ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE | ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL | ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks

Vishal Gupta   *Kumar Kartikeya Dwivedi* [*]   *Yugesh Kothari*   *Yueyang Pan*

Diyu Zhou   *Sanidhya Kashyap*

EPFL   [*]*SRMIST*

Abstract

Today's high-performance applications heavily rely on various synchronization mechanisms, such as locks. While locks ensure mutual exclusion of shared data, their design impacts application scalability. Locks, as used in practice, move the lock-guarded shared data to the core holding it, which leads to shared data transfer among cores. This design adds unavoidable critical path latency leading to performance scalability issues. Meanwhile, some locks avoid this shared data movement by localizing the access to shared data on one core, and shipping the critical section to that specific core. However, such locks require modifying applications to explicitly package the critical section, which makes it virtually infeasible for complicated applications with large code bases, such as the Linux kernel.

We propose *transparent delegation*, in which a waiter automatically encodes its critical section information on its stack and notifies the combiner (lock holder). The combiner executes the shipped critical section on the waiter's behalf using a lightweight context switch. Using transparent delegation, we design a family of locking protocols, called TCLocks, that requires zero modification to applications' logic. The evaluation shows that TCLocks provide up to 5.2× performance improvement compared with recent locking algorithms.

1 Introduction

Synchronization mechanisms are the basic building blocks for today's high-performance concurrent applications. In fact, applications heavily rely on locks as a concurrency control mechanism, as they provide a set of simple programming APIs for users to mediate concurrent access to shared data. Besides ensuring program correctness, locks also affect the scalability of applications [33, 34, 49]. For instance, various high-performance applications, such as the Linux kernel, have moved from coarse-grained to fine-grained locks [52] for minimizing the length of the critical section. However, thanks to diverse workloads and applications, the scalability problem due to lock algorithms still remains at large [41, 55, 62, 70].

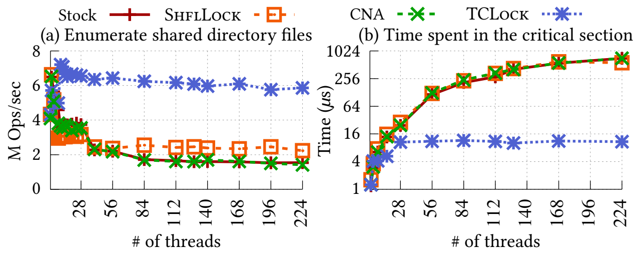


Figure 1: Impact of locks on a file-system micro-benchmark [62]. We compare three traditional lock algorithms: Linux's qspinlock (*Stock*) [41], CNA [43] and ShflLock [52] with our proposed TCLock. (a) Enumerating files in a shared directory on an 8-socket 224-core machine. (b) Time spent in the critical section: moving the critical section context (TCLock) compared with moving critical section shared data (*Stock*, CNA, and ShflLock).

As a result, research in lock algorithms focuses on minimizing the contention on cache-lines containing the lock word and shared data. The most widely used algorithms always move the shared data to the core executing the critical section [30, 41, 43, 52, 53, 60]. Lock evolution within this design philosophy has focused on reducing contention on the lock word. However, such a lock design still moves shared data across cores for every lock acquisition. Figure 1 shows that such shared data access cost increases with increasing cores, thereby limiting the scalability of applications.

On the other end of the design spectrum, some algorithms adopt the request-response style of communication, also called delegation-style locking [37, 47, 51, 56, 64, 67]. Specifically, waiters delegate their critical section execution context to a dedicated core [56, 67] or a *combiner* [47, 51] that executes that function on behalf of each waiter in a specific order. Figure 1 illustrates that this design outperforms traditional locks and improves application performance. In particular, such a lock design minimizes the shared data movement, and ensures almost constant critical section latency regardless of the number of threads.

Despite potential performance gains, the practical design and implementation of delegation-style locks faces several challenges. First, applications require major rewriting to explicitly encapsulate and pass the critical section as a function pointer [51, 56, 67]. Unfortunately, this rewriting becomes impractical for applications with large code bases, such as the Linux kernel, which has over 180k lock API call sites [52]. Second, every delegation-based work focuses on situations involving a single lock contention. However, today's applications often employ fine-grained locking and may acquire multiple locks for operations, such as memory, scheduler, and storage management in the Linux kernel [4, 12, 13, 20]. Finally, the third challenge involves managing the per-CPU or per-thread variables, which applications heavily depend on for either performance or correctness.

In this paper, we take the first step towards making delegation-based locks practical for concurrent applications with large code bases. We introduce the idea of *transparent delegation*, which enables developers to utilize delegation-style locking without rewriting the application. Our transparent delegation approach encapsulates the critical section using two observations: First, a thread's stack and CPU registers contain the state of the waiter's thread. Second, using the lock/unlock API pushes the thread's context on its stack. Thus, a waiter *saves its critical section context using CPU registers and stack pointer, and calling the lock API as a function.* Finally, the combiner executes the waiter's critical section on its behalf by assuming the role of the waiter using a lightweight context switch mechanism [29, 59]. This context-switch mechanism is transparent to the application.

Using transparent delegation, we design a new family of locks called TCLocks that augment existing locks, such as test-and-set (TAS) and MCS, by employing the combining technique for batching waiters' requests [47]. Our first lock is a spinlock, where waiters continuously spin while awaiting the lock. The combiner can execute multiple waiter's critical section before passing its role based on a counter-based mechanism. Similar to our prior work (ShflLocks) [52], our algorithms can enforce hardware and software policies on the fly. In particular, our spinlock version also incorporates NUMA-awareness policy. We then integrate the core oversubscription policy [52] to design a blocking lock, where the waiter can sleep while waiting for the lock. Lastly, we design a phase-based readers-writer lock built on top of our blocking lock.

Applying TCLocks directly in highly concurrent systems presents its own set of challenges. First, transparent delegation violates the single-writer property of a thread's stack, meaning that two threads (the combiner and the waiter) writing to the same stack can cause data races and stack corruption. Waiters need access to a stack due to specific events, such as interrupts in the kernel space, signals in userspace, and scheduling of waiting threads. We address the data-race issue using a *per-thread ephemeral stack* that a waiter switches to between the acquire and release phases.

Second, most concurrent applications use multi-level locking [4, 20, 28] and out-of-order (OOO) unlocking [12, 13] for higher concurrency and better scalability. TCLocks handle the arbitrary level of nested combining by maintaining combiner-specific state on the ephemeral stack before acquiring the nested lock. Meanwhile, we handle OOO unlocking by keeping track of the order of acquired locks. We delay the release of OOO unlocked locks until the order is the inverse of acquired locks. This effectively flattens the release of locks.

We evaluate TCLocks in both kernel space and userspace on NUMA machines. TCLocks improve the performance within and across sockets. Specifically, TCLocks boost application throughput by 1.7–5.2× compared to the locks used in the Linux kernel and state-of-the-art locks, respectively.

In summary, this paper makes the following contributions:

- **Design technique.** We introduce a new design technique called transparent delegation. Locks with this technique allow developers to use the same APIs as traditional locks while benefiting from the scalability improvements provided by delegation-style locking.
- **Delegation-based lock family.** We implement TCLocks that employ transparent delegation. We first design a spinning lock and extend it to blocking and readers-writer locks, utilizing per-thread ephemeral stacks to manage the parking of waiters.
- **Practical application.** TCLocks incorporate various lock use scenarios, including nested locking and out-of-order unlocking. This approach allows us to realize the potential of delegation-style locking for the Linux kernel without modifying any code.

2 Background

While executing a critical section, a thread accesses three types of memory locations (data):

1. **Lock word**, *i.e.*, its structure that determines the exclusive access for a thread.
2. **Shared data** among threads guarded by a lock word, accessible only to the thread holding the lock.
3. **Thread-local data** like stack and per-thread variables.

Most lock designs minimize the contention on the lock word, while some minimize the movement of shared data. Hence, there are two design philosophies based on shared data movement: traditional and delegation-style. We now discuss the evolution of locks based on these design philosophies. Later, we touch upon the systems-level challenges that are specific to delegation-style locks.

2.1 Traditional Locks

Traditional lock design adheres to the principle of moving data to computation. A core executes the critical section by moving shared data into its cache. Consequently, this design moves cache lines of both the lock word and shared data across cores while executing the critical section. The evolution of traditional lock algorithms [50] has focused on minimizing cache-line movement of the lock word. For example, queue-based locks [41, 42, 44, 58, 60, 68] minimize cache-line contention due to the lock word. Hierarchical locks [39, 46, 57, 66] further reduce the cache-line contention on non-uniform memory access (NUMA) machines, where accessing a local-socket memory location is faster than a remote one. These locks amortize the remote access cost of the lock word by reordering the wait queue to pass the lock within the same socket. ShflLock [52] and CNA [43] further generalize hierarchical lock design by reordering the wait queue based on various hardware and software policies. Moreover, our recent work [65] has also shown that the reordering policy can be changed dynamically without kernel compilation.

Readers-writer locks also follow traditional lock design, with most locks aiming to minimize contention on the lock word [36, 54, 61]. These locks augment mutually exclusive locks with different types of read indicators based on workload requirements. Some examples include centralized [61], per-socket [38], and per-CPU [40, 63, 71] indicators. These locks also require moving shared data across cores, even though they offer a broader semantics of mutual exclusion.

