fraction of threads that enter the lock or computation segment corresponding to node $node_i$ after leaving that of node $node_i$, which we obtain by profiling (see Section V). The routing matrix for helper threads is the same as that for non-helper threads, except that it passes through the extra helper node $node_i^h$ after each single-server node $node_i$. Finally, the service time for jobs of different classes is as follows:

- the service times at segments corresponding to local computation is the corresponding execution time, for both helpers and non-helpers,
- the service time at a single-server node $node_i$ is $t_i^h$ for non-helpers and $t_i^h$ for helpers, and
- with $n\_contending(i)$ non-helper threads in the queue at node $node_i$, the service time at a helper node $node_i^h$ is $n\_contending(i) \cdot t_i^h$. In practice, we merge $node_i^h$ with the next infinite-server node. Note that $n\_contending(i)$ is unknown at the time we construct the queueing network.

C. Analyzing the Model for Contention

The performance metrics of interest are the lock contention time at each lock queue, which is the mean waiting time at
the lock queues. This can be directly obtained by solving the queueing network with standard methods (e.g., mean value analysis). Before doing that, we must obtain the value of \( n_{contending}(i) \) for each helper node \( node^h_i \). We obtain this value by a fixpoint approach, which aims to reach a stable point at which (i) \( n_{contending}(i) \) is equal to the average queue length at \( node^i \), and (ii) the average queue length at \( node^i \) is obtained after solving the network with service time \( n_{contending}(i) \) times \( t^{nh}_i \) for each helper node \( node^h_i \). Such a fixpoint can be obtained iteratively, e.g., by initializing \( n_{contending}(i) \) to 0 for all \( node^i \). By solving the queueing network, a value for \( n_{contending}(i) \) is obtained, which is used to obtain a new value of the service time at helper nodes \( node^h_i \), which is used in the next iteration, and so on until convergence. We note that the fixpoint is stable, since in our queueing network model there is a negative correlation between the service times at helper nodes and the queue lengths at lock nodes. Intuitively, if the service time at a helper node is increased, then fewer jobs will compete for lock accesses and (as we see below) also have lower delegation overhead, which leads to shorter average queue lengths.

**Example:** We illustrate the model with a multi-threaded program accessing two data structures (e.g., a FIFO queue and a pairing heap) in a cyclic pattern. Each data structure has a lock segment with a series of operations (e.g., enqueue and dequeue for the FIFO queue). Let use denote the lock segments as \( cseg_{fifo} \) and \( cseg_{heap} \) respectively. There is a local computation segment between each pair of lock segments.

From profiling this program, we can observe the lock access pattern as follows. After accessing \( cseg_{fifo} \), a thread has \( p \) probability of accessing a parallel segment then \( cseg_{fifo} \) again; and \((1-p)\) probability of accessing another parallel segment followed by \( cseg_{heap} \). Similarly, after accessing \( cseg_{heap} \), a thread has \( q \) probability of accessing a parallel segment followed by \( cseg_{fifo} \) and \((1-q)\) probability of accessing \( cseg_{heap} \). The lock behavior of this example program can be described by the queueing network in Figure 5. Each lock segment is modeled as a single-server node. There are four infinite-server nodes representing the parallel segment between each pair of critical sections. Note that the helper nodes in Figure 4 are merged with the next infinite-server nodes.

**D. Introducing Lock Overhead**

In order to obtain a faithful performance model of locking behavior, the overhead associated with lock operations must be introduced into the models. In the case of QD locks, different overheads are associated to helpers and non-helpers, as can be seen from Figure 3. For the helper class, the execution time of the critical section should be increased with the lock overhead (LO in Figure 3). For the non-helper class, the service time should be increased to also include the delegation overhead (DO). Note that the delegation overhead increases with the number of non-helper threads currently at the lock, i.e., at \( node^i \), the delegation overhead depends on \( n_{contending}(i) \). Section IV-F discusses measuring this overhead. Note that we do not include the time needed to propagate the notification that the critical section completed, from the helper to the non-helper. To integrate this into the model presented in section IV-B, the service times at single-server \( node^i \) should account for the overhead: \( t^{nh}_i + \text{DO}(n_{contending}(i)) \) for non-helpers and \( t^h_i + \text{LO} \) for helpers.

