Contents

1 Introduction

Delegation locking adopts the request-response style of communication to minimize shared data movement. Specifically, the waiter delegates their critical section to a combiner (Flat-Combining/ CC-Synch) [1, 2], or a dedicated thread (RCL/ffwd) [3, 4]. We will describe the implementation details of these locks in Section 2. Further, the idea of combining allows us to batch process operations for special data structure. For example, multiple insertion and deletion operations can be batched into a single operation for a linked list.

Noticeably, the idea of *lock slice* utilized in U-SCL also accommodate with the idea of delegation-style locks by making sure that only one thread is executing the critical section at a given time slice, thus reducing the need of moving shared data around cores and improve performance [5]. This is very similar to the goal of delegation-style locks, and thus provide similar performance boost. However, it has a serious drawback: within a valid *lock slice*, no other thread can hold the lock even when the previous thread is not holding the lock. Thus, the non-critical section needs to be very short or else the throughout will be greatly impacted. On the other hand, delegation-style locks provides the similar performance benefits without the drawback of *lock slice*.

The main issue of delegation-style locks are the refactoring of old code, as it doesn't provide a similar API as regular locks. However, recent work has demonstrated the potential of transparent delegation to resolve this issue [6].

On the other hand, most delegation-style locks inherently provides some fairness guarantee. There's no reason that the combiner shall treat their critical sections different from others', so most of the policy is to enumerate all ready jobs. For example, the combiner of FlatCombining (or the server of RCL) enumerates all the thread list to check whether a waiter is trying to execute a critical section. This is not the case for spin-lock. A few of the threads might dominate the lock usage and starve the others. If threads are repeatedly reacquired the lock, the thread that are releasing the lock will have some advantage to reacquire the lock as the cache synchronization is faster for them.

However, previous work has demonstrated that acquisition fairness of lock is not enough to mitigate the problem of scheduler subversion [5]. For example, Figure 1 has demonstrated the unfairness of CC-Synch even if it maintains a strictly FIFO order given varying critical section size. The 16 threads that are incrementing a shared counter are split into two groups: the first group will run for 10ns, and the second group will run for 30ns. We can easily see that threads in the two groups contribute to the shared counter differently — proportion to their critical section size.

