Usage-Fairness in Delegation-Styled Locks

Hongtao Zhang

Advisor: Remzi Arpaci-Dusseau

Abstract

This proposal presents a comprehensive plan to explore the effectiveness of a novel delegation-styled locking mechanism that integrates the concepts of delegation and usage fairness. Prior research has identified challenges with scheduler subversion arising from locks that adapt strict acquisition-level fairness, which is common in state of art delegation-styled locks. Furthermore, some delegation locks dynamically elect a combiner from participants. This often compromise the combiner's latency as it introduces additional workload for the combiner. They suggest treating lock usage as a resource, akin to CPU time slices, warranting a usage-level fairness. I aim to enhance state-of-the-art combining locks (*Flat-Combining*, *CC-Synch*, *H-Synch*, and *DSM-Synch*) and client-server locks (*RCL* and *ffwd*) by incorporating a usage-fairness principle. The straightforward "banning" strategy will be implemented to ensure a proportional allocation of lock usage time across threads. A stochastic methods will be employed to proportionally distribute the voluntary workload based on lock usage. In addition, I plan to devise a new scheduling strategy inherently aligned with the usage-fairness principle by leveraging a concurrent relaxed Priority Queue. The efficacy of these enhanced locking mechanisms will be tested through micro-benchmarks on various commonly used Concurrent Objects complemented by an in-depth latency analysis.

1 Introduction

In the current landscape of computational technology, the focus to enhance Central Processing Unit (CPU) performance has transitioned from increasing clock speeds to multiplying core counts. This evolution has given rise to multi-core architectures, which have become ubiquitous across computer systems. The scalability of applications on such multi-core infrastructures is predicated on Amdahl's Law, which postulates that the theoretical maximum improvement achievable through parallelization is limited by the code that must remain sequential.

A principal challenge in parallel computing is thread coordination via shared resources. Lock-based synchronization mechanisms are widely employed to ensure mutual exclusion and are critical for threads to communicate accurately and reliably [2, 7]. These synchronization points, however, are often a source of contention and can become performance bottlenecks in a concurrent execution environment [4]. Theoretically, the synchronization duration should be invariant with respect to the number of threads; yet, contention for locks often leads to a serious degradation in performance that is disproportionate to the increase in thread count [4, 6, 7].

Delegation-styled locks have emerged as a innovative solution aimed at boosting synchronization efficiency by minimizing contention and the associated overhead of data movement. Instead of each

thread compete for a lock to execute their critical section, threads package their critical sections into requests and entrust them to a combiner, which processes these requests and returns the results. There are two predominant forms of delegation-styled locks: *combining* synchronization [4–6] and *client-server* synchronization [8, 10]. Combining locks allow for dynamic selection of the combiner role amongst the participants, whereas client-server locks dictates a consistent server thread to manage all requests. Empirical evidence suggests that this technique can outperform traditional locking mechanisms, even approaching the ideal of sequential execution efficiency regardless of number of threads.

Newly conducted studies have introduced concerns regarding scheduler subversion when locks are implemented without a sophisticated fairness mechanism or are limited to fairness at the point of acquisition [9]. This is particularly problematic when threads exhibit imbalanced workloads within their critical sections, as the presence of a lock can disrupt the CPU's scheduling policy, which intends to allocate equitable processing time to concurrent threads. Envision a scenario where interactive threads engaging with users are in contention with batch threads performing background tasks, all synchronized by a lock. In the absense of principle of usage fairness, the interactive threads may suffer from inordinate delays in lock acquisition, thereby subverting the CPU scheduler's objective of ensuring prompt response times for interactive tasks. Moreover, the issue is magnified in the context of delegation-styled locks, where the elected combiner thread may be burdened with an unequal share of work. If an interactive thread is chosen as the combiner, it could lead to severe latency issues for the user, thus diminishing the attractiveness of combining locks in systems with disparate workloads.

To remedy this problem, I propose to integrate existing delegation-styled lock with the concept of *usage-fairness* by employing the *banning* strategy [9]. By restricting access to the lock according to their usage, we can prevent any single thread from monopolizing CPU resources, thus upholding the principle of equitable computational opportunity amongst concurrent processes.