**E. Limitations of our model**

Due to our choice of queueing networks as our analytic model, there are a few limitations.

- The total number of threads is constant during the program execution.
- The program execution is sufficiently repetitive to achieve a steady state in which the lock access patterns are stable.
- The service times are exponentially distributed, which makes it easy to compute the performance metrics of the queueing networks. In practice, this over-estimates the lock contention.

**F. Measure the delegation overhead:**

To measure the delegation overhead \( \text{DO}(k) \) with \( k \) contending threads, we conduct an experiment with \( k \) non-helpers and one helper sharing a lock. By constructing a scenario where all \( k \) non-helpers delegate their critical sections, we measure the average time for each non-helper to delegation. In this experiment, all \( k + 1 \) threads execute a loop delegating empty critical sections. The measured average time per loop is then used as the delegation overhead \( \text{DO}(k) \). When there is only one helper thread and no non-helper thread, the measured overhead is \( \text{LO} \), which is measured to be 172 ns (averaged over 100 runs of a microbenchmark that executes 1 million critical sections). Figure 6 shows this \( \text{DO}(k) \) with \( k = 0 \) to 31.

**V. Profiling to obtain model parameters**

The queueing network requires as input both the execution time at each code segment and a routing probability between each pair of code segments. By profiling a single-threaded Pthread mutex run, one can obtain the execution time at each code segment and the routing probabilities between code segments. Then we combine them with the measured lock overhead and delegation overhead to obtain the inputs to our queueing networks model.
During the profile, we keep track of

1) the order of accessing lock code segments (including how many time each lock segment \( cseg_i \) is followed by a lock segment \( cseg_j \))

2) For each lock code segment \( cseg_i \), the time stamp attempting to take the lock \( t_{\text{attempt}}(i) \), acquiring the lock \( t_{\text{acquire}}(i) \) and releasing the lock \( t_{\text{release}}(i) \).

The time executing the critical section at \( cseg_i \) can be calculated as \( t^i_h = t^i_h = t_{\text{release}}(i) - t_{\text{acquire}}(i) \). The execution time of the local computation segment between two lock segments \( cseg_i \) and \( cseg_j \) is then \( t_{\text{attempt}}(j) - t_{\text{release}}(i) \).

The routing matrix is generated using the information recorded about order and quantity of accesses to lock code segments. We denote the (single-server) nodes corresponding to lock code segment \( cseg_k \) as \( node_k \), and the (infinite-server) nodes corresponding to the local computation segment between \( cseg_i \) and \( cseg_j \) as \( node_{i,j} \). To compute the routing probabilities between these nodes, we note that \( node_{i,j} \) is always followed by \( node_j \). The probability of going from \( node_i \) to \( node_{i,j} \) can be found by looking at all nodes that follow \( node_i \). Let \( F \) be the set of code segments \( cseg_j \) that follow some \( cseg_i \). Then we calculate the probability of going from \( cseg_i \) to \( cseg_j \) as \( r_{i,j} = \frac{n_{i,j}}{\sum_{f \in F} n_{i,f}} \). The routing probabilities then are defined as \( R(node_{i,j}, node_j) = 1 \) and \( R(node_i, node_{i,j}) = r_{i,j} \).

VI. EVALUATION

To evaluate our model, we need benchmarks accessing locks with potential lock contention. To thoroughly evaluate our model, the benchmarks should have varying lock access pattern, critical section time and local computation time. We designed a synthetic meta-benchmark where these three parameters are configurable.

There are two data structures in the benchmark: a FIFO queue and a pairing heap. Each data structure has operations protected by a critical section. The FIFO queue allows enqueue and dequeue operations. The pairing heap allows insertion and removal of the minimum valued node. Each benchmark accesses the critical sections of the two data structures and performs local computation in between. We can configure the lock access pattern by specifying the probability of using one code segment after using another one. For example, Figure 5 shows the different code segments (as nodes in a queueing network). These probabilities are defined as the routing matrix of the queueing network.