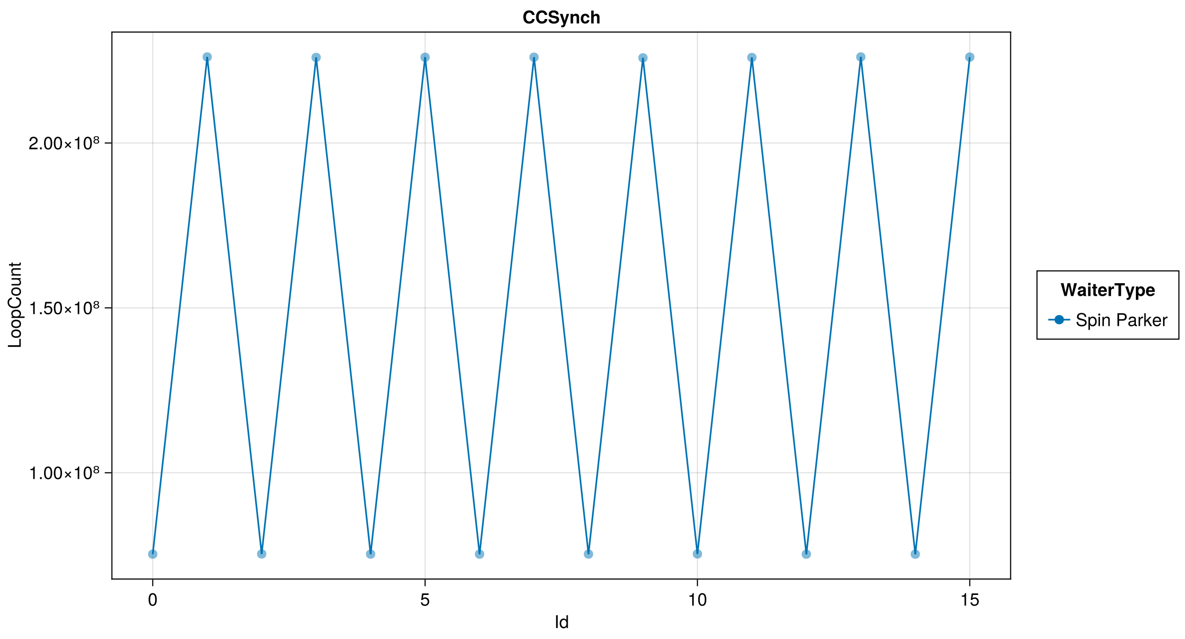


Figure 1: #ccsynch Loop Count per thread

To mitigate these problems, we modify the original algorithm of FlatCombining and CC-Synch to achieve the fairness goal. We majorly adopted two strategies to mitigate the issue.

1.1 Banning

Similar to how U-SCL is implemented, we ban the thread that is trying to execute the critical sections for too long. Currently, we have implemented two fair variants of Flat-Combining and CC-Synch based on this idea.

The banning time is calculated as below:

$$\max(0, \mathrm{cs} \times n_{\mathrm{thread}} - \mathrm{cs}_{\mathrm{avg}})$$

where the average critical sections are calculated incrementally.

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

The reason I use subtract an additional average critical section usage is to mitigate the cases where there's only a few threads that share similar critical section length. Without the subtraction, there might be some time when all threads are banned, which will greatly decrease the performance.

1.2 Priority Based Structure

On the other hand, we can replace the linear concurrent data structure with a prioritized data structure. For example, Flat Combining maintains a Linear list of threads. We can replace that with a Concurrent Skip-List, which allows us to prioritize the threads based on their lock usage.

2 Implementation Details

This work implement the following delegation-style locks in *rust* (so we assume some understanding toward *rust*): FlatCombining [1], CC-Synch [2], RCL [3].

2.1 Job Type

Each lock implement a common trait (interface) as follows, where the generic type represents the shared data:

```rust
pub trait DLock<T> {
    fn lock<'a>(&self, f: impl DLockDelegate<T> + 'a);

    #[cfg(feature = "combiner_stat")]
    fn get_current_thread_combining_time(&self) -> Option<std::num::NonZeroI64>;
}
```

and the trait `DLockDelegate` refers to the critical sections defined as follows:

```rust
pub trait DLockDelegate<T>: Send + Sync {
    fn apply(&mut self, data: DLockGuard<T>);
}
```

The `DLockGuard<T>` is similar to the `MutexGuard<T>` in *rust*, which implements the `Deref` and `DerefMut` traits. The `DLockGuard` is used to access the shared data. Unlike `MutexGuard`, which is also used to release the lock when dropped, the lock is automatically released when the delegate finishes.

```rust
pub struct DLockGuard<'a, T: ?Sized> {
    data: &'a SyncUnsafeCell<T>, // UnsafeCell is used to allow interior mutability
}
```

To make the lock easy to use, we implement `DLockDelegate<T>` for lambda function type `FnMut(DLockGuard<T>) + Send + Sync`. Thus, the user can use the lock as follows:

```
lock_type.lock(|mut guard: DLockGuard<u64>| {
    ...
});
```

2.2 Parker

TODO!

2.3 Delegation-style Lock

To implement a delegation-style lock, we need to consider to aspect.

1. How the jobs are published
2. How the combiner is selected

There are two natural ways to publish jobs: per thread or per job.

For "per thread", I mean each thread owns a dedicated memory location (node) that allows it to publish a job with its context. Then the combiner enumerate the threads' nodes, check whether the node is ready, and if it is ready, execute the job. Flat Combining (Section 2.3.1), RCL (Section 2.3.2) and FFWD adapts this way of publishing jobs.

For "per job", each thread should publish the job for execution to a dedicated job queue. The combiner will then enumerate the job queues to execute the jobs. CC-Synch and its variants adapt this way of publishing jobs.