Traditional locks do not require modifying applications since the lock/unlock programming APIs remain consistent. However, these locks move shared data cache lines among cores while executing the critical section. Unfortunately, this lock design incurs shared data movement for every critical section execution, thereby increasing critical section execution latency. Moreover, this latency grows with increasing core count (Figure 1 (b)), which saturates the throughput without efficiently utilizing hardware.

2.2 Delegation-style Locks

Delegation-style locks follow the principle of moving computation to data [45, 47, 51, 56, 64, 67]. These locks use an old technique called combining that has been used in hardware and software to mitigate memory contention by combining requests for the same memory location. In this approach, waiters pack their critical section as a function and pass that function pointer to the combiner as a request. The combiner then executes the waiter's function and notifies it upon completion. Executing the critical section on the same core eliminates shared data movement, leading to improved application throughput with increasing core count (Figure 1).

However, this lock design has a critical limitation. It does not provide the same lock/unlock APIs as traditional locks [47, 67]. Consequently, we need to modify applications, which involves identifying each critical section in the code, wrapping it as a function, and modularizing the application logic for delegation. Modifying application logic to encapsulate the critical section as a function is quite challenging and even impossible in some cases [46]. For instance, Roghanchi *et al.* [67] reported modifying ∼1,500 lines of code (LoC) to enable delegation for Memcached. This limitation, unfortunately, prevents the scalability offered by delegation-style locking from being applied to existing real-world applications, such as Linux, which comprises 28M LoC with more than 180k static lock call sites.

2.3 The Incompatibility of Delegation in Concurrent Applications

Real-world applications, such as the Linux kernel, employ fine-grained locking in multiple execution contexts. Fine-grained locking mostly involves acquiring multiple nested locks when working with several objects. To prevent deadlocks, the nested locks are acquired in a specific order, but they can be released in arbitrary order to enhance concurrency [12, 13]. However, none of the delegation approaches handle such common, but challenging cases. We measured that both nested locking calls and OOO unlocking calls are quite prevalent. For instance, booting Linux results in ∼80k nested locking calls and ∼20k OOO unlock calls. Thus, addressing these cases is essential to make delegation-style locks practical for every concurrent systems software.

In addition, the Linux kernel can call locks from various contexts. These contexts comprise of task [25] and interrupts (*e.g.*, non-maskable interrupt context [11], HardIRQ context [6, 7], or SoftIRQ context [23])). The kernel code typically executes in the task context. Depending on the kernel configuration, a scheduler can preempt or migrate a task to another CPU. Nevertheless, in special execution contexts, such as interrupts, or code regions that disable CPU preemption and migration, the scheduler prohibits the migration of such contexts. The Linux kernel code also heavily utilizes per-CPU variables and implicitly relies on stable access to these variables in such special execution contexts. Traditional lock design does not require any handling for special execution contexts because the critical section executes on the core that acquires the lock. In contrast, delegation-style locks break this property, necessitating special handling for these cases, *i.e.*, special contexts and stable access to per-CPU variables.

Goal. In accordance with the general design principle of minimizing data movement [43, 52], our objective is to reduce data movement for both lock word and shared data. Unlike existing delegation-style locks, we avoid making any code modifications. Hence, we take the initial step towards achieving the goal of transparently enabling delegation-style locking for any real-world application, including the OS.

3 TCLocks

We propose transparent delegation, a practical lock-design technique for real-world applications that allow developers to use the same lock/unlock APIs as traditional locks without modifying the application code. Transparent delegation involves two steps: First, it automatically encapsulates a critical section of arbitrary length in a set of registers and the thread stack. Second, waiters pass this encapsulated information to the combiner for execution. As a result, our approach enables applications to enjoy scalability without any modification. We apply this technique to design a family of lock algorithms called TCLocks, that transparently delegate waiters' requests to the combiner. TCLocks comprises spinning (§3.2) and blocking (§3.4) locks. We extend the blocking lock with read indicators to design a phase-based readers-writer blocking lock (§3.5) incorporating hardware and software-based optimizations (§3.6).

3.1 TCLock Design

We first discuss a set of insights and techniques that allows us to design and implement TCLocks.

Transparent delegation. When executing a critical section, a thread can access both shared data (*e.g.*, global variables, heap) and thread-local data (*e.g.*, registers, stack, and per-thread variables). In delegation-style locking, although shared data is globally available to all threads, the combiner requires access to the waiter's thread-local data and the set of instructions for executing its critical section.

Our technique overcomes the challenge of thread-local data and critical section context using three key insights.

1. A thread's execution context is well-defined by hardware, with thread-specific CPU registers and the stack containing all information for executing the critical section [2, 8, 10].
2. A waiter busy-waits without modifying its state once it sends its request to the combiner. It exits only after receiving the response from the combiner.
3. Calling the lock API as a function[1] ensures that hardware pushes the next instruction onto the stack, making the critical section's start address available to the combiner for executing the critical section.

Using these insights, the combiner pops the start address of a critical section from the waiter's stack using a *return* instruction and executes it. After completing the critical section, calling the unlock API pushes the first instruction of the non-critical section onto the waiter's stack. The waiter resumes executing the non-critical section after receiving a notification from the combiner. Thus, transparent delegation allows waiters to seamlessly pass context and resume after the critical section's execution.

Avoiding concurrent stack access with an ephemeral stack. To ensure program correctness, transparent delegation must prevent concurrent accesses to a thread's execution stack. Ideally, a waiter busy-waits for notification during its critical section execution. However, certain events, such as interrupts in kernel space, signals in user space, and the waiter's parking and wake-up mechanism [32] can access the waiter's stack during the execution of its delegated critical section. As a result, naive transparent delegation via stack switching violates the fundamental single-writer stack principle, leading to potential stack state corruption.

To address this issue, we introduce an initially empty, separate stack called the *ephemeral stack*. Each waiter switches to its ephemeral stack during lock acquisition, and delegates its critical section to the combiner. The waiter then busy-waits using the ephemeral stack while the combiner accesses the waiter's main stack to execute the critical section. Importantly, the use of an ephemeral stack does not introduce any new stack overflow bugs since it is a separate memory from the thread's execution stack. By incorporating the ephemeral stack, TCLocks maintain the single-writer principle, thereby preventing concurrent access and the corruption of waiters' stack.

[1]For example, the `call` instruction in x86.

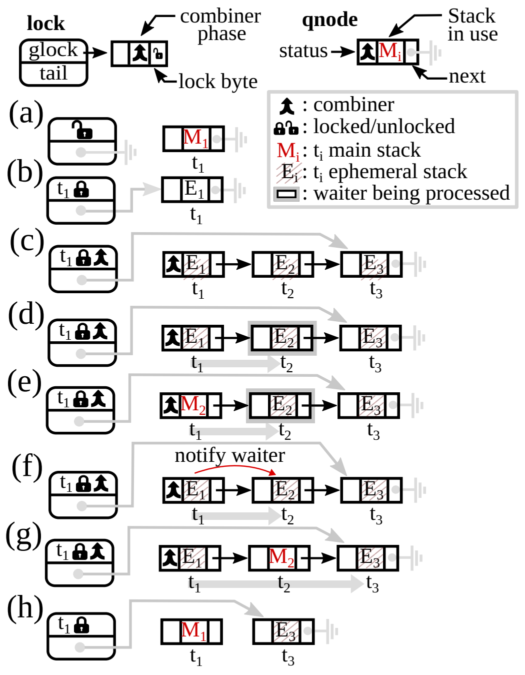


Figure 2: Lock and `qnode` structures of the TCLock. (a) Initially, the lock is in the unlock state. $\texttt{t}_1$ first switches from its main stack ($\texttt{M}_1$) to the ephemeral stack ($\texttt{E}_1$), and (b) joins the waiting queue. (c) $\texttt{t}_1$ being at the head of the queue, becomes the combiner. Meanwhile, $\texttt{t}_2$ and $\texttt{t}_3$ join the queue. They also switch their stack to $\texttt{E}_2$ and $\texttt{E}_3$, respectively. (d) $\texttt{t}_1$ begins the combining process by traversing the queue and finds $\texttt{t}_2$. (e) $\texttt{t}_1$ switches to $\texttt{t}_2$'s main stack ($\texttt{E}_1 \rightarrow \texttt{M}_2$) and executes $\texttt{t}_2$'s critical section. (f) Once finished, $\texttt{t}_1$ first switches back to $\texttt{E}_1$ and then notifies $\texttt{t}_2$ that $\texttt{t}_1$ has finished executing its critical section. (g) $\texttt{t}_2$ then switches back to $\texttt{M}_2$ and exits its unlock phase. Meanwhile, $\texttt{t}_1$ finds $\texttt{t}_3$ as the last waiter. (h) $\texttt{t}_1$ notifies $\texttt{t}_3$ that it is now at the head of the queue, then $\texttt{t}_1$ switches its stack to $\texttt{M}_1$, executes its critical section, and finally exits the unlock phase.

3.2 Spinlock: TCLock^{SP}

TCLock^{SP} augments the TAS and MCS lock by adopting the combining technique from MCS-style combining works [45, 47]. It involves a waiting thread becoming a combiner and batch waiters' requests up to a set threshold. Specifically, TCLock extends the DSM-Synch lock, using TAS as a top-level lock, and an MCS-style waiting queue for waiters. The waiter's queue node (`qnode`) maintains additional states: 1) `request` and `wait` flags for synchronizing between a waiter and the combiner and selecting the next combiner. 2) Socket ID for NUMA lock design (§3.6.3). 3) Batch count to limit excessive waiters, causing starvation or long-term fairness issues. And, 4) a pointer to the waiter's thread context for transparent delegation, which includes all registers and the stack pointer.

