Usage Fair Delegation Styled Lock

by

Hongtao Zhang

A thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science

(Computer Science)

at the

UNIVERSITY OF WISCONSIN-MADISON

Fall 2024

This thesis was completed under the direction of:
Remzi Arpaci-Dusseau, Professor, Computer Science

Signature of Professor: _____ Date: _____

All Right Reserved

©Copyright by Hongtao Zhang 2024

to everyone I meet

Acknowledgments

Abstract

In multi-threaded enviroment, resources sharing and synchronization is an important topic. Locks are the most common synchronization technique for such problem. There are two fundamental properties a good lock want to achieve: 1) performance 2) fairness. Specifically, latest research emphasizes the need of usage fairness. Threads are not only allowed to enter the lock with different frequency, but also may use the lock for different amount of time.

On the first glance, it seems that these two properties are contradictory. Can we have a high performance lock while providing usage fairness?

To answer the above question, this dissertation proposes that lock should be designed as delegation: thread should not execute their critical section directly, but delegate the execution to a combiner. This thesis shows that under the setting of delegation styled lock, we can have a usage-fair lock without sacrificing performance. I introduce two types of fair delegation styled locks: 1) non-work-conserving `FC-Ban` and `CC-Ban` 2) work-conserving `FC-PQ`. My evaluation shows that all three of them can achieve high throughput and usage fairness simultaneously, while `FC-PQ` outperforms the other two when the lock is heavily contended while threads are having large non-critical section.

Contents

Lists of Figures

Lists of Tables

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 [1, 5]. These synchronization points, however, are often a source of contention and can become performance bottlenecks in a concurrent execution environment [2]. 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 [2, 4, 5].

1.1 Scheduler Subversion and Usage Fairness

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 [8]. They points out that one critical lock property, *lock usage*, is missing from the previous literacy. Consider the scenario where two threads' critical sections size are different: e.g. the first thread take three times longer to finish its critical section compared to the second thread. The presence of *imbalanced* critical section can disrupt the CPU's scheduling policy: allocate equitable processing time to 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.

The non-preemtive nature makes this problem hard. It is not allowed to switch lock owners when one thread hasn't finished its own critical section. Previous work offers locks that adopt usage-fairness, such as *SCLs* [8]. However, their solutions involves several limitations.

1. Switching lock owner incurs cost. *SCLs* raises a workaround: lock slice. Lock slice dedicates a timeslice to a owner, eliminating the need of owner switching during the duration. However, this makes *SCLs* non-work-conserving, wasting the lock usage if threads do not re-enter the lock immediately after releasing the lock.
2. The solution to lock usage is by banning threads based on the length of their critical sections and the number of threads that are trying to acquire

the lock. However, this solution is not work-conserving and the banning strategy is based on heuristic strategy.

Some latest research proposes *CFL* [7], which is work-conserving and reordered similar to the linux scheduler *CFS*. However, their solution still doesn't address the foundamental performance problem causes by lock owner switching.

1.2 Performance issue of fair locks

Foundamentally, fair lock means that the lock owner will be switching very frequenctly (presumably every single acquire if under high contention). However, this raises serious performance problem that makes lock foundamentally slower than sequencial programs.

In most scenario, a lock is protecting some shared memory location which requires atomic access. In general, memory can be read concurrently by multiple threads without problem. What a lock is trying to prevent is concurrent write. Whenever a thread is write to a shared memory location protected by a lock, it will carry over a memory barrier that invalidate all other cpus' cache toward the memory location. In a single threaded program, this will not incur performance penalty, because its own cache is still valid. However, in a fair lock scenario, whenever new thread is acquiring the lock (assuming the new thread does not lie in the same physical cpu as the previous thread), it will see that the cache toward the memory of the shared resource is invalidated. This forces the new cpu to re-acquire the shared memory location through at least *L3* cache or even the underlying memory, which is significantly slower than *L1* and *L2* cache. What's worse is this will happen for every single acquire and release of the lock, because the lock is trying to previde fairness and switch frequenctly among threads.