2.3.1 Flat Combining

Flat Combining maintains a list of node that is owned by each thread. Each node contains the context of the thread and the job that the thread is trying to execute. The combiner will enumerate the list of nodes to check whether the node is ready. If the node is ready, the combiner will execute the job and update the node to indicate that the job is finished.

The struct is defined as follows:

```
#[derive(Debug)]
pub struct FcLock<T, L, P>
where
    L: RawSimpleLock,
    P: Parker,
{
    pass: AtomicU32,
    combiner_lock: CachePadded<L>,
    data: SyncUnsafeCell<T>,
    head: AtomicPtr<Node<T, P>>,
    local_node: ThreadLocal<SyncUnsafeCell<Node<T, P>>>,
}
```

The `pass` field is used to indicate the age of the lock, thus able to removing threads that hasn't executed job for too long. The `head` field is the head of the list of thread nodes. The `local_node` field is used to store the thread local node of the current thread, similar to `pthread_getspecific`. The `combiner_lock` is used for threads to elect the combiner. The implementation idea is retrieved from [7] (code are avaliable in Section 4.1).

The node struct is defined as follows:

```
pub(super) struct Node<T, P>
where
    P: Parker,
{
    pub(super) age: u32,
    pub(super) active: AtomicBool,
    pub(super) f: CachePadded<Option<*mut (dyn DLockDelegate<T>)>>,
    pub(super) next: *mut Node<T, P>,
    pub(super) parker: P, // id: i32,
    #[cfg(feature = "combiner_stat")]
    pub(super) combiner_time_stat: i64,
}
```

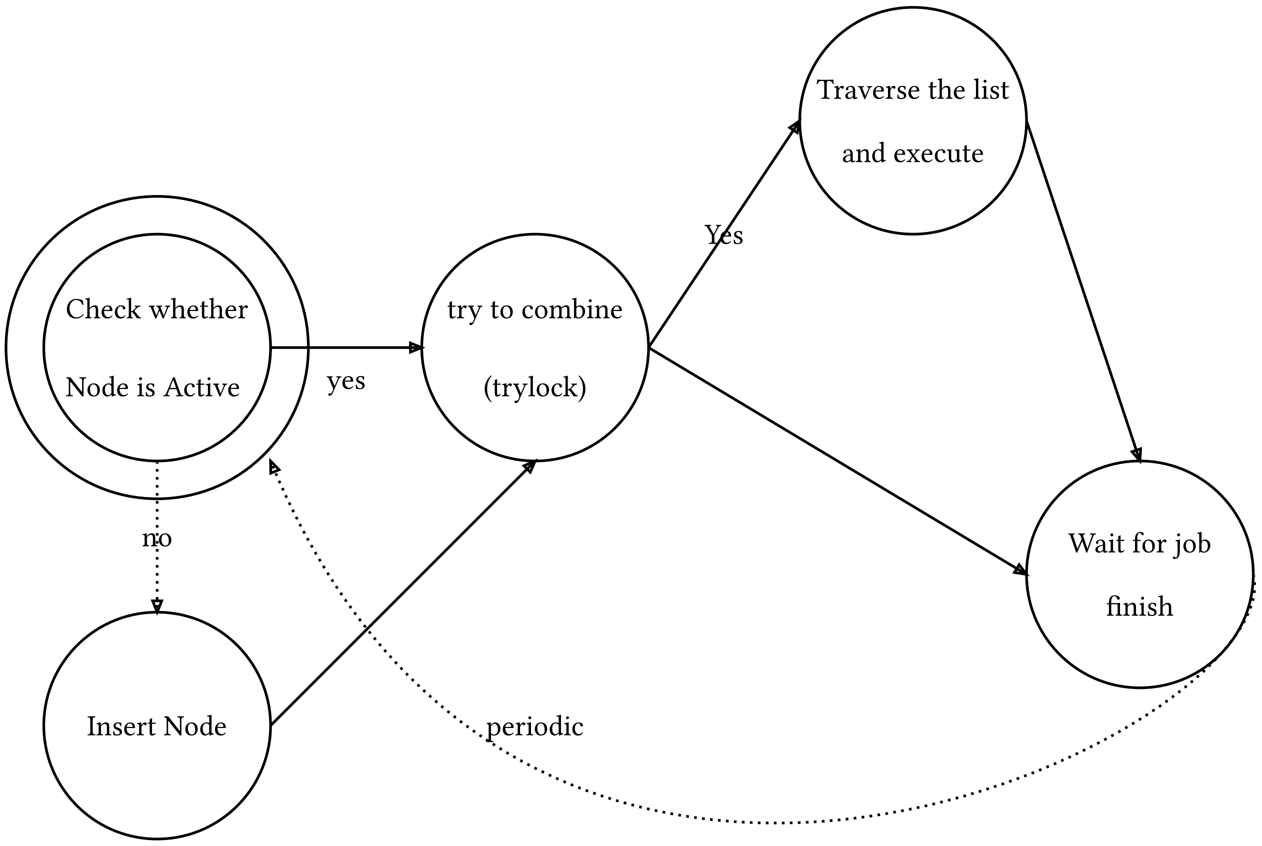
The `age` field synchrnizes with the `pass` field of the lock to indicate the age of the node. If `age` is too far from `pass`, the node will be removed from the linkedlist and marked as false at `active`. The `f` field is the function pointer to the job that the thread is trying to execute. The `next` field is the pointer to the next node in the linkedlist. The `parker` field is the parker that is used to block the thread when job is published but not yet executed (refer to Section 2.2).