Transparent delegation invariants. Our lock algorithm maintains four invariants: 1) A combiner is always at the head of the waiting queue. 2) A waiter never uses its main stack while busy waiting. 3) All instructions in a critical section are executed only once, either by a waiter or the combiner executing on the waiter's behalf. 4) A combiner exclusively executes the waiter's critical section instructions defined between the lock and the unlock phase.

Workflow. Figure 2 presents a running example of TCLock^{SP}. Before requesting a lock, every thread executes in its main context (a). When a thread requests a lock, it switches to an ephemeral stack, saves its main context in its qnode, and joins the queue (b). Now, the head of the queue (t_1) becomes the combiner, while other threads (t_2 and t_3) join the queue after switching to their respective ephemeral stacks. They wait for notification from the combiner, while processing any interrupts and signals on their ephemeral stacks. The combiner iterates through the queue and finds t_2's request (d). t_1 context-switches to t_2's main context using t_2's qnode, and starts executing t_2's critical section (e). Reaching the unlock API of t_2, t_1 switches back to its ephemeral stack, notifies t_2, and checks for other requests (f). Once t_2 receives notification, it switches back to its main context, which now points to the end of the critical section. It then continues executing its non-critical section (g). Finally, the combiner iterates through the entire queue, it passes the combining role to t_3, switches to its main context, and executes its critical section (h). Finally, t_1 unlocks the lock, allowing t_3 to acquire it and continue the combining process.

Algorithm. Listing 1 presents the TCLock^{SP} pseudocode. A thread `t` first attempts to acquire the TAS lock on the fast path (line 17). On success, `t` executes its critical section directly. Otherwise, `t` finds its thread-local combining structure (line 21), saves its register state on the main stack, switches to the ephemeral stack, and begins the slow path (lines 26–27). The slow path comprises four phases: 1) `t` joins the queue and busy-waits locally. 2) `t` then waits to acquire the TAS lock after becoming the head of the queue. 3) After acquiring the TAS lock, `t` checks the combining conditions. 4) Finally, `t` combines waiters' critical sections.

Phase 1: Busy-waiting phase. Upon entering the slow path, `t` initializes its `qnode` (line 32). Specifically, it sets the `wait` field to True, `request` field to `UNPRCSD`, and the `next` pointer to `None`. The combiner notifies a waiter with the `wait` flag and uses the `request` flag to specify whether it executed a waiter's critical section. `t` then adds itself to the waiting queue by atomically swapping the tail with the `qnode`'s address (line 36). After that, `t` checks for any preceding waiters in the queue. If true, `t` joins the queue as a waiter (line 38) and waits for the combiner's notification (lines 39–40); otherwise, it proceeds to phase 2. While in the queue, `t` busy-waits for the combiner to execute its critical section. After reaching the end of `t`'s critical section, the combiner pushes the first instruction after the unlock API (line 107) on `t`'s main stack. It then marks `t`'s request as complete (*i.e.*, `PRCSD`) (line 28), which switches to its main context and begins the non-critical section (line 107). If, however, `t`'s request is not completed, `t` reaches the head of the queue and moves to phase 2.

Phase 2: Global lock acquisition phase. `t` now tries to acquire the TAS (global) lock using the `CAS` operation (lines 46–50).

Phase 3: Combining-role decision phase. After acquiring the global lock, `t` checks whether it can be a combiner (lines 52–56). If `t` is the only one in the queue, it resets the queue tail, (lines 52–53), switches to its main stack (line 28), executes its critical section, and releases the lock. Otherwise, `t` checks if there are at least two waiters in the queue. If true, `t` proceeds to phase 4 as a combiner. Otherwise, `t` passes the combining role to the next waiter (lines 58–60) by setting the `wait` bit to false, and releases the lock after executing its critical section.

Phase 4: Combining phase. `t` begins the combining phase by disabling the fast path, thereby forcing new waiters to join the queue (line 63). `t` iterates over the queue to execute each waiter's critical section (lines 65–75). Within the loop, `t` selects the next waiter (4a line 67), and records the waiter's information (`qnode`) in its thread-local combiner struct (`cst`) to later use it for resuming the combining process. Then `t` switches from its ephemeral stack to the waiter's main stack, and executes the waiter's critical section (4b line 70). Once finished, `t` notifies the waiter by first setting the `request` flag to `PRCSD` and resetting the `wait` flag (4c line 72). `t` continuously iterates until it reaches the combining threshold or cannot find two subsequent waiters in the queue (4d lines 73–75). After exiting the loop, `t` ends the combining phase by changing the locking mode to the non-combining mode (line 77). `t` then notifies the next waiter to be the head of the queue (line 78), and finally executes its critical section.

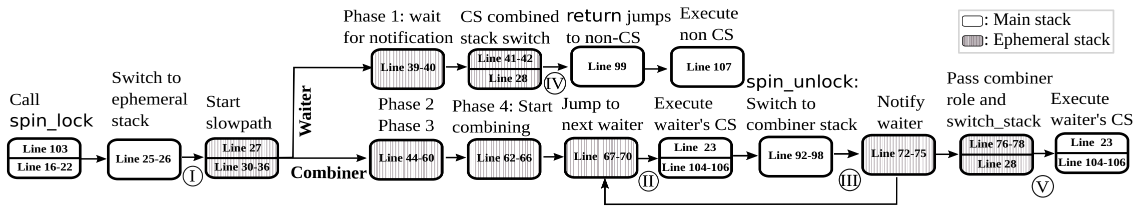
During the unlock phase, `t` can be in one of the two states: G_LOCKED: `t` unlocks the TAS lock by resetting its value and returns (lines 92–94). G_LOCKED_COMBINER: `t` does not release the TAS lock. It context switches from a waiter to the combiner by switching from the waiter's main stack to the combiner's ephemeral stack (lines 98) After switching, `t` resumes the combining loop and notifies the waiter about the completion of the critical section (line 72).

3.3 Proof Sketch of Correctness

Mutual exclusion. TCLock ensures mutual exclusion by maintaining two invariants: First, only one thread can hold the global TAS lock, which can also be a combiner; Second, the main stack of a thread is active on only one thread at any time. TCLock piggybacks on the mutual exclusion property

```
1  PRCSD    = 0 # Waiter's request is processed by the combiner
2  UNPRCSD  = 1 # Waiter's request is not processed until now
3  G_UNLOCK = 0, G_LOCKED = 1 # TAS known states
4  G_LOCKED_COMBINER = 2 # State to mark combining phase
5  WAITERS_TO_COMBINE = 1024 # Combining batch count
6
7  class thread_local_combiner_struct:
8    qcurr = None, qprev = None, qnext = None, node = init_node()
9    counter = 0, lock_addr = Array[None]
10   estack_rsp = init_ephemeral_stack()
11
12 class lock:
13   glock = 0, tail = None # TAS: top level lock, MCS queue
14
15 def spin_lock(lock):
16   # Fastpath: Try to acquire the TAS lock
17   if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
18     return # Got the lock, going to execute the critical section
19
20   # Switch to the ephemeral stack and acquire the lock in slowpath
21   cst = this_thread_comb_struct() # Get the per-CPU combiner struct
22   switch_stack(lock, cst) # Switch stack and begin slowpath function
23   return
24
25 def switch_stack(lock, cst):
26   switch_to_ephemeral_stack(cst.node) # Main → ephemeral stack
27   lock_slowpath(lock, cst)
28   switch_from_ephemeral_stack(cst.node) # Ephemeral → main stack
29
30 def lock_slowpath(lock, cst):
31   qnode = cst.node # Get the pointer to qnode in the combiner struct
32   init_qnode(qnode, wait = True, request = UNPRCSD ,
33     next = None, skt_id = numa_id()) # Initialize waiter's qnode
34
35   # Phase 1: Busy waiting: Join the queue and wait until notified
36   qprev = SWAP(&lock.tail, &qnode) # Atomically add node to tail
37   if qprev is not None: # Waiters are already present in the queue
38     qprev.next = qnode  # Link qprev with qnode to form a queue
39     while qnode.wait is True:
40       continue # Wait for the combiner to halt waiter's spinning
41     if qnode.request == PRCSD : # Waiter request has been processed
42       return #Combiner executed my CS; jump to non critical section
43
44   # Phase 2: Global lock acquisition: Acquire the TAS lock
45   # Waiter is at the head of the queue; get the TAS lock
46   while True: # Wait for the glock to be unlocked
47     while lock.glock != G_UNLOCK:
48       continue
49     if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
50       break  # Got the TAS lock
51   # Phase 3: Combining-role decision: Whether to combine
52   if CAS(&lock.tail, qnode, None) == qnode:
53     return # If only one in the queue, return
54   qnext = qnode.next # Someone joined the queue; get qnode ptr
55   while qnext is None:
56     qnext = qnode.next
57   # If there are at least two waiters, start combining
58   if qnext.next == None:
59     notify_next_queue_head(qnext) # next waiter is combiner
60     return
61
62   # Phase 4: Combining: Batch requests with dynamic policies
63   lock.glock = G_LOCKED_COMBINER  # Declare combining phase
64   counter = 0
65   while True:  # Combiner loop
66     qcurr = qnext # Get the very next waiter after combiner
67     qnext = select_next_waiter(qcurr) # 4a Get the next node
68     cst.qcurr = qcurr # For qcurr's stack switch in unlock()
69     # 4b Combiner's ephemeral stack → next waiter's stack
70     switch_stack_from_combiner_to_waiter(cst, qcurr)
71      # Waiter's critical section execution finished
72     notify_waiter(qnode) # 4c Mark as completed
73     if qnext is None or qnext.next is None or
74       counter >= WAITERS_TO_COMBINE: # 4d Check comb. cond.
75       break
76   # Combiner phase is over, now combiner runs its CS
77   lock.glock = G_LOCKED   # Reset TAS lock to normal lock
78   notify_next_queue_head(qnext) # Next waiter is the combiner
79
80 # Select the next node based on the policy, eg., NUMA etc.
81 def select_next_waiter(qnode):
82   return qnode.next
83
84 def notify_next_queue_head(qnode):
85   qnode.wait = False
86
87 def notify_waiter(qnode):
88   qnode.request = PRCSD
89   qnode.wait = False
90
91 def spin_unlock(lock):
92   if lock.glock == G_LOCKED:
93     lock.glock = G_UNLOCK # Only true for no combining phase
94     return
95   # Jump back to combiner
96   cst = this_cpu_comb_struct()
97   # Waiter's stack → combiner's ephemeral stack
98   switch_stack_from_waiter_to_combiner(cst.qcurr, cst)
99   return
```

```
101
102 class main:
103     spin_lock(A)
104     instr1
105     instr2
106     spin_unlock(A)
107     instr3
```