Due to this problem, most modern high performance locks only adopts what's called *eventually* fairness (which is what *SCL* with lock slice is trying to adopt) [8] **TODO: some other reference**. These locks don't try to switch the lock owner for every single acquire-release, but ensure that asymtotically (or in the long term) the acquisition/usage-fairness will be achieved. However, this means that threads may suffer from large tail latency because of the unfairness.

This problem raises to a question: can we have a lock that maintains fairness in its best effort in short term while preserving performance?

1.3 Delegation-styled locks

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. This execution manner allows the lock to switch owner (by executing other threads' critical sections) while do not need to invalidate cache (because it is still the same CPU that is accessing the shared memory location).

There are two predominant forms of delegation-styled locks: *combining* synchronization [2–4] and *client-server* synchronization [6, 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. This thesis is going to focus on combining locks, while the proposed technique can be easily adopts to the client-server locks.

Most delegation styled lock adopts either eventual acquisition fairness ([4, 6, 10]) or acquisition fairness ([2, 3]). However, none of them adopts usage fairness, in which the goal of this thesis is to bring delegation styled locks to the stage that it can demonstrate uncomparable advantage. We start by transforming existing delegation styled lock to a variant that adopts usage fairness through adopting the banning strategy utilized in *SCLs* — *FC-Ban* and *CCSynch-Ban*. Then we modify the inner structure with a concurrent re-ordering mechamism to produce a usage fair delegation styled lock — *FC-Skiplist*. At the very last, we show how to adopt any kind of serialized scheduling policy with delegation semantics with *FC-Channel*.

1.4 Overview

TODO:

Background

Concurrency, Synchronization & Mutual Exclusion

In modern computing systems, concurrency is fundamental to achieving high performance and resource utilization. Concurrent programming allows multiple computations to progress simultaneously, whether through true parallelism on multiple processors or through interleaved execution on a single processor. However, concurrent execution introduces challenges when multiple threads or processes need to access shared resources.

Synchronization mechanisms are essential tools that help coordinate concurrent operations and maintain program correctness. These mechanisms ensure that concurrent accesses to shared resources follow a proper order and maintain consistency. Without synchronization, concurrent access to shared data can lead to race conditions, where the program's outcome becomes unpredictable and depends on the precise timing of operations.

3.1 Race Conditions and Data Races

Race conditions and data races are two distinct but related concurrency hazards. A data race occurs when two or more threads access the same memory location concurrently, and at least one of these accesses is a write operation, without proper synchronization. For example, if two threads simultaneously increment a shared counter (`count++`), a data race can occur because this operation actually involves three steps: reading the value, incrementing it, and writing it back. The final value might be incorrect if the threads interleave these steps.

A race condition is a broader concept that occurs when the correctness of a program depends on the relative timing or interleaving of multiple operations. While data races are a common cause of race conditions, race conditions can exist even in programs free of data races. Consider a check-then-act sequence: checking if a file exists and then opening it. Even with properly synchronized individual operations, a race condition exists if another process deletes the file between the check and the open operation.

3.2 Mutual Exclusion

Mutual exclusion is a core concept in concurrent programming that ensures only one thread can access a critical section at a time. A critical section is a

region of code that accesses shared resources and must be executed atomically to maintain consistency. The classic example is a bank account balance: if two threads simultaneously try to modify the balance, concurrent access without mutual exclusion could lead to lost updates or incorrect final values.

Various mechanisms exist to implement mutual exclusion, with locks being one of the most common approaches. Locks provide a way for threads to coordinate access to shared resources by acquiring exclusive access before entering a critical section and releasing it afterward. The following sections explore different lock implementations, their characteristics, and the trade-offs they present in terms of performance, fairness, and complexity.

3.3 Lock

This section reviews the common lock primitives.