The original paper proposed that the ready state can be embedded into `f` field, thus saving a bool field [1]. However, we choose not to do this in our implementation for tow reason.

The `combiner_time_stat` field is used to record the time that each thread becomes the combiner. It is not marked with `Atomic` because only the combiner (which is guarded by the `combiner_lock`) will overwrite this field (maybe volatile is needed?).

1. In rust, as we encapsulate the job into a trait object, whose pointer is a fat pointer (which contains two pointer). We don't want to have another layer of indirection that points to the fat pointer, which cannot be accessed atomically (with rust std). With an additional bool field as a write fence (`Release` Ordering), we make sure that the combiner will see the whole job before the node appears to be ready.
2. We want to extract the waiting as a `parker` field, so that we can share blocking code among different locks. This is not so possible if we embed the ready state into `f` field.

The overall algorithm can be roughly represented by the following graph:



The insertion is done via a simple CAS loop similar to a singly linked list:

```
fn push_node(&self, node: &mut Node<T, P>) {
    let mut head = self.head.load(Ordering::Acquire);
    loop {
        node.next = head;
        match self
            .head
            .compare_exchange_weak(head, node, Ordering::Release, Ordering::Acquire)
        {
            Ok(_) => {
                node.active.store(true, Ordering::Release);
                break;
            }
            Err(x) => head = x,
        }
    }
}
```

The combiner will periodically remove the unactive nodes. The head of the list will not be removed to make sure the removal can be done when new node is added.

2.3.2 Remote Core Locking

Similar to FlatCombining, RCL maintains a list of node that is owned by each thread. The main difference is that RCL dedicate a thread to behave as the server (combiner). Therefore, the original paper suggests that it behaves like a remote procedure call. Subtly, rcl employs an array of nodes, where FlatCombining maintains a linked list of nodes. This should provide better performance if the number of inactive threads are small.

One of the main goal of RCL is that it allows easy migration of legacy code. Therefore, it employs complicated algorithm to make sure that the server thread won't be blocked or actively spinning so that the performance is affected. Further, the design of RCL allows the server to handle multiple lock so that legacy code can be easily ported. We didn't implement the full version of RCL and thus will not focus on these.

Though, one of the main goal is kept in our implementation that one server can handle different types of locks. This is a large burden toward *rust* type system, as we want to statically dispatch the type information so that no performance regression will be found. The detail of the implementation is in appendix Section 4.2.

2.3.3 CC-Synch/H-Synch

CC-Synch maintains a FIFO queue of jobs. In contrast to Flat-Combining, each node represents a job instead of a thread. Further, the combiner is not elected by a lock, but instead done lock-freely based on the position of the job queue.

