Usage-Fairness in Delegation-Styled Locks

Hongtao Zhang

Advisor: Remzi Arpaci-Dusseau

Abstract

This proposal outlines a plan to investigate the efficacy of delegation-styled lock and the combination with the idea of delegation and usage-fairness. Previous studys has shown the problem of scheduler subversion when locks only adapt fairness in acquisition-level, in which most delegation-styled lock provides. They propose that lock usage (the amount of time holding the lock) should be viewed as resource as cpu time slice and fairness gurantee need to be provided. We propose to modify the state of art combining locks (*Flat-Combining*, *CC-Synch*, *H-Synch*, *DSM-Synch*) and client-sever locks (*RCL*, *ffwd*) to adapt usage-fairness principle with a simple strategy, "banning" to ensure the proporsional share of lock usage among threads. Beyond that, delegation-styled locks where combiner is elected on the fly sacrifice the response time of the combiner, as it is doing volunteer work for other threads. We propose to employ stochastic methods to ensure the volunteer work proporsional to lock usage. Beyond that, we plan to redesigned combining strategy that are native with usage-fairness principle based on a concurrent MPSC Priority Queue. The performance benchmark of these locks will be performed in various *Concurrent Object* (Fetch&Add, Fetch&Multiply, Queue, PriorityQueue, etc.) with response time analysis.

198

1 Introduction

In modern computer industry, the focus of improving of the Central Process Unit (CPU) has shifted from increasing the clock frequency to increasing the number of cores. This shift has led to the development of multi-core processors, which are now widely used in many computer systems. The scalability of applications on these multi-core machines are captured by *Amdahl's Law*, stating that the maximum speedup of a program is limited by the fraction of the program that cannot be parallelized.

One of the most common place where parallelization is hard is when different threads are communicating with each other via shared data. One of the most common strategy used to synchronize concurrent programs is the use of locks, which provides the gurantee of mutual exclusion [1, 6]. Since synchronization must be executed in mutual exclusion, their execution becomes a hot spot in various concurrent environment [3]. Ideally, the time to execute the same number of synchronization should be the same regardless the number of threads. However, in practice, threads that are contenting for a lock can drastically impact the performance of the system.

Delegation-styled lock is a new class of locks that aims to reduce the contention and data movement on the lock by delegating the work to one threads. In this technique, instead of having all threads to

compete for the lock, each threads will wrap their critical section in a request and send it to a combiner. The combiner will then execute the request and return the result to the threads. This technique has been shown to outperform traditional locks in various circumstances. There are two main classes of delegation-styled locks: *combining* synchronization [3–5] and *client-server* synchronization [7, 9]. The former each participant acts as combiner temporarily, while the latter has a fixed thread as combiner. Specifically, resent work has demonstrate the potential of employing a delegation-styled lock with traditional lock api, which open the potential of using delegation lock in large system that is hard to modify [4].

However, earlier study has introduce the problem of scheduler subversion when locks do not adapt fairness or only in acquisition-level, in which most delegation-styled lock provides [8]. When thread has inbalanced workload in their critical sections, the present of lock will subvert the scheduling policy of CPU provided by the operating system, in which both threads should have the same amount of CPU time. This problem is crucial when some threads require low latency with minimal access to the lock, such as a thread handling user interactive work, while others requires large computation with the data, while the competion of the lock will subvert the original goal of CPU scheduler ensuring low latency for short work. Furthermore, this issue is more severe in the delegation-styled lock where thread will temporarily act as combiner. For example, if the thread handling the interactive work is elected as combiner, user may experience serious latency issue, which causes combining lock less attractive in the system.

To remedy this problem, we propose to adapt a similar strategy employed in the original paper, "banning" to ensure lock usage fairness for these delegation-style locks. Furthermore, we plan to design our own combining strategy that are native with usage-fairness principle based on a concurrent MPSC Priority Queue. We will also employ stochastic methods to share the combining evenly.

