Shared Stack Implementation and Analysis Using Queue Delegation with Elimination (Final Project)

Esha Choukse (ec27876), Mike Thomson (mt29253)
Multicore Computing, Fall 2014

November 20, 2014

Abstract

Queue delegation and Elimination are two techniques that help us improve the performance of shared data structures, by reducing contention. We use both these concepts together to implement a shared stack and analyse the performance we get from it. We have used the C++ queue delegation locking libraries published by Klaftenegger [1], and implemented elimination on top of it. We compare the performance of our implementation against the basic queue delegation, and MonitorT [2].

1 Introduction

Shared data structures between threads tend to slow down the execution due to the inherent sequentiality associated with them. A lot of implementations still use locks. Locks, because of their mutual exclusivity, make the execution of the critical section completely sequential. Several techniques are used to alleviate this disadvantage. One of these techniques is queue delegation. Queue delegation allows a thread to offload its task to a delegate thread, which is already operating on the shared structure. This allows the thread to carry on with its execution, till the delegate thread executes the queued task. Another technique is to use elimination. Elimination tries to take advantage of operations on the shared data structure, which have reverse semantics, for example a push and a pop on a stack. This way, none of the two threads actually access the shared data structure, resulting in less contention.

We use both these concepts together to implement a shared stack and analyse the performance we get from it. We have used the C++ libraries published by Klaftenegger [1], and implemented elimination on top of it. We compare the performance of our implementation against the basic queue delegation, and MonitorT [2].

2 Background

Let us first discuss the concepts of queue delegation and elimination in detail.

2.1 Queue Delegation

Using locks on shared data structure is a very common practice. In doing so, each operation on the shared data structure by a thread, needs it to first acquire the associated lock. Even though fine grained locks perform better than coarse grained locks, there is still a lot of performance hit. Queue delegation is a method to reduce the hit by allowing threads to outsource their shared data structure operations to a delegate thread. This is done by pushing the task into a delegation queue. Once the task is in queue, the thread can carry on with its execution and does not need to wait on the result.

The delegate thread reads tasks from the delegation queue, and executes them one by one, giving out the results to the associated threads. Note that since a single thread is executing the operations on the shared data structure, no locks are required.

There are several ways to implement the Queue delegation. One of the main decisions to make is that whether we want a single thread to remain the delegate thread throughout the execution, or can any thread become a delegate thread. In the QD Lock libraries we have used, the second approach is implemented. The threads, upon reaching a shared data structure operation, try to acquire a lock, to become the delegate thread. If they fail to acquire the lock, they put the task into the delegation queue and leave. Instead, if they do acquire the lock, they execute their task, and all the tasks in the delegation queue, till the queue becomes empty eventually. Once this happens, the delegate thread gives up the lock and leaves. Upon contention, a thread may need to become the delegate thread for very long periods, affecting execution of that thread. To fix this, we have a limit on the number of tasks in the delegation queue, which we call delegation queue length. Once this limit is reached, the delegation queue stops accepting tasks. The choice of the delegation queue length is therefore an important factor in the performance.

2.2 Elimination

Elimination identifies the operations on a shared data structure, which have reverse semantics. Such operations can be made to collide outside the shared data structure, and return, without ever getting to access the actual structure. This helps reduce contention.

The elimination uses an elimination array, which has multiple slots. Each slot can be in a waiting, busy or empty state. An operation, on finding an empty slot, changes it to waiting and waits. Another operation comes in, and on finding the slot waiting, makes it busy. In case these two

operations are of reverse semantics, our work is done. We then return both the operations with the expected results. The choice of number of slots in the elimination array is an important one, since too big an array will lead to fewer collisions.

3 Implementation

We started by using the examples provided by the QD Lock libraries. We modified the critical section functions to emulate a stack and its corresponding operations, pop and push.