Listing 1: Pseudocode of TCLock along with the algorithm flow. In the bottom figure, Shade of the boxes show which stack is currently active and the numbers Ⓘ–Ⓥ shows stack switching locations in the algorithm. At each of these locations, following return addresses are present on the outgoing stack: Ⓘ Outgoing: Waiters' main stack → line 23. Ⓘⓘ Outgoing: Combiner's ephemeral stack → line 71. Ⓘⓘⓘ Outgoing: Waiter's main stack → line 99. ⒾⓋ and Ⓥ Outgoing: waiter's and combiner's ephemeral stack → contents are discarded.

of the TAS lock as it uses atomic compare-and-swap (CAS) to guarantee thread exclusivity. Hence, thanks to the TAS lock, only one thread can hold the global lock at any given point and only one thread can access shared data at a time. Finally, our transparent delegation invariants ensure that a waiter never touches its own main stack while waiting for a combiner's notification. Moreover, TCLock ensures that the switch from the waiter's main stack to the combiner's ephemeral stack (line 98) occurs at the end of the critical section, *i.e.*, at the end of the unlock function. Thus, after the waiter restores its context from the main stack (line 28), it never enters its critical section.

Correct thread state. TCLock preserves the correct waiter's state using a lightweight context switch mechanism and avoids concurrent stack modification. Specifically, a waiter yields ownership of its main stack (line 26) before joining the waiting queue (line 36). Thus, a combiner thread can only obtain the ownership of a waiter thread's main stack after the waiter gives up the ownership. Finally, the combiner thread concedes its ownership of the waiter's main stack (line 98) before notifying the waiter (line 72). Therefore, our approach ensures that when a waiter reacquires the ownership of its main stack (line 28), the combiner is not using that stack.

3.4 Blocking Lock: TCLock^{B}

TCLock^{B} follows a similar design philosophy of the blocking ShflLock, where waiters use the spin-then-park strategy. In this approach, a waiter spins locally until its time quota expires. Upon expiration, it schedules itself out if the system is oversubscribed; otherwise, it yields to the scheduler, which eventually reschedules the waiter. In addition, the lock queue maintains both active and passive waiters.

We design TCLock^{B} by augmenting TCLock^{SP} to support the parking/wakeup policy. We extend the combiner's role, which now wakes up sleeping waiters while executing their critical sections. The use of an ephemeral stack becomes critical for TCLock^{B} because parking of waiters requires calling a function, which pushes the function frame on the waiter's stack. Hence, TCLock^{B} uses the thread-local ephemeral stack to prevent concurrent accesses. The stack switching protocol remains the same as in TCLock^{SP}. To enable efficient parking and wakeup, we add two new states to the `request` field of the `qnode`: PARKED, in which a waiter is scheduled out, and PRCSING, which indicates that the combiner has started executing a waiter's critical section.

In the slow-path phase, while spinning locally (*i.e.*, phase 1), a waiter `t` checks if its time quota is up. If so, `t` attempts to park itself out. Specifically, `t` tries to change its `request` field from UNPRCSD to PARKED atomically. If successful, `t` parks itself out; otherwise, it continues spinning as the combiner has changed `t`'s state. In phase 4c, while selecting the next head of the queue (`notify_next_queue_head()`), the combiner atomically swaps `t`'s state to PRCSING to prevent the waiter from going to sleep. Furthermore, after executing the critical section, the combiner atomically swaps `t`'s state to PRCSD. In both cases, the combiner checks the old state of the `request` field. If it is PARKED, the combiner wakes up `t`. We use atomic instructions for changing the state to prevent the lost wakeup problem.

3.5 Readers-writer Version: TCLock^{RW}

TCLock^{RW} is a combining-aware readers-writer lock that allows readers to execute in parallel, while writers are combined. TCLock uses a phase-based mechanism [35, 36] that alternates between readers and combined writers. TCLock^{RW} comprises the following: 1) A counter that includes the reader count (`RCNT`), writer present byte (`WP`) to indicate if a writer is holding the lock, and writer waiting byte (`WW`) indicating a writer waiting to acquire the lock. 2) A writer queue (`tail`) for combining and parking waiting writers. This queue is similar to our TCLock^{B}'s queue. 3) A

```
RCNT  = 1 << 16; WW  = 0x100; WP  = 0x1;
WCOMBINER = G_LOCKED_COMBINER
class rwlock: (32 byte lock)
  # rwcounter → [RCNT: 17-63; WW: 8-16; WP: 0-7]
  rwcounter = 0 # 8-byte readers-writer rwcounter
  tail = None # Writers enqueue in this queue
  wlock: mutex # Coordinate bw readers & first writer

def down_read(rwlock): # Acquire read lock
  atomic_inc(&rwlock.rwcounter, RCNT) # Increment reader count
  if !(rwlock.rwcounter & 0xffff): # Check the first two bytes
    return # Lock acquired, if writer not present or waiting
  atomic_dec(&rwlock.rwcounter, RCNT) # Decrement reader count
  read_lock_slowpath(rwlock) # execute read slowpath

def read_lock_slowpath(rwlock):
  mutex_lock(&rwlock.wlock) # Acquire mutex
  atomic_inc(&rwlock.rwcounter, RCNT) # Increment reader count
  while (rwlock.rwcounter & 0xffff) > 0: # Check first two bytes
      continue # Wait for writer to finish
  mutex_unlock(&rwlock.wlock) # Release the mutex

def up_read(rwlock): # Release read lock
  atomic_dec(&rwlock.rwcounter, RCNT) # Decrease reader count

def down_write(rwlock): # Acquire write lock
  # The writer tries to set the WP byte (as 1)
  if CAS(&rwlock.rwcounter, 0, WP) == 0:
    return # Writer fastpath

  # Switch to the ephemeral stack and acquire lock in slowpath.
  cst = this_cpu_comb_struct() # Get the per-CPU comb struct
  switch_stack(rwlock, cst)
  return
def lock_slowpath(lock, cst): # Write lock slowpath
    ...
-  # Waiter → queue's head; get the TAS lock
-  while True: # Wait for the glock to be unlocked
-    while lock.glock != G_UNLOCK:
-      continue
-    if CAS(&lock.glock, G_UNLOCK, G_LOCKED) == G_UNLOCK:
-      break  # Got the TAS lock

+  # Replace spinning on glock with rwcounter
+  mutex_lock(&lock.wlock)  # Acquire mutex
+  if CAS(&lock.rwcounter, 0, WP) == 0:
+     goto unlock  # Success if no readers are present.

+  atomic_inc(&lock.rwcounter, WW)  # Indicate writer waiting
+  while True:  # Spin until all readers finish CS
+    if CAS(&lock.rwcounter, WW, WP) == WW:
+      break
+  unlock:
+  mutex_unlock(&lock.wlock)  # Release mutex
  # MCS unlock phase
  ...

-  # Now, qnext is the combiner, indicated by glock word
-  lock.glock = G_LOCKED_COMBINER
+  # Now, qnext is the combiner, indicated by rwcounter word
+  lock.rwcounter = WCOMBINER

-  # Combiner phase is over, now combiner will run its CS
-  lock.glock = G_LOCKED  # Reset TAS lock to normal lock
+  lock.rwcounter = WP

def up_writer(rwlock): # Release write lock
+  if rwlock.rwcounter == WP:
    rwlock.rwcounter = 0
    ...
```

Listing 2: Pseudo-code for TCLock^{RW}.

mutex, called `wlock`, that synchronizes the phase between readers and the head of the writers queue. Hence, `wlock` handles the parking of readers and the head of the write queue. We use the ShflLock^B algorithm [52]—a traditional NUMA-aware queue-based mutex—for `wlock` than TCLock^B because maintaining a centralized count of readers (shared data) contradicts the design of combining that tries to localize the access to the shared data.