Intuitively, a thread will be elected as a combiner if one of the following conditions is satisfied:

1. Its job is at the end of the queue
2. Previous Combiner has executed too much critical sections.

Compared to FlatCombining, it is important to note that an additional limit is required to make sure that the combiner will not execute too many critical sections. The job queue can be extended infinitely, and thus the combiner might never return to complete its other jobs. In the contrast, the largest number of jobs done by FlatCombining's combiner is bounded by the number of threads, which we can assume to be finite.

The algorithms can be sketched below

1. Each thread holds a *mutable* thread-local pointer to a node. (Note that in FlatCombining the node always belong to the thread that creates it).
2. When a thread wants to execute a critical section, it swaps the pointer storage with the current head.
3. Then assign the task to the **old head node** that is just being swapped, and connect to the previous node that is now the **head**.
4. Then there's a slightly complicated mechanism to make sure no lock is required.
   1. Each node has two properties: `wait` and `completed`
   2. `wait` will be false if no one is combiner. Before a thread swaps its thread-local node to head, the `wait` will be assigned to true.
   3. After a thread assigns its task. It will check the `wait` property. If `true`, then wait till it becomes false.

4. After `wait` becomes `false`, it will check the `completed` property. If it hasn't been completed, then the thread becomes the combiner. There's a subtle argument about ordering that I will skip saying why this is enough to only check the `completed` property after `wait` is false.
5. The combiner will stop at the `head` node, where it doesn't have children (there will always be an extra dummy node for swapping). Mark its wait as false, and exit.

H-Synch is a NUMA-awareness version of CC-Synch, which is not yet implemented.

2.4 Fair delegation-style locks

We present two kinds of approach to implement a fair delegation-style locks. One of the idea is based on banning (similar to how SCLs is implemented [5]), and the other is based on priority-based structure.

2.4.1 Banning

The idea of banning is to ban the thread that is trying to execute the critical section for too long. Banning is not special or easy to implement for delegation-style locks, but given that it is the most natural way of implementing a fair lock, we still implement a fair variant of FlatCombining and CC-Synch based on this idea.

2.4.1.1 Flat-Combining (Banning)

Adapting the idea of banning, the combiner will calculate how long it takes to execute the critical sections, and calculate when the thread should be banned based on the algorithm in Section 1.1. Similar to how U-SCL is implemented, only the timestamp that the thread can resume execution will be recorded.

When enumerating the thread list, the combiner will firstly check whether the thread is banned, and only then perform the ordinary FlatCombining algorithm.

Note that this will involve some additional cost, as more nodes will behave "inactive" because they are banned. One resolution is to remove threads that are banned from the list, but this will involve some additional cost to maintain the list. Some heuristic algorithm may be used to decide whether the node should be removed.

Another subtle thing is that we can let the waiter decide whether it is banned instead of wasting combiner's time to check whether the thread is banned. This is the way CC-Synch Ban is implemented Section 2.4.1.2, but involves some additional cost of transferring the critical section lengths of the job. Given we implement a pre-wake mechanism, one resolution is to let waiter record the timestamp when it is pre-waked, and calculate the critical section when it is eventually waked.

2.4.1.2 CC-Synch (Banning)

In contrary to Flat-Combining (Banning), the combiner now have no idea about whether a thread is banned. Instead, each waiter will stop inserting its node into the job queue when it is banned. The combiner will only record the length of critical section and send that to the nodes that are waiting. The node will then decide when it is banned until based on the algorithm in Section 1.1.

Noticeably, the banning strategy utilized here involves a much less performance penalty compared to Flat-Combining (Banning) given that combiner won't need to traverse useless nodes. However, this also suggests that slightly threads may be needed to achieve the same number of combined jobs as CC-Synch.

2.4.2 Priority-based Structure

The idea of priority-based structure is to replace the linear data structure with a prioritized data structure. For example, FlatCombining maintains a Linear list of threads. We can replace that with a Concurrent Priority Queue, which allows us to prioritize the threads based on their lock usage. This idea is similar to *Linux CFS*, which uses a red-black tree to prioritize the threads based on their execution time. This method is more inherent to delegation-style locks, as there is a scheduler like character: the combiner.