We need two versions of each benchmark: one implemented with Pthread mutex locks and one with QD locks. The evaluation process goes as follows: first we extract the input to our model by profiling a single-threaded application as described previously in Section V. With the extracted input, we build a queueing network model. Each lock’s predicted lock contention is then obtained by solving the queueing network. Last, we measure the delegation time (time including both the critical section and contention) with the QD lock. Our measurement framework takes two time stamps before and after the delegation. Taking each time stamp introduces about 50 – 100 ns overhead, which may cause inaccuracy in measurement, especially for small critical sections. By comparing it with the predicted lock contention, we calculate the relative error rate of our model.

We choose to design our own synthetic benchmarks instead of using existing benchmark suites such as PARSEC [1] or SPEC [9] for two reasons. First, it is non-trivial to port a benchmark to QD lock. Based on our experience of porting one benchmark (streamcluster) to using QD lock, which takes a week, it could take a few months to port the whole PARSEC benchmark suite to QD lock. It is hard to predict the effort to port an existing benchmark to use QD locking. Second, to thoroughly evaluate our model, we should ideally be able to configure the program parameters (e.g., length of lock segment and local computation segment). It is difficult to reconfigure the existing benchmarks to suit our evaluation needs. Using synthetic benchmarks provides the flexibility of changing program parameters with a reasonable porting effort.

A. Experiment setting

All experiments are conducted on a 32-core (4 sockets with 8 cores on each socket) server with Sandybridge architecture with Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz processors. The benchmarks are compiled with clang++ 3.4. The queue delegation library implementation is retrieved from its Github repository [11]. To minimize context switch overhead, threads are pinned to cores (one thread per core) and one socket is filled up first before utilizing the next available socket. The critical section segments of the benchmarks vary from 200 – 100000 nanoseconds and the local computation segments vary from 100 – 300000 nanoseconds. Twenty randomly generated routing matrices are used to define the access patterns between the code segments.

B. Results of the synthetic benchmarks

Figure 7 shows the evaluation result of 15 randomly selected runs with varying critical section segment (cs in the title of each subgraph, the unit is in nanoseconds), local computation segment (local) and routing matrix (matrix). In each subgraph, the x-axis is the number of threads and the y-axis shows the measured vs. modeled time of QD lock contention for both locks.
Fig. 7: Meta-benchmark evaluation result
When the critical section is above 1000 ns, our model has an average relative error of 14% among all 1950 configurations of the meta-benchmark. When the critical section is smaller than 1000 ns (Figure 7(a), Figure 7(b) and Figure 7(c)), our model is less accurate (30–40% relative error). This is due to the fact that our framework for measuring the lock contention has a relatively high overhead compared to the critical section. Taking each timestamp is 50 – 100 ns, which is on the same magnitude as the critical section size. The average relative error over all 3500 configurations is 24%.

When a lock is not contented, our model has a tendency to underestimate the lock contention (see lock 2 in Figure 7(a), Figure 7(b), Figure 7(c), Figure 7(h) and Figure 7(m)). This is due to the cache coherence misses caused by the critical section data moving from one core to another. When a lock is not contented, most threads are not being helped by the helper thread. They execute their own critical sections and therefore cause the critical sections to move among cores, causing the cache coherence misses and increasing the time to execute critical sections. This effect is not captured in our model.

VII. A CASE STUDY

As a case study, we analyze the lock performance of the kyotocabinet version 1.2.76 kccachetest wicked benchmark [14] with 100,000 iterations. It heavily exercises an in-memory database with the intent to both test its correctness and measure its performance. It spawns a number of worker threads which access the database using randomized operations, many of which have to protect the database using a lock. To reduce contention, it uses Pthreads reader-writer locks. This causes problems with profiling the expected execution time of lock accesses, as a single-thread run does not record information about the extent to which readers can share a lock. Additionally, contention may develop very differently when increasing the amount of threads waiting to read-lock or to write-lock.