Algorithm. Listing 2 shows the necessary changes to TCLock^{SP}. A reader first atomically increments `RCNT` and executes its critical section if no writer is present (lines 10–11). Otherwise, it decreases the `RCNT` (line 13), and enters the slow-path phase. The reader first acquires `wlock` (line 17), it then increments the `RCNT` (line 18) to mark that a reader phase should begin soon, and waits for existing writers to exit (line 20). Finally, it unlocks `wlock` (line 21) and executes its critical section. In the unlock phase, a reader releases the lock by atomically decreasing `RCNT` (line 24).

A writer enters the critical section if it successfully switches `WP` from 0 to 1 (line 28). Otherwise, it switches to the ephemeral stack and begins the slow path phase (line 33). This slow path follows the same protocol as TCLock^{SP} except that the head of the waiting queue (line 44) acquires the `wlock` (line 45). After acquiring `wlock`, the writer tries to enter the critical section by atomically setting the value to `WP` (line 46). On failure, it sets the `WW` byte to 1 to prevent new readers from entering the critical section (line 49) and waits for other readers to leave. Once they leave, the writer atomically modifies the `rwcounter` from `WW` to `WP` (line 50–52), releases `wlock`, and starts the combining process. In the unlock phase, a writer resets the `rwcounter` to 0 if the value is `WP`.

3.6 Optimizations

We propose three key optimizations to minimize further the data movement between the combiner and a waiter, and the cache-line bouncing of the lock word.

3.6.1 Direct stack switching: waiter → waiter

In the current TCLock^{SP} version (§3.2), the combiner switches stack twice before executing the next waiter's critical section. Specifically, it first switches to its own context, finds the next waiter to combine, and then switches to the next waiter's context. To avoid the switch to the combiner's context, we split the combining loop. In particular, after switching the stack to a waiter's context, the combiner tries to select the next waiter to combine after the current waiter (4a), and notifies the previous waiter that its critical section execution is over (4c). The combiner then exits the lock function call and executes the critical section of the current waiter. After executing the critical section, *i.e.*, in the unlock phase of the current waiter, the combiner checks the combining loop conditions (4d). If the condition holds, the combiner directly switches to the next waiter's context (4b). Otherwise, it marks the end of the combining phase, switches back to its context, notifies the previous waiter, and finally executes its own critical section.

3.6.2 Minimizing context switch overhead

Our combining approach suffers from saving, transferring, and restoring a thread's contexts while executing the critical section. We leverage both compiler and hardware techniques to minimize extra latency incurred inside the critical section.

Leveraging function's caller-callee convention. Our basic context-switch algorithm saves, transfers, and restores all CPU-specific registers. We minimize this overhead by leveraging the function calling convention. Specifically, we explicitly make the slow-path of lock acquisition as a function to prevent compiler inlining. This has two benefits: First, this phase is triggered only in the case of contention, as in the uncontended case, the thread acquires the TAS lock. This approach is similar to the Linux spinlock implementation [41]. Second, we leverage the function calling convention. In particular, we save, transfer, and restore only the callee-saved registers, while the compiler saves and restores the necessary caller-saved registers [3]. The compiler, using its register liveness information, knows exactly which caller-saved registers are in use when the slow-path function is called, and spills only those registers to the stack. Moreover, the number of callee-saved registers is small [1, 8, 10]. For example, with x86_64, there are only six callee-saved registers compared with 16 general-purpose registers. Thus, the combiner only transfers at most one cache line to encapsulate any critical section of arbitrary length.

Prefetching thread-local data. Accessing thread-local data within the critical section requires moving the waiter's specific code and data residing on its CPU to the combiner's CPU. Unfortunately, this movement extends the length of the critical section. We find that most of the local data resides on the stack due to aggressive compiler optimizations that a thread either accesses or modifies in the critical section. Thus, we prefetch contiguous cache lines from the top of the stack[2] to minimize the critical section latency. The combiner issues the prefetch requests before executing the current waiter's critical section. Traditional lock designs cannot adopt this approach because, unlike the combining approach, the lock holder already has access to its local data. Meanwhile, shared data cache lines are always going to move to the CPU holding the lock, and their movement is only possible at the end of the critical section.

3.6.3 NUMA awareness

Similar to ShflLock, TCLock can adopt different policies to choose the next waiter (`select_next_waiter()` in §3.2). TCLock currently employs a NUMA-aware policy that minimizes cache-line bouncing among NUMA nodes. In particular, the combiner only executes critical sections of waiters

[2]We prefetch data in the write mode using the `prefetchw` instruction.

belonging to the same NUMA node. Upon reaching the combining limit, the combiner passes the role to a waiter residing on another NUMA node. TCLock adopts the dynamic queue-splitting approach from the CNA algorithm. Specifically, TCLock maintains a combiner-local waiting queue for remote NUMA node waiters and uses the primary queue for local NUMA node waiters. Initially, all waiters join the primary queue (same as before), and the selection of the next waiter happens as follows: The combiner tries to find the next waiter from the same socket. If it succeeds, it executes that waiter's critical section; otherwise, it adds the remote waiter's node to its local queue. While passing the role, the combiner first enqueues the local queue waiters at the beginning of the primary queue and then passes the combining role to the primary queue head.

4 TCLocks with Real-World Applications

Although TCLocks offer a compelling case to minimize overall shared data movement, applying them to real-world applications introduces two major challenges. The first challenge involves fine-grained locking, which requires support for multi-level locking [4, 20, 28] and out-of-order unlocking [12, 13]. The second challenge stems within the OS kernel, such as Linux, in which locks can be acquired within special execution contexts, which guarantees stability about per-CPU variables while executing within these contexts. We now discuss our approaches to overcoming these challenges in the context of the Linux kernel.

4.1 Multi-level Locking

Multi-level locking leads to two notable usage patterns that require additional effort to design and implement correctly in the context of combining. First, a combiner thread can be a waiter while executing a nested lock (henceforth called *waiter-combiner*). Second, locks can be released in arbitrary order leading to out-of-order unlocking. Although out-of-order unlocking does not affect traditional locks, TCLock requires extra care in handling such cases, as it can lead to data corruption as well as deadlocks. We now present our approach to supporting these usage patterns.

Nested combining. For handling nested combining, TCLock adopts the same interrupt processing mechanism by OSes. Interrupt handlers, before processing the interrupt, push the current thread state on the stack. When the interrupt handler finishes, it restores the thread's state from the stack. This allows for handling nested interrupts without affecting the execution of the interrupted thread. There are three cases that occur when TCLocks interplay in the context of nested locking: First is the case of nested locks when both locks are in their combining phase. The second one is when the outer level lock is a combiner and the inner one is the fast-path TAS lock. Finally, the third case is the opposite of the second scenario.

Listing 3 shows the changes required to implement the first case, which works similar to the interrupt processing approach. Specifically, a combiner may acquire a nested lock inside the critical section. Before acquiring that lock, the combiner pushes its state onto its ephemeral stack (line 24), which it restores after releasing that nested lock (line 35). This allows TCLocks to handle arbitrary levels of nesting without violating application correctness.

```
1  +  G_UNLOCKED_OOO = 4
2  # Extra state to handle out-of-order unlocking
3
4  def switch_stack(lock, cst):
5     switch_to_ephemeral_stack(cst.node) # Main → ephemeral
6  -  lock_slowpath(lock, cst)
7  +  ret_val = lock_slowpath(lock, cst)
8  +  if ret_val == G_UNLOCKED_OOO: # Only for nested combiner
9  +    switch_to_combiner_previous_stack_frame()
10 +  else:
11 +    switch_from_ephemeral_stack(cst.node) # Ephemeral→main
12
13 def lock_slowpath(lock, cst):
14   ...
15   if qprev is not None: # Waiters are in the queue
16     ...
17     if qnode.request == PRCSD : # Waiter rqst is processed
18 +      if lock.glock == G_UNLOCKED_OOO:  # OOO unlock
19 +        return G_UNLOCKED_OOO
20       return 0 # Combiner executed my CS
21   ...
22
23   # For handling nested combining
24 +  save_on_curr_stack_frame(lock.glock, cst.qcurr, qnode.rsp)
25
26   lock.glock = G_LOCKED_COMBINER
27 +  j = find_first_empty_index(cst.lock_addr)
28 +  cst.lock_addr[j] = lock  # Record the current lock address
29   # Combiner loop ...
30   # Combiner phase is over, now combiner will run its CS
31 +  cst.lock_addr[j] = None  # Remove the current lock address
32
33   # For handling nested combining
34 +  restore_from_curr_stack_frame(lock.glock,
35 +                               cst.qcurr, qnode.rsp)
36   notify_next_queue_head(qnext) # Next waiter is the combiner
37
38 def spin_unlock(lock):
39   if lock.glock == G_LOCKED:
40     lock.glock = G_UNLOCK # Only true for no combining phase
41     return
42 +  max_idx = find_last_not_empty_index(cst.lock_addr)
43 +  my_idx = find_my_lock_index(cst.lock_addr, lock)
44 +  if my_idx < max_idx :  # Acquire order != release order
45 +    lock.glock = G_UNLOCKED_OOO
46 +    return
47   # Jump back to combiner
48   cst = this_cpu_komb_struct()
49   # Waiter's stack → combiner's ephemeral stack
50   switch_stack_from_waiter_to_combiner(cst.qcurr, cst)
51   return
```

Listing 3: Out-of-order (OOO) unlocking protocol.