2 Method

All the experiment will be performed on Cloudlab, a large-scale testbed for cloud research that provides researchers with control and visibility all the way down to the machine.

The implementation of these locks will be programmed in *rust*, a modern systems programming language that is designed to be safe, concurrent, and performant. The performance benchmark of these locks will be performed in various *Concurrent Object* (Fetch&Add, Fetch&Multiply, Queue, Pri-

orityQueue, etc.) with response time analysis. The performance of these locks will be compared with the original implementation and the state of art locks.

Datas will be stored in Apache Arrow format, and the analysis will be done in Julia with the help of DataFrames.jl and Arrow.jl, where plots will be drawn with Makie [2] with Algebra Of Graphics.

2.1 Performance Counter

3 Timeline

The task of this project is majorly broken down into the following parts

1. Implementation of various delegation-styled locks in *rust*.
    1. *Flat-Combining* [5] (Done)
    2. *CC-Synch* [3] (Done)
    3. *H-Synch* [3] (March, 2024)
    4. *DSM-Synch* [3] (April, 2024)
    5. *RCL* [7] (Done)
    6. *ffwd* [9] (April, 2024)
2. Employing banning strategy to ensure usage-fairness principle in delegation-styled locks.
    1. *Flat-Combining* [5] (Done)
    2. *CC-Synch* [3] (Done)
    3. *H-Synch* [3] (April, 2024)
    4. *DSM-Synch* [3] (September, 2024)
    5. *RCL* [7] (Done)
    6. *ffwd* [9] (September, 2024)
3. Employing stochastic methods to ensure volunteer work is shared evenly (October, 2024)
4. Redesigning combining strategy that are native with usage-fairness principle based on a concurrent MPSC Priority Queue (October, 2024)
    1. Implementation based on existing concurrent MPSC Priority Queue (October)
    2. Design a new (probably relaxed) concurrent MPSC Priority Queue (November)
    3. Implementation of the new concurrent MPSC Priority Queue (November)
5. Implementation of benchmark suites
    1. Fetch&Add (Done)
    2. Fetch&Multiply (Done)

3. Queue (Feburary)
4. PriorityQueue (March)
5. Inbalance Workload Data Structure (September)

6. Performance analysis Suite
    1. Throughput/Scalability (Done)
    2. Fairness (March)
    3. Response Time (April)

7. Performance analysis (On the fly)

4 Conclusion

In conclusion, we propose to demonstrate that delegation-styled locks suffers from the scheduler subversion problem. To remedy this problem, we propose to integrate existing delegation-styled locks with "banning" strategy to ensure their usage fairness. Further we propose to employ stochastic methods to share the combining evenly.

In the future, we plan to resolve the response time issue of the combiner by swapping the non-critical work of combiner to one of the waiter that expected to wait long. One possible approach is to employ a similar strategy used in the TCL Lock [4], while another proposal is to embrace the asynchronous programming model provided by *rust* to delegate the `Future` for execution and create a custom runtime that adapts takes lock-usage.

Bibliography

[1] Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. 2018. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC

[2] Danisch, S. and Krumbiegel, J. 2021. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*. 6, 65 (2021), 3349

[3] Fatourou, P. and Kallimanis, N. D. 2012. Revisiting the combining synchronization technique. *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), 257–266

[4] Gupta, V. et al. 2023. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with {TCLOCKS}. *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), 1–16

[5] Hendler, D. et al. 2010. Flat combining and the synchronization-parallelism tradeoff. *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures* (2010), 355–364

[6] Herlihy, M. et al. 2020. *The art of multiprocessor programming*. Newnes

[7] Lozi, J.-P. et al. 2012. Remote core locking: Migrating {Critical-Section} execution to improve the performance of multithreaded applications. *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), 65–76

[8] Patel, Y. et al. 2020. Avoiding scheduler subversion using scheduler-cooperative locks. *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), 1–17

[9] Roghanchi, S. et al. 2017. Ffwd: Delegation is (much) faster than you think. *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), 342–358