An investigation about a stochastic model that estimates lock sharing and predicts the impact accordingly is beyond the scope of this work. Therefore, the original benchmark was modified to use Pthread mutex instead of readers-writer locks. Consequently, a 16-element lock array for fine-grained locking when holding the main lock in read mode became obsolete and was removed. This patch causes the benchmark to slow down from 6.76 seconds to 12.45 seconds when using 32 threads. As readers-writer locks aim to reduce lock contention by allowing read-only operations to access in parallel, it is expected that the lock contention problem is also higher.

The Pthreads mutex lock was then manually ported to QD locks, reusing porting efforts by the QD lock’s authors [12]. It has a speedup of 1.83 compared to the Pthread mutex version, reducing the runtime to 6.81 seconds, which is nearly the same as with Pthreads readers-writer locks. In earlier work MR-QD locks (QD locks with support for multiple parallel readers) have been compared to Pthreads readers-writer locks on the kyotocabinet benchmark [12]. This comparison has shown MR-QD locks have a speedup of 1.56 over Pthreads readers-writer locks, which is slightly lower than the speedup we observe without parallel readers. As the Pthread mutex version of the benchmark increases the contention this is to be expected, yet the trends seem to be similar.

Let us now have a look at the details of this example application. There are two locks in the benchmark: The method lock (mlock) which protects the data in the database, while the file lock (flock) is only used to manage the cursors that access the database. We first measure the amount of lock contention induced by the current Pthread mutex implementation. While the file lock has very little contention, the mlock becomes a performance bottleneck with more threads. Figure 8 shows the measured lock contention of mlock. It increases super-linearly with an increasing number of threads. After identifying such a performance bottleneck, we investigate if using QD lock could reduce the contention. To do so, we first profiled a single-threaded run of the kyotocabinet benchmark. This collects the information about lock usage which is needed for building a model. The profile shows that after accessing the mlock, the application accesses the mlock again with 75% frequency and flock with 25% frequency. Plugging in the lock overhead and delegation overhead and using the model in section IV, we form a closed queueing network. By solving the queueing network, we forecast the lock contention that would result after adopting the QD lock implementation.

Figure 8 shows the predicted lock contention from 2 – 32 threads if we replace the mlock with a QD lock. It shows a potential significant reduction of 20+ times in lock contention. With such a potential performance improvement, we re-implemented the benchmark with QD locking. Figure 9 shows the comparison between the measured contention (averaged over 100 runs) and predicted lock contention. The average prediction error over 2 – 32 threads is 18%.

![Fig. 8: Measured Pthread mutex lock contention of kyotocabinet and predicted QD lock contention](image)

VIII. CONCLUSION

Lock contention can be a performance bottleneck for parallel applications on multi-core systems. While the widely used Pthread mutex locks are known to perform poorly in the high contention case, the problem can be mitigated by using other locking algorithms. QD locking is an efficient lock algorithm under high contention, but it does not have the same interface as the traditional lock()/unlock() interface.

In this paper, we present an analytic model to forecast the lock contention of a program if the current Pthread mutex would be replaced by QD locks. The information provided...
by the model can be used to estimate the performance of the alternative locking algorithm before adopting it. This gives the programmers insights into the potential performance gain and saves the effort to reimplement the whole program with QD lock in order to measure its performance. The input to our model can be obtained by profiling a single-threaded run of the target program. We evaluate our model by comparing the predicted and measured lock contention on a set of synthetic benchmarks. The relative error rate is 24% among all 3500 configurations of the evaluation benchmark, and 14% over all 1950 configurations with critical sections larger than 1000 nanoseconds. As a case study to demonstrate how our model can be used, we investigate into the kyotocabinet benchmark. By adopting the QD lock, the benchmark can reduce its lock contention by 20+ times, as it is predicted by our model. Our model can be made into a fully automated tool to forecast lock contention for parallel applications. It can also be used to guide optimization in a compiler in choosing the appropriate lock algorithm to reduce lock contention.

Fig. 9: Measured vs predicted QD lock contention of kyotocabinet

Figures and diagrams

REFERENCES