2 Method

2.1 Implementation

I propose a simple heuristic strategy, *banning*, inspired by SCLs to remedy the unfairness of delegation-styled locks, by restricting thread that recently enter the lock to reenter the lock [9]. Specifically, every threads will be banned with a heuristic algorithm based on their critical section length. Formally, a thread is banned from reacquiring the lock for a duration calculated by the expression:

$n_{\text{thread}} \times cs - cs_{\text{avg}}$, where cs refers to the critical section length of the thread and cs_{avg} is the average length of critical section accross threads that trying to acquire the lock. This methodology promises an equitable distribution of lock usage among threads over time given the assumption that all threads are actively contenting for the lock.

To relax the assumption, I propose to develop a bespoke combining strategy that adheres to the usage-fairness principle analogous to the CFS (Completely Fair Scheduler) employed in the Linux kernel. The combiner will prioritize tasks that have consumed the least amount of lock usage, much like the CFS selects tasks with the minimum time slice used. The strategy's foundation will be the integration of sophisticated concurrent priority queues and their relaxed variants [1, 12].

2.2 Experiment

Experiments will be conducted on Cloudlab, a sophisticated cloud research testbed that provides comprehensive control and visibility down to the bare metal. The implementation of these locks will be developed in *rust*, a modern system programming language known for its safety, concurrency, and performance. The performance benchmark of these locks will be conducted on commonly used *Concurrent Objects*. The key performance metric will include throughput, latencies, and fairness. 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 [3] with Algebra Of Graphics.

2.2.1 Benchmark Suite

The benchmark suite will be implemented in *rust* and will be open-sourced. *rdtscp()*, a special time stamp counter in x86_64 instruction set, is used to measure the time for micro-benchmark. The benchmark suite will record the following execution data for a set of commonly used concurrent object: `thread_num` (the number of threads that is concurrently accessing the concurrent object), `operation_num`, `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).[1]

2.2.2 Concurrent Object

I aim to evaluate the efficacy of the proposed locks using the following concurrent objects:

[1]Some of the data will only valid for a subset of the concurrent objects.

1. `Fetch&AddLoop`: a synthetic benchmark will help in assessing the performance characteristics of the locks across different workloads.
2. `Fetch&Multiply`: a short operation that no hardware offers direct atomic support.
3. `Queue` and `PriorityQueue`: Common concurrent data structures with significant relevance in concurrent applications. They are also avaliable lock-freely.

2.2.3 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, capturing the response time.
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 [9].

I will examine the scalability of the locks by analyzing throughput in relation to an increasing number of threads. Fairness will be evaluated by considering both the duration each thread retains the lock (`hold_time`) and the responsiveness of the concurrent object to operations (`latencies`). This comprehensive methodological approach is designed to yield a lock mechanism that is both equitable and efficient.

3 Previous Experience

Starting in the interim between high school and college, I start contributing to an open-source project, Flow Launcher, for several years. I helped this project garnered over 5,000 stars on GitHub. It is developed in C#, a language that captivates me, especially its asynchronous programming model and parallelism capabilities.

In particular, I am intrigued by the application framework we utilize, *WPF*, which includes a feature known as Dispatcher to synchronize work to the UI thread. This is akin to a delegation-styled lock. My focus is on enhancing the application's performance through the implementation of various parallelism techniques.

My academic journey has included comprehensive coursework in systems programming, specifically CS 537 and CS 564. Prior to embarking on the proposed project, I have thoroughly studied two

influencial texts in the field of multiprocessor shared memory synchronization: *Shared-Memory Synchronization* and *The Art of Multiprocessor Programming*, which served as the foundational references for my work [7, 11].

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.

For future improvements, the project plans to tackle the combiner's response time issue by offloading non-critical work to a waiting thread that is anticipated to experience a longer wait time. This could be achieved through strategies inspired by the TCL Lock or by leveraging the asynchronous programming model provided by many modern languages (C++/Rust/C#/JavaScript) and manage `Future` executions within a custom runtime that adapts to lock-usage patterns.

Bibliography

[1] Alistarh, D. et al. 2015. The spraylist: A scalable relaxed priority queue. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), 11–20

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

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

[4] 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

[5] 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

[6] 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

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

[8] 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

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

[10] 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

[11] Scott, M. L. 2013. *Shared-memory synchronization*. Morgan & Claypool Publishers

[12] Shavit, N. and Lotan, I. 2000. Skiplist-based concurrent priority queues. *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000* (2000), 263–268