TCLocks also supports the last two scenarios, in which one of the locks is in the combining phase. We do not require any additional support for these cases because each lock is independent and the underlying lock mechanism doesn't interact with each other, which is exactly the same in the traditional lock design. Specifically, every lock has its own lock word and its underlying lock mechanism only interacts with its own lock word. Thus, acquiring the lock in the fast-path (TAS lock), does not interact with the lock which is held by the combiner thread.

Out-of-order (OOO) unlocking. The algorithms discussed thus far for TCLocks incorrectly handle OOO unlocking, leading to wrong program execution. We illustrate this through an example: Suppose multiple threads acquire L_A and then L_B, which leads to contention. As a result, the combining phase becomes active, and C_A and C_B hold locks L_A and L_B, respectively. C_A becomes a waiter-combiner when it tries to acquire lock L_B. Now, if L_A is released before L_B—unlocked OOO—the combiner C_B will return to its own combining loop, as the unlock function does not track of the order of unlocked locks. Therefore, the combiner C_B breaks application semantics by starting to execute the next waiter's critical section, while the lock L_B which it holds is not unlocked yet.

To handle OOO unlocking, we rely on a simple insight: we can release a lock at a later point in time without affecting the correctness of the application. Specifically, we do not release L_A; we only release it once L_B has been released. This approach is similar to the handling of nested transactions, in which we effectively flatten the out-of-order lock hierarchy and release all the locks at the same time.

Listing 3 shows the changes required to TCLocks for handling OOO unlocking. We make three specific changes in the lock and unlock function of TCLocks:

- To identify OOO unlocks, we maintain a per-thread `lock_addr` array to record the acquisition order of locks. Before starting the combining loop, a combiner stores its lock's address in the `lock_addr` array (line 28), and removes it once the combiner loop finishes (line 31).
- In the unlock function, the combiner checks if the lock that is being unlocked is the last entry in the `lock_addr` array. This is because the last entry in the `lock_addr` array is the lock holding which the current combiner is executing the critical section. We have two cases now: The first one is the non-OOO case (line 47): The combiner follows the original algorithm and returns to the combiner's ephemeral stack (line 50) and continues with its combiner loop. While the second one is an OOO case (line 44): We simply mark the lock as *OOO-unlocked* (line 45) and let the current combiner continue executing until the unlock function for its lock is called. The waiter-combiner for the OOO-unlocked lock waits for the notification from the current combiner which doesn't happen until the current combiner reaches its combiner loop. Therefore, delayed notification effectively flattens the lock hierarchy for out-of-order unlocked locks as the waiter-combiner cannot progress until it gets the notification.
- After receiving the notification, the waiter-combiner checks if its lock is unlocked OOO (line 18) and if true, it return G_UNLOCKED_OOO (line 19). The combiner switches to its previous state (line 9) which was saved when the nested lock was called (line 5). The combiner returns to its combiner loop (line 29), notifies the current waiter and continues combining the next waiter.

The waiter for the outermost lock will get the control back once the outermost lock and all the nested locks are released. The waiter then starts executing its non-critical section.

4.2 Special Execution Contexts and Per-CPU Variables

Delegation via transparent combining breaks assumptions of Linux kernel code about the stability of access to per-CPU variables under special execution contexts. This includes interrupt handlers, non-preemptible contexts, non-migratable contexts, etc. This raises a critical question for our design: How do we enable delegation-style locking transparently in the kernel without compromising on correctness?

A potential solution involves the combiner accessing the per-CPU variables of the waiter's CPU while executing the critical section. For example, on x86, we can save and restore the `gs` registers that allows access to per-CPU variables of the waiter's CPU [14]. Unfortunately, this approach leads to data races when waiters are busy-waiting, as interrupts on the waiter's CPU may still access per-CPU variables. Moreover, this approach further leads to additional overhead of accessing per-CPU data of a remote CPU. Besides that, it requires annotating parts of the kernel code that access per-CPU variables for functional correctness, such as scheduler [22], RCU [18], and many more during the combining phase. As a result, these challenges make it very difficult to enable transparent combining in special execution contexts within the kernel.

We adopt a more conservative approach of disabling combining for such execution contexts and falling back to default kernel locking (currently qspinlock [41]). We leverage the property that any part of a critical section requiring stable access to per-CPU variables ensures appropriate protection against CPU migration for that region of code. For example, the Linux kernel's `spinlock_t` APIs do not guarantee stable access to per-CPU variables, as they do not disable preemption for the critical section. This is because the `spinlock_t` type is transparently replaced with a mutex on real time kernels [19]. Hence, the scheduler is allowed to preempt threads and migrate them to a different CPU when they are holding such a lock. To ensure that preemption is disabled within the critical section regardless of the kernel configuration, developers use specific `raw_spinlock_t` APIs [17].

When invoking the TCLock APIs, we only enable combining for threads that can migrate from one CPU to another. Otherwise, we disable combining and fallback to the existing traditional lock. We identify these code regions by leveraging well-defined APIs of the Linux kernel. In particular, we enable combining for the following cases: 1) the kernel thread executes in the *task* context [27]; 2) it does not disable *migration* or *preemption* [9, 16], and 3) it does not execute in a context where HardIRQs and SoftIRQs are disabled [5, 24]. It is safe to execute traditional and combining queue-based lock because mechanism for both types of locks are independent and only one of them will be active for a particular instance of lock at any given point.

5 Implementation

We implement TCLocks in the Linux kernel v5.14.19 and replace all *spinlock*, *mutex*, and *rwsem*. We add 1349, 955, and 1652 LoC for spinlock, mutex, and readers-writer lock (rwsem), respectively. For userspace applications, we use LiTL [50] library. It uses the `LD_PRELOAD` mechanism to interpose different POSIX locks used by userspace applications.

We implement TCLock for the x86 architecture, but it is easily extensible to other architectures as well. x86-64 has six callee-saved registers: `rbx`, `rbp`, and `r12–r15`. We push these registers on the stack along with the stack pointer on the waiter's `qnode`. When the combiner switches to the waiter's main stack, it uses the stored stack pointer and pops the callee-saved registers from the stack. We mark the stack-switch function as *noinline* and *noipa* to prevent any compiler optimizations and function inlining. Our code is publicly available here: `https://github.com/rs3lab/TCLocks`.

6 Evaluation

We evaluate TCLocks by answering the following questions:

Q1. How does the kernel-based TCLock implementation impact micro-benchmarks (§6.1) and real applications (§6.2)?

Q2. How does each design decision affect TCLocks' performance (§6.3)?

Q3. How does the userspace TCLock implementation impact an application's performance (§6.4)?

Evaluation setup. We use micro-benchmarks that mostly stress a lock and application benchmarks that stresses various kernel subsystems. In addition, we use a hash-table nano-benchmark [69] to show the effectiveness of TCLock design decisions. We evaluate on an 8-socket, 224-core Intel machine with hyper-threading disabled. We use `tmpfs` in all experiments to minimize the file system overhead. We evaluate three traditional locks within the Linux kernel: Linux's stock locks, CNA, and ShflLock. CNA replaces the stock qspinlock, while we replace all locks with ShflLock.

6.1 TCLock Performance Comparison

We evaluate TCLocks using a set of micro-benchmarks [31, 62]. Each micro-benchmark instantiates a set of threads and pins them to cores. These threads mostly contend on a single lock (sometimes two) while performing specific tasks (Table 1) for 30 seconds.

Spinning TCLock. Figure 3 ((a) and (b)) show that TCLock^{SP} outperforms the Linux version (Stock) by 3.7× and 4.4× on MRDM and lock1, respectively. TCLock performs similarly to Stock from two to eight cores for two reasons. First, the stack switching adds an average of 47 ns latency. Second, the combiner is unable to perform effectively at such a low core count. As a result, TCLock does not reach its potential. On the other hand, the benefit of TCLock^{SP} is evident after eight cores where the gains from localizing shared data cache lines outweighs the overhead of stack-switch. TCLock^{SP} combiner on average batches 950 waiter's request. Thus, even within a socket (up to 28 cores), TCLock^{SP} maintains consistent throughput.

Lock type	Workload	Lock: Usage
Spinlock	MRDM [62] lock1 [31]	**rename seqlock**: Rename files within a directory **files_struct.file_lock**: fd allocation / fcntl
Blocking	MWRM [62]	**sb->s_vfs_rename_mutex**: Rename a file across directory **dentry->d_lock**: Dentry lock
RW Blocking	mmap1 [31]	**mm_struct->mmap_lock**: Memory map file within a directory

Table 1: Lock usage in various micro-benchmarks [31, 62].

Compared to NUMA-aware locks, TCLock^{SP} outperforms ShflLock and CNA by 2-3× across sockets. The combining-based NUMA-aware policy of TCLock^{SP} minimizes the cache-line bouncing of both the lock word and the shared data. On average, 190K combiners execute during a 30-second run where every TCLock^{SP} combiner batches ∼980 waiter's request before passing the lock to different NUMA socket. In essence, every combiner is reducing extra coherence traffic for accessing shared data within the waiter's critical section, which would be generated in a traditional lock design.

