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 at acquisition-level, in which most delegation-styled lock provides. They propose that lock usage (the amount of time holding the lock) should be vieId as resource as cpu time slice and fairness gurantee need to be provided. I 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. I propose to employ stochastic methods to ensure the volunteer work proporsional to lock usage. Beyond that, I 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 latency analysis.

197

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. HoIver, in practice, threads that are contenting for a lock can drastically decreases the performance [3, 6].

Delegation-styled lock is a new class of locks that aims to improve the performance of synchronization by reducing the contention and data movement on the lock by delegating the work to a single thread. 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 and even close to the ideal performance of sequential execution. There are two main classes of delegation-styled locks: *combining* synchronization [3–5] and *client-server* synchronization [7, 9]. *Combining* refers that each participant may act as combiner temporarily to combine the works and execute. The latter class has a fixed thread refered as a server to execute all critical sections sent by other thread trying to hold the lock. One of the criticision of delegation-styled lock is that they are hard to deploy in complicate application. HoIver, 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 [4].

However, very recent study has introduced the problem of scheduler subversion when locks do not adapt fairness or only at acquisition-level, in which most delegation-styled lock provides [8]. When thread has imbalanced 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. Imagine a scnerio when there's a thread handling interactive work with user and a bunch of batch threads that are handling background work. They uses a lock to synchronize their work. Without usage fairness, user may experience very long response time to acquire the lock, which subvert the CPU scheduling goal aiming to provide fast response time for interactive work. Furthermore, this issue is more severe in the delegation-styled lock where thread will temporarily act as combiner. If the interactive thread is elected as combiner, user may experience serious latency issue, which causes combining lock less attractive in the system that has imbalanced workload.

2 Method

2.1 Implementation

I propose to adapt a simple heuristic strategy employed in the SCLs (Scheduler Cooperative Locks), "banning" to ensure lock usage fairness for delegation-style locks [8]. Specifically, every threads will be banned with a heuristic algorithm based on their critical section length. Specifically each thread cannot reacquire the lock before $(\text{cs} \times \text{num\_threads} - \text{avg\_cs})$. This strategy will ensure that eventually, every threads will have evenly distributed lock usage.

Furthermore, I plan to design our own combining strategy that are native with usage-fairness principle based on a concurrent MPSC Priority Queue. This strategy is similar to how Linux kernel CFS

(Completely Fair Scheduler) works. The scheduler will select the next task that has used the minimal time slice. Delegation-styled locks are inherently suitable for employing existing scheduling policy as the combiner will be elected or designated, in which it can act as a scheduler to ensure the usage-fairness principle. By embracing this, I are able to construct a lock which provides both the fairness and the low latency given imbalanced work load.

2.2 Experiment

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 machine running the experiment is still under consideration.

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 conducted on commonly used *Concurrent Objects* (`Fetch&AddLoop`, `Fetch&Multiply`, `Queue`, `PriorityQueue`) with latency analysis. I plan to measure the throughput of various concurrent objects as the indicator of performance, and measure individual latency of each jobs to analyze the latency. Datas will be stored in Apache Arrow format, and the analysis will be done in Julia with DataFrames.jl and Arrow.jl, where plots will be drawn with Makie [2] with Algebra Of Graphics.

2.3 Benchmark Suite

The benchmark suite will be implemented in *rust* and will be open-sourced. `rdtscp`, a special instruction recording time stamp counter in x86_64 instruction set, is used to measure the time of each operation with minimal performance influence.

The benchmark suite will record the following execution data for a set of commonly used concurrent object: `cpu_id` (the cpuid the thread is running on), `thread_num` (the number of threads that is concurrently accessing the concurrent object), `operation_num` (the number of operations done), `latencies` (how long each operation takes to start), `self-handled` (whether operation is performed by the same thread), `hold_time` (the total time the thread is using the concurrent object), `combine_time` (the total time the thread is performing volunteering work), `noncs_length` (a time slice where the thread doesn't touch the concurrent object), `name` (the name of the concurrent object).[1]

[1]Some of the data will only valid for a particular sets of concurrent objects. `hold_time` will not be valid in lock-free concurrent objects. `combine_time` will only be valid in combining-styled locks. `self-handled` will only be valid for a delegation-styled lock.

2.4 Concurrent Object

I will measure the performance of the locks with the following widely used concurrent objects:

- `Fetch&AddLoop`: a simple synthetic workload that is used to demonstrate the performance and behavior of locks. When the loop limit is 1, it is equivalent to a `Fetch&Add` operation. We will measure with different loop limit to demonstrate the performance with varying workload.
- `Fetch&Multiply`: a short operation that no hardware offers direct atomic support.
- `Queue`: A widely used concurrent object that is used in many concurrent applications.
- `PriorityQueue`: A concurrent object that can be very important in scheduling.

2.5 Performance Metrics

I plan to measure the performance of the locks with the following performance metrics:

1. **Throughput**: The number of operations that can be performed in certain time slice. This metric shows the bulk performance of the concurrent object. For `Fetch&AddLoop`, this is equal to the number of times through each loop, while for other concurrent objects, this is equal to the number of operations performed.
2. **Latency**: The time it takes for each operation to start. This metric shows the response time of the concurrent object.
3. **Lock Usage Fairness**: This metric follows follows the original definition of *Lock Oppourtunity* and The Fairness Index capturing the idea of fairness of the lock among all threads [8].

We will measure the throughput of the locks by the number of operations that can be performed in certain time slice. The scalability will be demonstrated by the throughput of the locks with the increasing number of threads. The fairness analysis involves two part. One is the fairness of the lock usage, which is the amount of time each thread holds the lock (`hold_time`). The other is whether the concurrent object achive scheduler's goal for response time of each operations (`latencies`).

3 Timeline

Total Project Hours: 500 (300 for Spring and Summer 2024 and 200 for Fall 2024)

- Spring 2024:
  - Implementation of the delegation-styled lock that adapts usage-fairness principle based on banning strategy.
  - Basic Benchmark Suite for calculating the throughput, scalability, and fairness.
- Summer 2024: Analysis of the locks and additional state of art concurrent data structure to compare.

- Fall 2024: Implementation of the delegation-styled lock that adapts usage-fairness principle based on a concurrent MPSC Priority Queue.

4 Conclusion

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

In the future, I 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