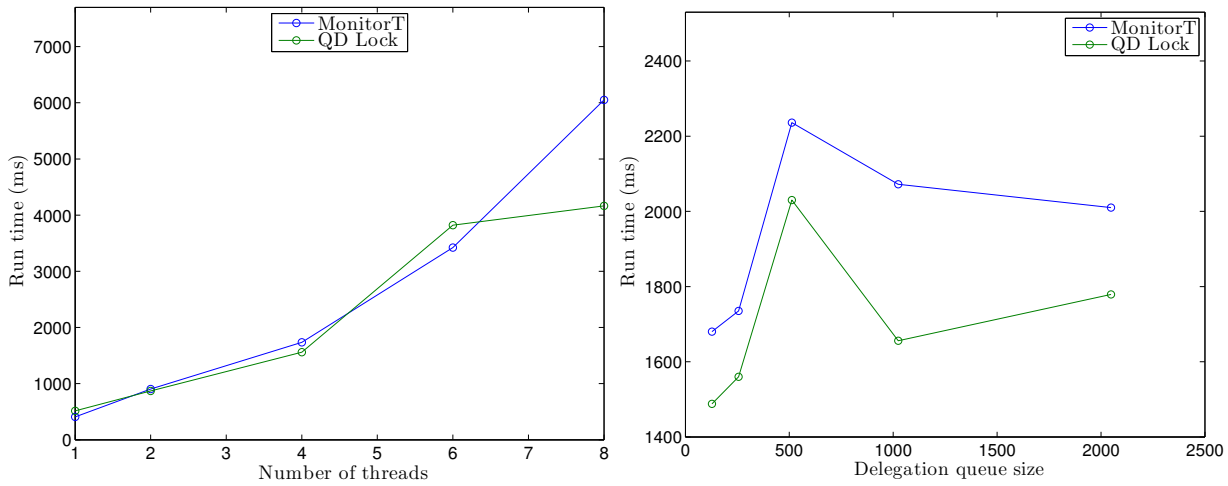
Next, we implemented the elimination array and exchanger. We modified the QD Lock libraries, so that in case the thread is not able to become the delegate thread, nor insert its task into the delegation queue (because of the queue being full), it tries the elimination array instead. The elimination array size is an important factor in the performance, since we need to keep a balance between contention and match-timeouts. Match-timeouts happen when a thread waits in the elimination array for too long, without getting a partner. It then times out and tries to get the delegation lock again.

We encountered several issues with the C++ code of the libraries, which hugely uses templates. Templates restrict the usage of some functions and parameters and it was difficult to get past these barriers. It was necessary to match the future/promise assignments as expected by the threads to guarantee correct execution.

Once we had the implementation ready, we pulled out all the affecting parameters out as command line arguments and profiled the behaviour of the implementation (to automate the data collection). Lastly, we implemented a stack using MonitorT and collected and analyzed the performance of both these implementations.

The computer used to perform run the benchmarks has an 8-threaded processor (4 cores with 2 threads per core). The ranges/values of various parameters we used are:

- Number of operations : 1000000
- Number of threads: 1, 2, 4, 6, 8
- Queue Delegation size: 128, 256, 512, 1024, 2048
- Elimination Array size: 2, 4, 8, 16
- Percentage of Pushes: 50, 60, 70, 80, 90



(a) (b)

Figure 1: Baseline comparison of the MonitorT implementation (Java) and the QD Lock (C++), without elimination for either implementation: (a) varying the number of threads (queue size = 256, push ratio = 0.5); (b) varying the delegation queue size (threads = 4, push ratio = 0.5).

4 Results / Analysis

We recognize that the Java and C++ queue delegation implementations will have inherent performance differences. The purpose of the analysis in this section is to determine a baseline performance difference between MonitorT (Java) and QD Lock (C++), where neither have elimination. Then, we analyze the performance difference by adding elimination to the QD Lock version (i.e., QD Elimination Lock).

Figure 1 shows the baseline performance comparison between MonitorT and QD Lock (neither having stack elimination). Figure 1a shows that the two implementations are relatively similar, with the C++ implementation slightly outperforming MonitorT, as would be expected. It is interesting to note that they both seem to scale equally well with increasing thread counts, with the exception of 8 threads (which can be attributed to the contention inherent with running 8 threads on an 8-threaded machine). However, Figure 1b shows that their performance is relatively volatile when the delegation queue size varies. As the queue size increases, both implementations tend to have longer run times.

In Figure 2, we see that the performance gap between MonitorT and QD Elimination Lock is greater than the gap between MonitorT and QD Lock in Figure 1a. This expected behavior also confirms that the performance of QD Elimination Lock is better than QD Lock. The performance benefits are greater at higher thread counts. After this initial comparison of MonitorT, QD Lock and QD Elimination Lock, more focus can be given to analyzing the performance benefits of moving from QD Lock to QD Elimination Lock.