Blocking TCLock. We compare TCLock^{B} with Linux `mutex` and ShflLock. Figure 3 (c) shows that TCLock^{B} is 1.8× faster than both `mutex` and the blocking version of ShflLock. Both Stock and ShflLock suffer from shared data movement at a high core count. In addition, ShflLock's performance degrades similarly to that of Stock due to its lock stealing, which renders its NUMA-policy ineffective. Whereas, TCLock^{B} retains performance because it reduces cache-line bouncing for both the lock word and shared data.

Readers-Writer Blocking TCLock. Figure 3 (d) shows the impact of TCLock when stressing the writer side of `rwsem`. We use the mmap1 benchmark [31], which populates and deletes VMAs within an address space. TCLock maintains the best throughput irrespective of contention after eight cores. Within a socket, TCLock^{RW} outperforms Stock by 1.7×, as a combiner combines ∼1000 waiter's request, thereby minimizing cache-line movement of shared data cache lines. Moreover, across the socket, TCLock^{RW} combiner batches similar number of waiter's request resulting in 3.1× and 1.5× better throughput than Stock and ShflLock.

6.2 Application-level Benchmarks

We evaluate two applications that extensively stress various subsystems of the Linux kernel. Figure 4 reports applications' throughput. The kernel subsystem uses a mix of blocking locks and spinlocks, which are present in several data structures such as inodes, task structures, and memory mappings.

Psearchy is a parallel version of searchy that does text indexing. It is mmap intensive, which stresses the memory subsystem with multiple userspace threads. It does around 96,000

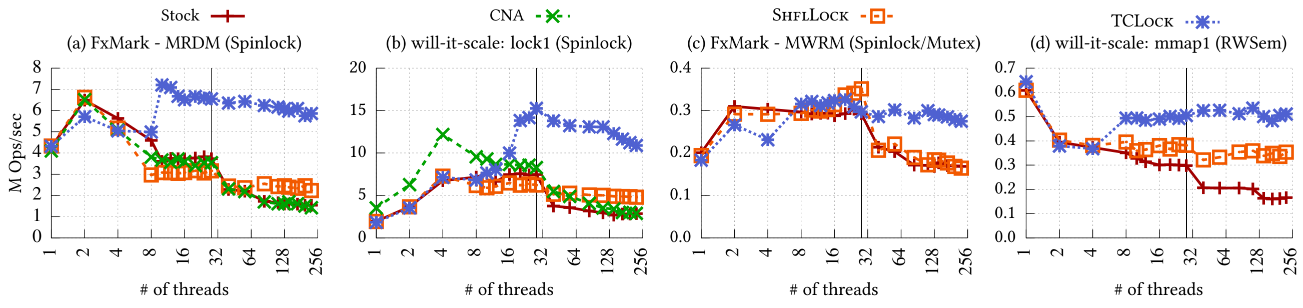


Figure 3: Impact of spinlock, blocking locks and read-write semaphore on the scalability of micro-benchmarks [31, 62].

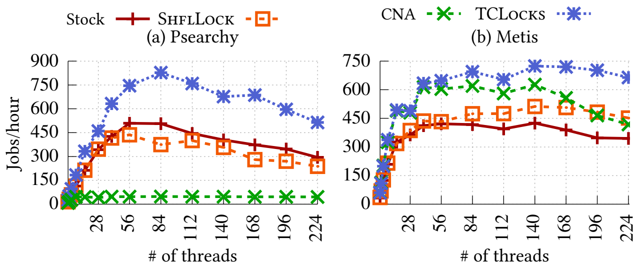


Figure 4: Impact of kernel locks on application scalability.

small and large mmap/munmap operations from 96,000 files with multiple threads. It stresses the writer side of the `rwsem` in the memory subsystem and inode allocation in the file system layer. Figure 4 (a) shows that TCLock outperforms existing locks up to 2.2×. Because of its effective combining strategy, TCLock^{RW} is able to localize access to shared data. We find that ShflLock and Stock have similar performance as they inefficiently use up hardware bandwidth. Moreover, we observe that psearchy's performance drops with increasing core count, which happens due to the contention in the file stream glibc library.

Metis is an in-memory map-reduce framework, representing a page-fault-intensive workload that stresses the readers' side of the `mmap_sem` (`rwsem`) in the Linux kernel. Figure 4 (b) shows that TCLock outperforms both ShflLock and Stock by 1.3×. The reason is due to the phase-based design of TCLock^{RW}, which improves the performance by batching the writers in one phase, meanwhile executing readers in parallel in the next phase. Across sockets, it improves performance compared to ShflLock and Stock by 1.7× and 1.4× at 140 cores, respectively.

6.3 Nano benchmark: RCUHT

We now do an in-depth analysis of TCLocks using a hash-table benchmark in the kernel [69]. A global lock guards the hash table. For TCLock^{SP} and TCLock^{B}, we generate 100% writes, whereas for TCLock^{RW} (readers-writer blocking lock), we generate 1% and 20% writes on the hash table. Figure 5 presents the results and the factor analysis of TCLocks.

Spinning TCLock. Figure 5 (a) and (b) shows the throughput and 99.99% latency of spinlocks, respectively. (a) We find that TCLock^{SP} maintains similar performance within and across sockets because of the effective combining policy. In particular, the combining batches up to 50,000 waiter requests, thereby localizing the requests for that many invocations. In addition, the average and 99%ile latency of the critical section is 188 ns and 474 ns at 28 cores, respectively, whereas both stock and ShflLock have up to 2.5× and 2.1× higher average and 99%ile latency, respectively.

In the case of NUMA, TCLock^{SP} outperforms ShflLock and Stock by up to 9.4×. The improvement occurs because of minimizing cache-line bouncing and Localizing shared data, which reduces the time spent in critical section. For example, at 168 cores, the average latency of TCLock is 213ns, which is similar to average latency at 28 cores. Whereas ShflLock and Stock have 10.5× and 11× higher latency, respectively. The 99%ile latency increases to 1516 ns for TCLock. This happens because of NUMA-aware moving of the shared data, which increases the 99.9% latency of TCLock. However, this latency is still 3× lower than that of ShflLock.

Figure 5 (b) shows the combined latency for the lock function, critical section execution, and the unlock function. TCLock^{SP}, even with batching 50,000 waiters, has up to 5.6× and 4.4× lower latency compared to ShflLock and Stock, respectively. This is because of lower critical section latency, which reduces the overall latency of the whole system as the critical sections are executed sequentially.

Nested Locking and OOO unlocking. We evaluate the impact of our OOO unlocking with a hash-table nanobenchmark that acquires nested lock and can release locks in an OOO manner. Specifically, every bucket has a lock and nested locks are acquired when moving an entry from one bucket to another. Figure 5 (c) shows that, within a socket, TCLock^{SP} performs similar to other locks. At 28 cores, TCLock^{SP} is 5% slower than Stock. This is because the overhead of saving/restoring the combiner state with nested lock along with delaying the unlock degrades TCLock^{SP} performance within a socket. Across socket, TCLock^{SP} outperforms Stock by up to 3.7×. The performance gains with localization of shared cache lines outweighs the overhead of TCLock^{SP} implementation of nested locking.

Blocking TCLock. Figure 5 (d) shows the throughput with blocking locks. TCLock^{B} outperforms stock in both under-subscribed and over-subscribed scenarios. With the help of its efficient spin-then-park strategy, TCLock^{B} outperforms Stock by up to 9.5× in under-subscribed scenarios. More-

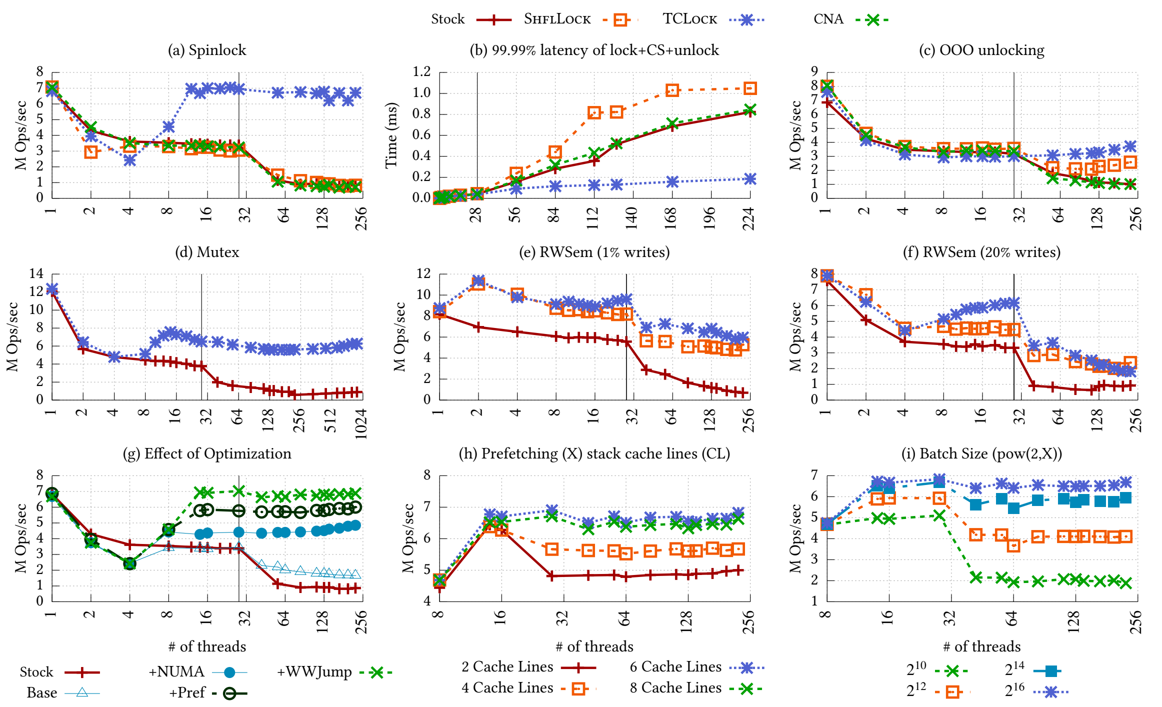


Figure 5: (a – f) Impact of spinning, blocking and read-write lock on the hash-table nano benchmark with an eight-socket Intel machine. (b) Latency of executing lock function+CS+unlock function with different spinlocks. (c) Performance with nested locking and OOO unlocking. (g) Impact of different optimization introduced in TCLock^{SP}. On top of baseline, we add NUMA-awareness, stack prefetching, and waiter-to-waiter jump. (h – i) Impact of prefetching and batch size on TCLock^{SP}'s performance.