2.4.2.1 Flat-Combining (Priority-Queue) (Not implemented)

We can replace the linked-list of FlatCombining with a priority queue (either a heap or a balanced tree). However, they are not inherently concurrent as linked list, and thus either we need to find a concurrent implementation or make sure mutual exclusion to them. Luckily, the combiner is a natural character to modify and control the queue.

Similar to Flat-Combining, we still utilize a thread-local node to publish the job. The difference is that thread will push the node to a prepared queue (linked list) lock-freely, and then the combiner will pop the node from the queue and push the thread node to the priority queue.

Before each pass, the combiner will check whether there are additional prepared thread waiting to be inserted into the priority queue. If there is, then the combiner will pop it from the prepared queue and insert into the priority queue.

Then it just executes jobs like Flat-Combining through the prioritized data structure.

2.4.2.2 Flat-Combining (Skip-List)

We can maintain a job queue that is prioritized based on the critical section length with the help of a concurrent skip list. The combiner will be elected as Flat-Combining and then execute the job from the head of the priority queue.

This is probably one of the simplest idea one can think of when trying to adapt the idea of priority-based structure to implement a fair delegation-style lock. Subtly, it is not accurate to call it a variant of Flat-Combining, as it opts to the idea of a job queue (similar to CC-Synch) instead of a thread queue.

Similar to CC-Synch, a combiner limit is important here, given that it is possible to have infinitely many jobs waiting to be executed. Our current implementation is based on a *contribution* limit, which is similar to the *H* limit in CC-Synch but calculated based on timestamp instead of number of jobs.

Currently, we use a 3rd-party implemented concurrent skip list, which is not yet optimized for our use case [8].

2.4.2.2.1 Future work

The election process can be done similar to how CC-Synch is done to avoid election through a lock. This will require a customized concurrent skip-list and a more involved understanding toward how to elect a combiner.

2.4.3 Response Time

Beyond lock-usage unfairness, delegation-style lock involves an additional unfairness. The combiner is doing work for others, and thus the response time of the combiner is greatly impacted. For example, in Flat-Combining, the combiner of the lock will need to wait the worse case of the response time, which is twice as the average response time of the waiters. This is a serious issue for Flat-Combining when the waiter are blocked to wait. As shown in the left plot of Figure 2, two of the 32 threads are combining more than 80% of the time. The situation will become even worse when thread number if larger. (Remark: When implementing the first version of FLAT-COMBINING in *rust*, I found that the blocking version of FLAT-COMBINING is faster than the spin wait one, and the analysis of combining time resolves the reason behind that.)