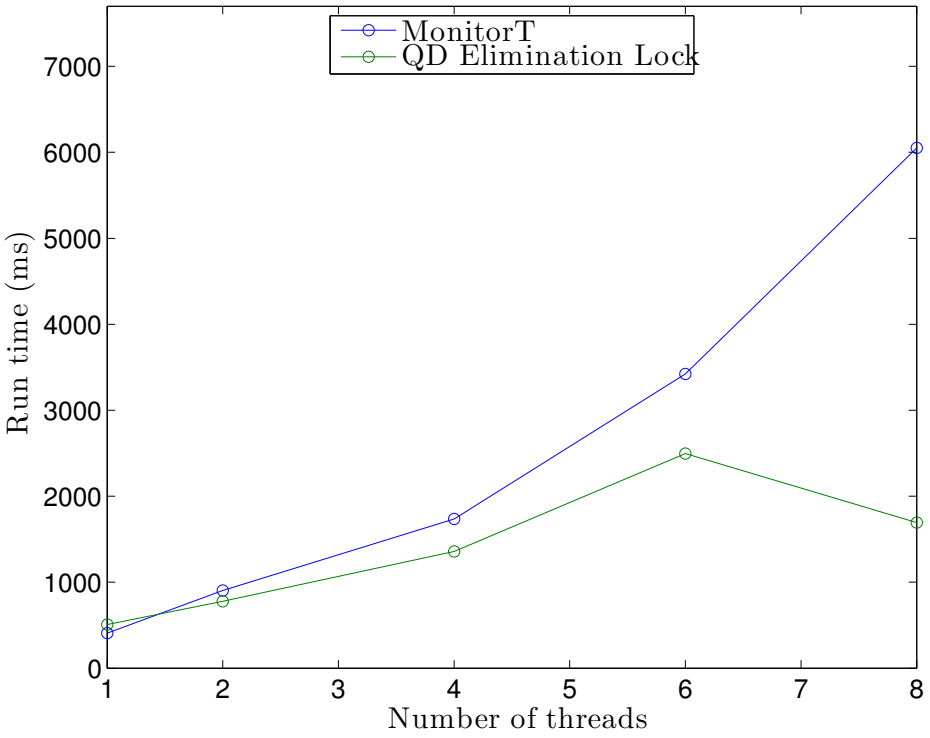


Figure 2: Comparison of the MonitorT implementation (Java) and the QD Elimination Lock (C++), varying the number of threads (elimination included for C++ implementation, not for Java). Queue size = 256, elimination array size = 4, push ratio = 0.5.

Figure 3 shows that QD Elimination Lock consistently outperforms QD Lock, across sweeps of all parameters. Additional insights can also be gained from these figures. Figure 3a shows that as the number of threads increases, the speedup of QD Elimination Lock over QD Lock increases. Figure 3b shows that choosing the correct elimination array size can be very important. The elimination array size should be neither too small nor too large, as this yields either too much contention (shown where array size = 2) or too few matches between pushes and pops (shown where array size = 16). In Figure 3c, as the delegation queue size increases, the delegate thread spends more time executing the other threads' workloads, eventually slowing the execution of the delegate thread's own work. This is seen for both the QD Lock and QD Elimination Lock. This behavior would not be expected in MonitorT, since MonitorT uses a separate thread for the delegation queue. In Figure 3d, as the push percentage increases, we would have expected the performance to decrease since there would be fewer matches between pushes and pops. But instead we get the unexpected result that the performance either increases or remains steady. Another anomaly is that we would expect QD Lock to have flat performance across all push percentages, since it views all operations alike.