3.4 Common Lock Primitive Implementations

3.4.1 Naive Spinlock

3.4.2 Pthread Spinlock (test and test lock)

3.4.3 Exponential Backoff Spinlock

3.4.4 Pthread Mutex

3.4.5 Ticket Lock

3.4.6 MCS & K42 variant

3.5 Lock Usage

3.5.1 Scheduler Subversion

3.5.1.1 Imbalanced Scheduler Goals

3.5.1.2 Non-Preemptive Scheduling

3.6 Usage-Fair Lock

3.6.1 Scheduler-Cooperative Locks

3.7 Banning

3.7.0.1 u-SCL

3.7.0.2 k-SCL

3.7.0.3 RW-SCL

3.7.1 CFL

[7]

3.8 Delegation Styled Locks

3.8.1 Combine Style Locks

3.8.1.1 Flat Combining

[4]

3.8.1.2 CCSynch/DSM-Synch

[2]

3.8.2 Client-Server Styled Locks

`todo`

3.8.2.1 RCL

[6]

Not Sure whether to include this

3.8.2.2 ffwd

[10]

Not Sure whether to include this

3.9 Lock-Free Data Structures

3.9.1 Linked List

[5]

3.9.2 Skip-List

[5, 11]

3.9.3 Priority Queue

[5]

3.9.4 MPSC Channel

[9]

Usage Fair Delegation Styled Locks

This chapter will introduce different designs of delegation styled locks that are adopt the concept of usage fairness. We start with the banning strategy, then consider a more complex design that is inspired by the CCSynch. At last, we introduce a new design that allows us to adopt any serialized scheduling policy to the delegation styled locks.

We will only focus on the combining locks (e.g. FlatCombining, CCSynch, DSMSynch) in this chapter, while the revision of the client-server locks can be done similarly.

4.1 Banning Locks

We start by migrating the banning strategy of SCLs to the delegation styled locks. As mentioned in (@), SCLs adopts the following banning strategy:

$$t_{\text{banned until}} = t_{\text{last unlock}} + t_{\text{lock duration}} * \left(\frac{w_{\text{thread}}}{w_{\text{total}}} \right)$$

where $t_{\text{last unlock}}$ is the timestamp of the last unlock of the lock, $t_{\text{lock duration}}$ is the duration of the lock, w_{thread} is the weight of the thread, and w_{total} is the total weight of all threads.

By adopting a similar banning strategy, we dervie FC-Banning and CCSynch-Banning in this section.

4.1.1 Flat Combining (with Banning)

Implementing the banning strategy for FlatCombining is straightforward, because in FlatCombining, each thread owns its own node [4]. Therefore, we can simply record the banning time of the thread in the node and skip the critical section if the thread is banned.

The first step is to calculate the lock usage. Since only the combiner knows when the lock starts to acquire, we have two solutions to calculate the lock usage.

1. Let the combiner to record the timestamp before start executing a critical section, and then record again after finishing the critical section, then calculate the banning time of the thread.
2. Let the combiner to record the timestamp before start executing a critical section, and then pass the timestamp to the node before marking the job as finished. Then the waiter can calculate the lock usage by subtracting the timestamp from the current time, and mark itself as banned.

We implement the first solution in this thesis for simplicity. It may incur a tiny performance overhead since the combiner is doing more work. However, combiner will need to record the timestamp before start executing a critical section regardless and thus it can reuse the last unlock time as the start time of the next job with some small errors, which reduces the additional timestamp recording overhead. While although accessing w_{total} may incur a memory stall, the modification of w_{total} is rare (only happens when a new thread joins the lock because the combiner won't clean the old thread's weight until after finish the current pass) and thus the performance impact is acceptable.