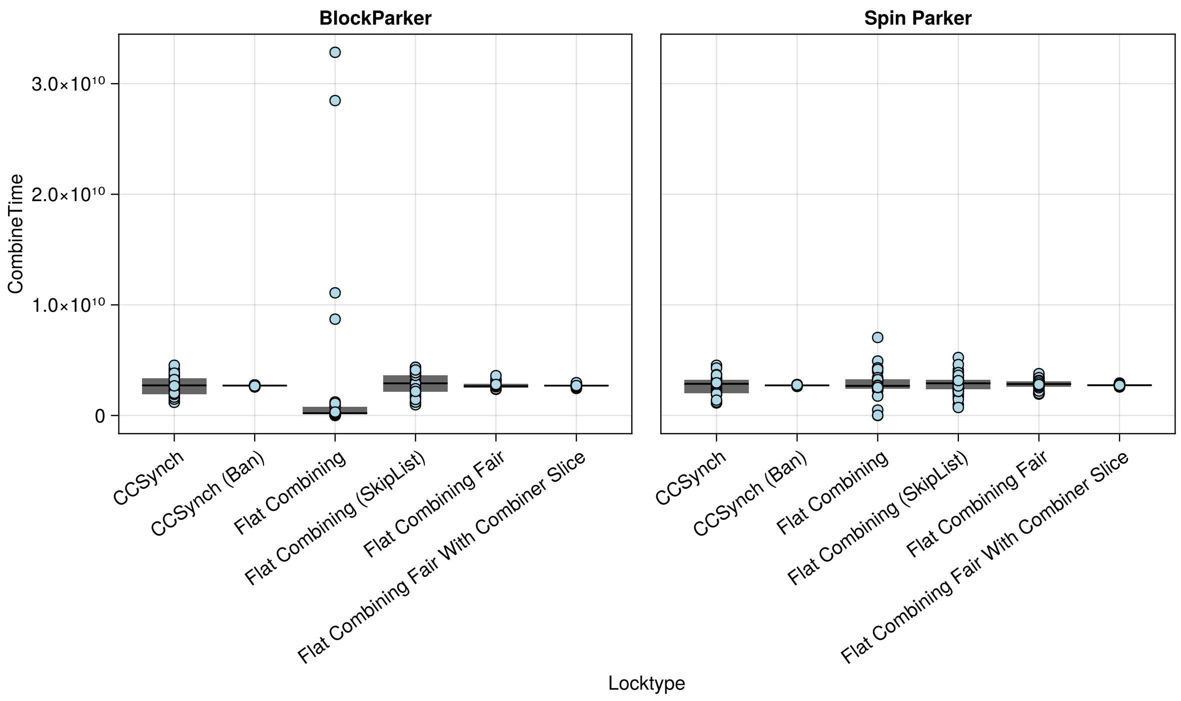


Figure 2: Combining Time Distribution of 32 threads execution

This gives us some insight that why we not only need to care about the scheduling of waiters, but also the scheduling of the combiner. From another aspect, we can reduce the average response time by carefully choosing which thread should become the combiner. For example, we should always let the tail of the threads list to become the combiner in FLAT-COMBINING. Because of the "flat" execution, the tail node will always be executed at last, which means that its response time is always the sum of all critical sections in the current pass. Therefore, becoming a combiner will not give any penalty toward its response time. In the contrary, if the head node becomes the combiner, its response time reduces greatly from its own critical section length to the sum of all critical sections in the current pass (this is not always the case though. Considering the case that its job is ready after combiner already started and skipped its node).

Experiments regarding response hasn't been added, and all these ideas also haven't been implemented.

2.4.3.1 Combiner Slice

The idea of *Combiner slice* is similar to the idea of *lock slice*, but slightly differently. A thread should always become the combiner given a time slice, and yield when other combiner slice is valid. This might yield performance boost when the length of the slice is carefully chosen, but also face the same issue of *lock slice* that the non-critical section needs to be short enough.

TODO!

2.4.3.2 Flat-Combining (Banning, Combiner-Slice)

TODO! This is not a good implementation and is not an implementation of the combiner slice I describe above.

3 Experiments

We benchmark the performance by incrementing a shared counter for a given time slice.

The following locks are benchmarked in our example:

1. Flat-Combining
2. Flat-Combining (Banning)
3. Flat-Combining (Banning, Combiner-Slice) (Please ignore now)
4. Flat-Combining (Skip-List)
5. CC-Synch
6. CC-Synch (Banning)
7. RCL
8. Mutex (rust)
9. Spinlock (Section 4.1)
10. U-SCL ([5])

The workers will be split into two groups. The first group will run for 10ns, and the second group will run for 30ns.

Since all delegation-style is implemented with a generic parker. We will split the result based on the two parker.