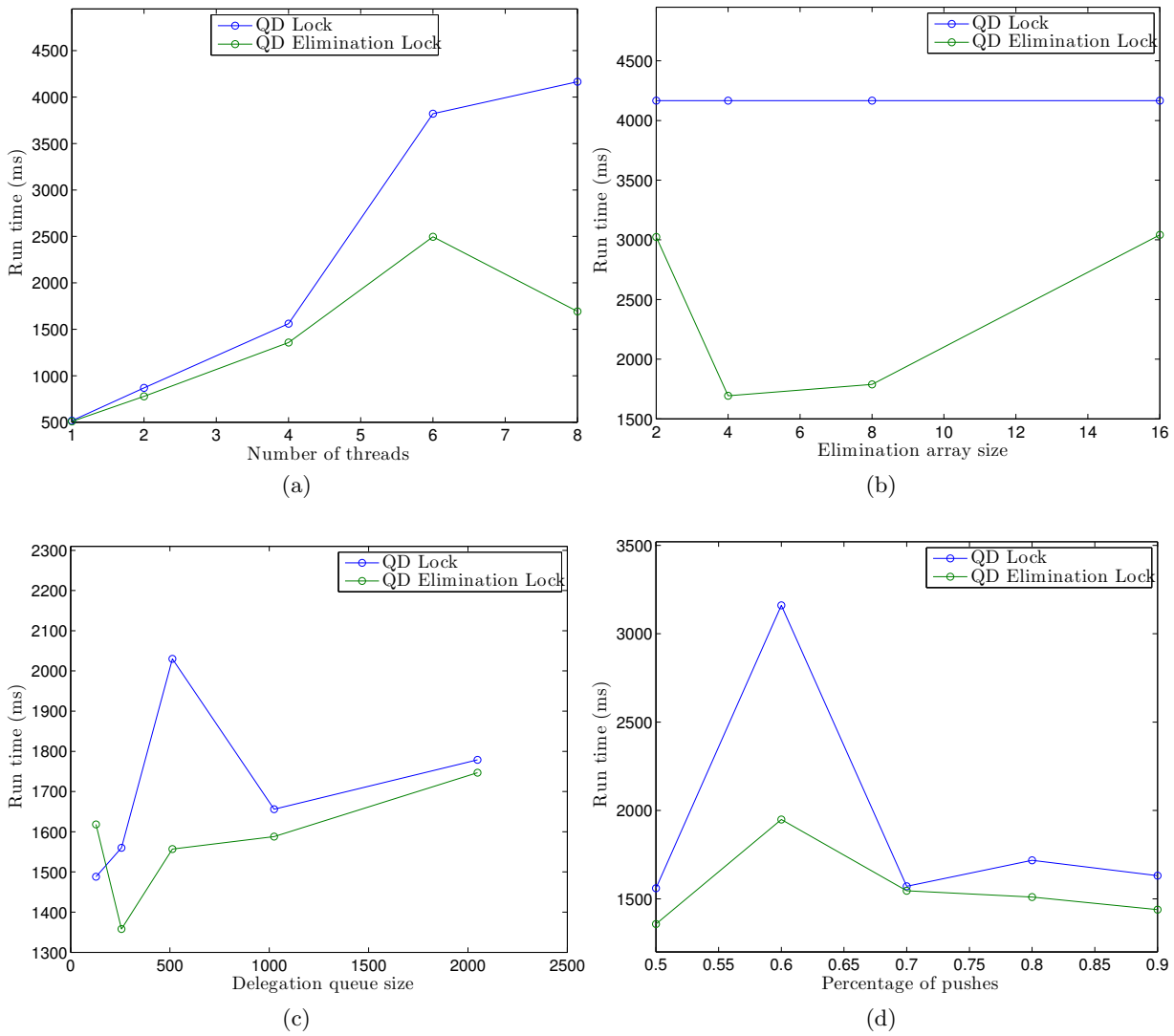


Figure 3: QD Elimination Lock versus QD Lock (both C++), while varying each parameter separately: (a) varying the number of threads (elimination array size = 4, queue size = 256, push ratio = 0.5); (b) varying the elimination array size (threads = 4, queue size = 256, push ratio = 0.5); (c) varying the delegation queue size (threads = 4, elimination array size = 4, push ratio = 0.5); (d) varying the push ratio (threads = 4, elimination array size = 4, queue size = 256).