Listing 1 refers to the node structure of FlatCombining with banning. The first 4 fields are the same as the node of FlatCombining, and the last field is the next available timestamp to execute the critical section of the thread.

```rust
pub struct Node<T> {
    pub age: UnsafeCell<u32>,
    pub active: AtomicBool,
    pub data: SyncUnsafeCell<T>,
    pub complete: AtomicBool,
    pub next: AtomicPtr<Node<T>>,
    pub banned_until: SyncUnsafeCell<u64>,
}
```

Listing 1: Node structure of FC-Banning

Listing 2 refers to the lock structure of FlatCombining with banning. The first 4 fields are the same as the lock of FlatCombining, and the last field is the next available timestamp to execute the critical section of the thread.

```rust
pub struct FCBan<T, I, F, L>
where
    T: Send + Sync,
    I: Send,
    F: Fn(&mut T, I) -> I,
    L: RawMutex,
{
    pass: AtomicU32,
    combiner_lock: CachePadded<L>,
    delegate: F,
    num_waiting_threads: AtomicI64,
    data: SyncUnsafeCell<T>,
    head: AtomicPtr<Node<I>>,
    local_node: ThreadLocal<SyncUnsafeCell<Node<I>>>,
}
```

Listing 2: Lock structure of FC-Banning

Listing 3 is a pseudo rust code[1] of the combining function of FlatCombining with banning. We can see that the combiner is executing the critical section of the thread only if the thread is not banned (Listing 3-11). Start at Listing 3-11, the combiner uses the current timestamp to calculate the lock usage and then update the banned time of the thread.

```rust
fn combine(&self) {
    let mut current_ptr = NonNull::new(self.head);

    while let Some(current) = current_ptr {
        if current.active && !current.complete {
            unsafe {
                current.age = pass;
                if work_begin >= current.banned_until {
                    ... // perform the delegation work
                    current.complete = true;
                    let work_end = __rdtscp(&mut aux);
                    // Banning
                    let cs = (work_end - work_begin);
                    current.banned_until += cs *
                    (self.num_waiting_threads);
                    work_begin = work_end;
                }
            }
        }
        current_ptr = NonNull::new(current.next);
    }
}
```

Listing 3: Combining function of FC-Banning

[1] omitting some rust details of unsafe and atomic operations: access the `UnsafeCell` and atomic load/store operations.

4.1.2 CCSynch with Banning

The implementation of CCSynch with banning is slightly more complex than FC-Banning, because in CCSynch, the nodes are cycled and utilized by different threads each time to ensure the FIFO order of the execution [2]. Therefore, threads must maintain additional thread-local data to record the banned time.

Compared to FC-Banning, CCSynch-Banning assigns the threads that trying to acquire the lock to spin (or wait) until its own banned time is passed for simplicity. This is slightly more performant compared to FC-Banning: the combiner in FC-Banning needs to perform additional checking toward whether a thread is banned.

Code Block 4 demonstrates the thread local structure and the node structure of CCSynch-Banning. At Code Block 4-3 we see the additional thread-local data used to detect whether the thread is banned.

```rust
pub struct ThreadData<T> {
    pub(crate) node: AtomicPtr<Node<T>>,
    pub(crate) banned_until: SyncUnsafeCell<u64>,
}

pub struct Node<T> {
  pub age: SyncUnsafeCell<u32>,
  pub active: AtomicBool,
  pub data: SyncUnsafeCell<T>,
  pub completed: AtomicBool,
  pub wait: AtomicBool,
  pub banned_until: SyncUnsafeCell<u64>,
  pub next: AtomicPtr<Node<T>>,
}
```

Code Block 4: Node structure of CCSynch-Banning

Listing 5 demonstrates the lock structure of CCSynch-Banning. This is identical to the lock structure of CCSynch, except that it adds aan additional field `num_waiting_threads` to record the number of threads waiting for the lock.

```rust
#[derive(Debug, Default)]
pub struct CCBan<T, I, F>
where
    F: DLock2Delegate<T, I>,
{
    delegate: F,
    data: SyncUnsafeCell<T>,
    tail: AtomicPtr<Node<I>>,
    num_waiting_threads: AtomicU64,
    local_node: ThreadLocal<ThreadData<I>>,
}
```