3.1 General Performance Comparison

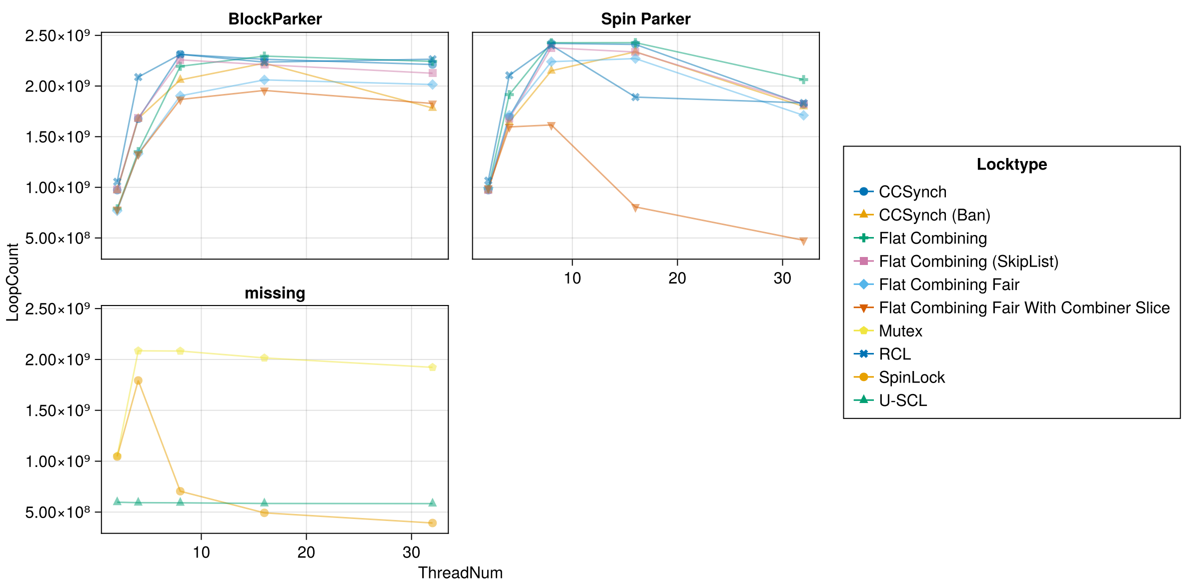


Figure 3: Performance Comparison

Bibliography

[1] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. Twenty-Second Annu. ACM Symp. Parallelism Algorithms Architectures*, 2010, pp. 355–364.

[2] P. Fatourou, and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 257–266.

[3] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: migrating $\{$critical-section$\}$ execution to improve the performance of multithreaded applications," in *2012 USENIX Annu. Tech. Conf. (USENIX ATC 12)*, 2012, pp. 65–76.

[4] S. Roghanchi, J. Eriksson, and N. Basu, "Ffwd: delegation is (much) faster than you think," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 342–358.

[5] Y. Patel, L. Yang, et al., "Avoiding scheduler subversion using scheduler-cooperative locks," in *Proc. Fifteenth Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

[6] V. Gupta, K. K. Dwivedi, et al., "Ship your critical section, not your data: enabling transparent delegation with $\{$tclocks$\}$," in *17th USENIX Symp. Operating Syst. Des. Implementation (OSDI 23)*, 2023, pp. 1–16.

[7] Taewoo An (codingskynet), *Concurrent Data Structure for Rust*, (2023). [Online]. Available: https://github.com/codingskynet/concurrent-data-structure

[8] Taiki Endo, Jules Bertholet, et al., *Crossbeam Skiplist*, (2023). [Online]. Available: https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-skiplist

4 Appendix

4.1 RawSpinLock

The RawSpinLock is implemented as a test-and-test-and-set lock with exponential backoff:

```
#[derive(Debug)]
pub struct RawSpinLock {
    flag: AtomicBool,
}

unsafe impl RawSimpleLock for RawSpinLock {
    fn new() -> Self {
        Self {
            flag: AtomicBool::new(false),
        }
    }

    #[inline]
    fn try_lock(&self) -> bool {
        if !self.flag.load_consume() {
            self.flag
                .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
                .is_ok()
        } else {
            false
        }
    }

    #[inline]
    fn lock(&self) {
        let backoff = Backoff::new();

        while !self.try_lock() {
            backoff.snooze();
        }
    }
    #[inline]
    fn unlock(&self) {
        self.flag.store(false, Ordering::Release);
    }
}
```

4.1.1 Remark

The additional load doesn't seem to provide better performance to FlatCombining given that most of the time the thread will only `try_lock` once and then block for a while. However, it does provide better performance when used completely alone.

4.2 RCL Implementation Details

TODO!