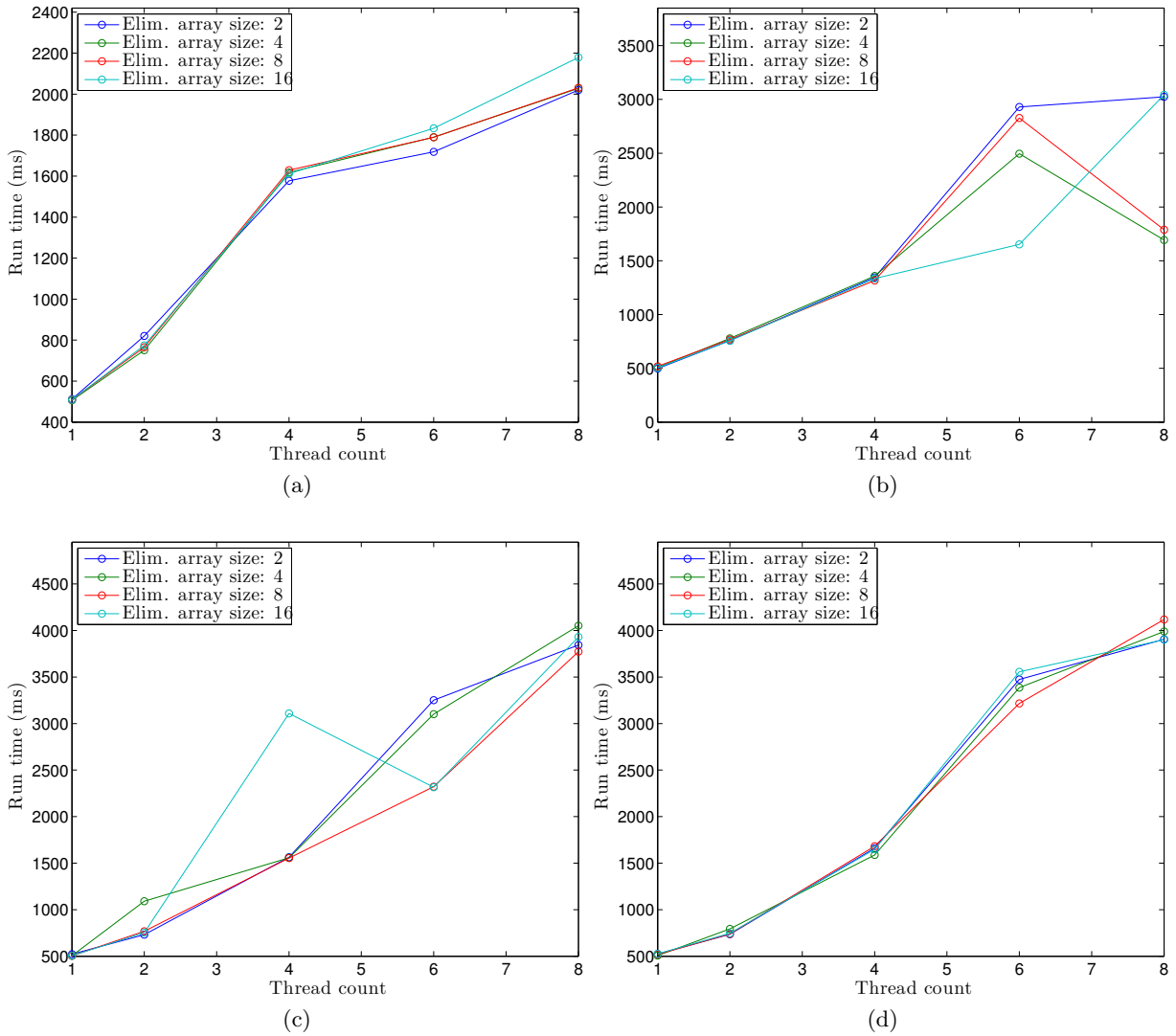
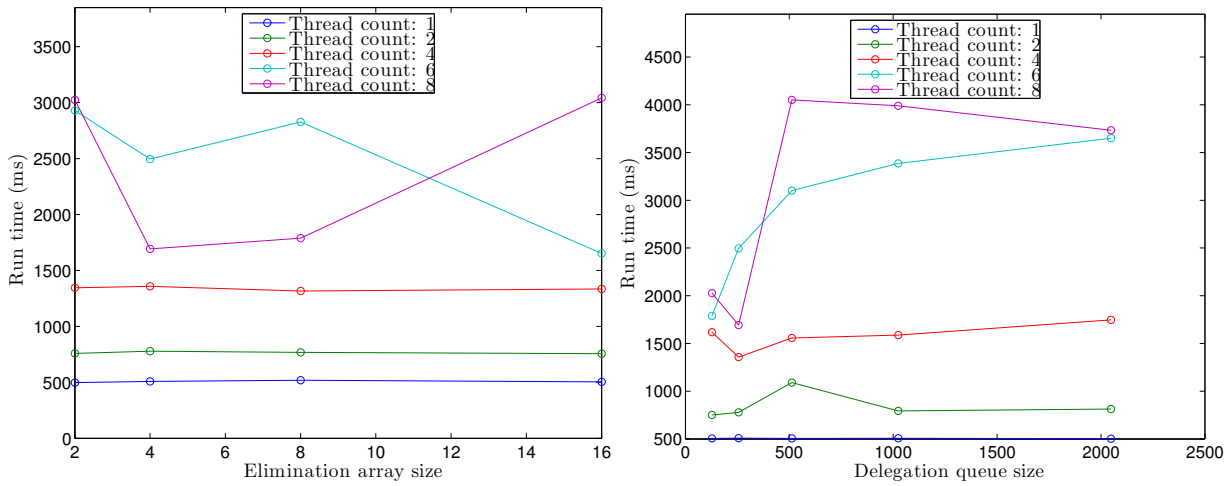