Listing 5: Lock structure of CCSynch-Banning

Listing 6 is the pseudo rust code of the local banning of CCSynch-Banning. At Listing 6-10, the thread checks whether it is banned. If it is banned, it will perform exponential backoff until the banned time is passed (theoretically one could use sleeping mechanism from the OS to reduce spinning).

```rust
fn lock(&self, data: I) -> I {
    ... // load thread data

    let banned_until = thread_data.banned_until.get().read();

    let backoff = Backoff::default();
    loop {
        let current = __rdtscp(&mut aux);

        if current >= banned_until {
            break;
        }
        backoff.snooze(); // expontential backoff
    }

    ... // Normal ccsynch logic

    // if your node is completed, ban yourself and return
    // otherwise, you are the combiner
    if current_node.completed {
        self.ban(thread_data, current_node.panelty);
        return current_node.data;
    }
}
```

Listing 6: Banning of CCSynch-Banning

```rust
fn ban(&self, data: &ThreadData<I>, panelty: u64) {
    data.banned_until += panelty;
}
```

Listing 7: Banning of CCSynch-Banning

Listing 8 is the pseudo rust code of the combine portion of CCSynch-Banning. Listing 8-12 calculates the length of the critical section and then updates the penalty of the node, and Listing 8-13 pass the panelty to current node, which

will be readed by the waiter and stored in the thread-local data. Listing 8-14 will carry a write fence (Release ordering) to ensure that waiter can see the updated panelty.

```rust
let mut work_begin = __rdtscp(&mut aux);
while let Some(next_node) = next_ptr {
    if counter >= H {
        break;
    }
    counter += 1;
    tmp_node.data = (self.delegate)(
        self.data.get().as_mut().unwrap_unchecked(),
        tmp_node.data.get(),
    );
    let work_end = __rdtscp(&mut aux);
    let cs = work_end - work_begin;
    tmp_node.panelty = cs * self.num_waiting_threads;
    tmp_node.completed = true;
    tmp_node.wait = false;

    work_begin = work_end;
    tmp_node = next_node;
    next_ptr = NonNull::new(tmp_node.next);
}

// ban yourself and return
self.ban(thread_data, current_node.panelty);

return current_node.data;
```

Listing 8: Combine portion of CCSynch-Banning

4.1.3 Alternative Banning Strategy

The banning strategy inherited from SCLs is simple and effective. It has an interesting property that it only consider the total usage of the lock. Thus, if a

thread has slept for a long time and then wakes up and acquire the lock for a very long time, it won't be banned.

We can propose an alternative banning strategy that doesn't consider the previous behavior, but using a moving average of the average critical section to calculate the banning time.

The banning time is calculated as below:

$$t_{\text{banned until}} = t_{\text{last unlock}} + \max\left(0, \text{cs} \times n_{\text{thread}} - \text{cs}_{\text{avg}}\right)$$

where the moving average of the critical section is calculated as below:

$$\text{cs}_{\text{avg}} \leftarrow \text{cs}_{\text{avg}} + \frac{\text{cs} - \text{cs}_{\text{avg}}}{n_{\text{exec}}}$$

The rationale behind maintaining an additional average critical section usage is to address situations where only a few threads share similar critical section lengths. This approach helps minimize the duration during which all threads are banned.

4.2 Naive Priority Locks

4.2.1 FC-Skiplist

[11]

4.3 Serialized Scheduling Locks

Charateristics & Experiments

5.1 Throughput

5.2 Latency of single acquire

Future Work

References

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

[2] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012. 257–266.

[3] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. 2023. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with {TCLOCKS}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023. 1–16.

[4] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, 2010. 355–364.

[5] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[6] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: Migrating {Critical-Section} execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012. 65–76.

[7] Jonggyu Park and Young Ik Eom. 2024. Locks as a Resource: Fairly Scheduling Lock Occupation with CFL. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*, 2024. Association for Computing Machinery, Edinburgh, United Kingdom, 17–29. https://doi.org/10.1145/3627535.3638477

[8] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. 2020. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020. 1–17.

[9] Erik Rigtorp. 2021. Optimizing a ring buffer for throughput. Retrieved from https://rigtorp.se/ringbuffer/

[10] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017. 342–358.

[11] Nir Shavit and Itay Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000. 263–268.