Figure 4: QD Elimination Lock performance versus thread count and elimination array size - each graph represents a different delegation queue size, while multiple elimination array sizes are plotted in each graph (with push ratio = 0.5): (a) queue size = 128; (b) queue size = 256; (c) queue size = 512; (d) queue size = 1024.

The performance values observed in Figure 4 show the relationships between thread count, delegation queue size and elimination array size. Figures 4a – 4d represent changing the delegation queue size between 128, 256, 512 and 1024, respectively. The different queue sizes appear to have a negligible effect on the run time, except for the first case where the performance is better with a queue size of 128. These figures also show that the thread count had the largest effect and the elimination array size had minimal effect, since the multiple lines plotted for run time versus thread count exhibit minimal changes for the different elimination array sizes.



(a) (b)

Figure 5: (a) QD Elimination Lock performance versus elimination array size and thread count (queue size = 256, push ratio = 0.5). (b) QD Elimination Lock performance versus delegation queue size and thread count (elimination array size = 4, push ratio = 0.5).

The elimination array size's minimal effect on performance is reflected again in Figure 5a, where the performance for each thread count remains relatively flat across all elimination array sizes. This plot also shows how increasing the number of threads monotonically decreases the performance (except for 8 threads), establishing the effects of contention at the higher thread counts. The difference in behavior for the line depicting 8 threads is likely due to contention from running 8 threads on a 8-threaded machine. Figure 5b shows that initially, increasing the delegation queue size adversely affects performance. However, for delegation queue sizes above 1024, the performance levels out.

5 Conclusion

We have seen that implementing elimination for a shared stack with queue delegation overall gives better performance when compared with a simple non-elimination shared stack with queue delegation. We have seen the effect on performance due to delegation queue size, elimination array size, number of threads and percentage of push operations. Apart from a few data anomalies as discussed in the Results / Analysis section, the rest of the results matched our expectations.

Several areas of this project can be further explored in the future. First, we have found that the implementation style of the QD Lock libraries was very disruptive to updating for elimination. Future work can focus on making the libraries more amenable to this type of customization, so that other features can also be included (such as an efficient elimination array range policy). Second, further experiments could be performed with MonitorT if it is updated to include elimination. This would allow more comparison with the QD Elimination Lock implementation. Last, the data

collected here represents a broad, coarse spectrum of configurations. More work can be done to focus on the best-performing configurations, leading to exploration of better and more fine-grained configurations.

References

[1] D. Klaftenegger, K. Sagonas, and K. Winblad, "Delegation locking libraries for improved performance of multithreaded programs," in *Euro-Par 2014 Parallel Processing*, vol. 8632 of *Lecture Notes in Computer Science*, pp. 572–583, Springer International Publishing, 2014.

[2] W.-L. Hung, H. Chauhan, and V. K. Garg, "ActiveMonitor: Non-blocking monitor executions for increased parallelism," *arXiv preprint arXiv:1408.0818*, 2014.