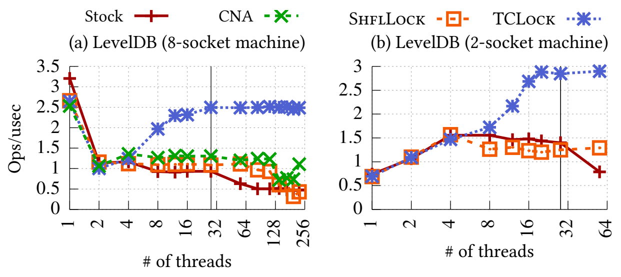


Figure 6: Impact of locks on userspace applications.

over, TCLock^{B} maintains the same performance even after crossing the socket boundary. Although latency to wake up a waiter on a remote socket costs more than that of the local socket, TCLock^{B}'s usage of NUMA-aware design amortizes the overhead of waking up waiters from other socket.

Reader-writer TCLock. Figure 5 ((e) and (f)) show that the TCLock^{RW} has higher throughput than the stock version by 6.8× and 2.2× for 1% and 20% writes, respectively. ShflLock and TCLock^{RW} use similar design for readers. Because of using the phase-based design, TCLock^{RW} is able to improve performance by up to 1.28× and 1.37× at 1% and 20% writes, respectively. We further observe that combining is not effective with centralized readers counting, as the readers counter cache line is always moving across cores.

TCLocks optimizations. Figure 5 (g) shows the effect of different optimizations used by TCLock. TCLock^{SP} without any optimizations outperforms Stock by 2.2× because it localizes the shared data access. The overhead of stack switch is apparent at lower core count because jumping to a waiter's critical section requires access to the stack which needs to be fetched from a waiter's core. On adding NUMA-awareness to the current design, we improve the performance by 2.6×, as we now prevent moving the waiter's stack cache line across sockets. It also helps within a socket because checking the socket ID of the next waiter's node fetches the next waiter's node in the combiner's cache. As a result, this simple check reduces the time spent in the combiner loop.

In addition, our stack prefetching approach, on top of NUMA-awareness policy, further improves performance by 1.3×, as it reduces the time spent in starting the execution of critical section. Finally, our waiter → waiter jump (WWJump) further improves the throughput by 1.2× as it reduces the overhead of stack switch (∼50 ns) from two switches to one. Overall, our optimizations reduce the overhead of stack switching and improve performance compared to the baseline by 4×.

TCLocks sensitivity. Figure 5 ((h)–(i)) shows the impact of changing the number of prefetched cache lines and the number of waiter's combined. Figure 5 (h) shows that prefetching up to six cache lines provides the best performance for this benchmark. It depends entirely on what is accessed inside the critical section. We can write a compiler pass to tune this parameter, as the compiler has the information on what is accessed within the critical section. Figure 5 (i) shows the impact of batching. Higher batch count improves throughput at the expense of short-term fairness, but TCLocks maintain long-term fairness. Batching is also able to reduce the latency for all requests, if it can reduce the time spent per request as shown in Figure 5(e).

6.4 Performance With Userspace TCLock

We evaluate TCLocks on the LevelDB benchmark [49]. We integrate both TCLock, CNA and ShflLock into LiTL [50] for evaluation. LevelDB is an open-source key-value store [48]. We use the `readrandom` benchmark with 1M key-value pairs, that contends on the global database lock. Figure 6 (a) shows the performance with spinlocks on an 8-socket machine. Within a socket, TCLock^{SP} improves throughput compared to other locks by 1.9×–2.6×. Localizing shared data movement helps to achieve better performance than traditional locks. Across sockets, NUMA-awareness coupled with minimal shared data movement helps TCLock^{SP} outperform other locks by up to 5.2×. Figure 6 (b) shows the performance on a 2-socket machine. TCLock^{SP} performs similar to the 8-socket machine and improves throughput compared to other locks by 2.1×–3.6×.

7 Discussion and Limitations

TCLocks implement transparent delegation, which enables developers to use delegation-style locking without rewriting the application. However, TCLocks have limitations both in terms of algorithm design and kernel implementation. We discuss them below.

Overhead at two–four cores. We observe overhead with TCLocks when very few threads (two–four) contend for a lock. Contending threads execute slowpath after stack-switching, but combining is only enabled when more than two waiters are present in the queue. Waiters pay the cost of two stack-switching but their critical section is not executed by the combiner. This can be solved by disabling combining when we have less than four threads in the queue. The challenge lies in efficiently identifying the size of the queue without using extra memory or traversing the queue.

Resource accounting. The kernel requires accurate accounting of resources like CPU usage, allocated memory, etc. Kernel subsystems, such as the scheduler or cgroup, are guided by the accounting of resources used by a particular thread. Delegation-based techniques can break this accurate accounting for resources used within the critical section. Thus, TCLocks complicate resource accounting. Even though a combiner thread executes the critical section on behalf of other waiter threads, resources like CPU time or allocated memory in the critical section need to be accounted to the waiter thread, for maintaining broader kernel semantics. We leave this extension as future work.

TCLock vs 'current'. Apart from per-CPU variables, Linux also uses a macro named *current*, which resolves to a per-CPU pointer variable to the currently executing thread's task structure. This pointer is used to access the task structure for multiple purposes, including but not limited to resource accounting with cgroups [21], permission checks using credentials [15], thread scheduling [26], etc. While executing a waiter's critical section on the combiner's CPU, if this pointer is not switched to the waiter's task structure, then it could lead to subtle bugs. For example, if a combiner thread has higher privileges than the waiter thread, and the permission checks are done within the critical section, it may lead to privilege escalation bugs, since the combiner thread's credentials will be inspected.

One possible solution is to modify *current* macro's implementation to resolve to the waiter's task structure while executing waiter's critical section on combiner CPU. Unfortunately, this will also lead to bugs. For example, if a thread sleeps within its critical section, the scheduler code uses the *current* macro to put the running task to sleep. When combining a waiter's critical section, we want the combiner thread to sleep. However, if we switch the *current* macro to use the waiter's task structure, it will lead to confusion within the scheduler as the waiter task is already seen to be running on another CPU.

We currently keep the `current` macro unchanged, and suggest judicious use of the TCLock APIs in cases where a different thread identity within the critical section may lead to unexpected behavior.

8 Conclusion

Delegation based techniques are known to offer better scalability and provide better performance for highly contended scenarios, but prior work requires application changes to enable delegation. In this paper, we propose a new technique called *transparent delegation* that makes delegation-style locking practical. We design the first-ever transparent delegation based locks, called TCLocks, for both userspace applications and the Linux kernel. This is achieved by lightweight context switching and using ephemeral stacks to maintain consistency. Using transparent delegation, we design spinning, blocking and phase-based readers-writer locks. We replace all the locks in the Linux kernel with TCLocks, and discuss the technical challenges involved. Our evaluation shows that TCLocks provide better performance and scalability compared to traditional lock design.

9 Acknowledgment

We thank Changwoo Min for his comments on the initial draft. We also thank Dave Dice, Alex Kogan, the anonymous reviewers, and our shepherd, Geoffrey M. Voelker, for their helpful feedback. This work is supported by the SNSF project grant 212884.

References

[1] List of callee-saved registers. https://developer.arm.com/documentation/102374/0100/Procedure-Call-Standard, . [Accessed on 22/04/2023].

[2] List of arm registers. https://developer.arm.com/documentation/dui0473/m/overview-of-the-arm-architecture/arm-registers, . [Accessed on 22/04/2023].

[3] Gcc calling convention. https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html. [Accessed on 22/04/2023].

[4] Lock ordering for file mmap. https://elixir.bootlin.com/linux/v6.1/source/mm/filemap.c#L72. [Accessed on 22/04/2023].

[5] Locking Between Hard IRQ and Softirqs/Tasklets: Unreliable Guide To Locking — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html#locking-between-hard-irq-and-softirqs-tasklets, . [Accessed on 30/04/2023].

[6] Hardirq. https://www.kernel.org/doc/htmldocs/kernel-locking/hardirq-context.html, . [Accessed on 22/04/2023].

[7] Hardware interrupts (hard irqs). https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-hardirqs.html, . [Accessed on 22/04/2023].

[8] List of x86-64 registers. https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start. [Accessed on 22/04/2023].

[9] [PATCH 7/9] sched: Add migrate_disable(). https://lwn.net/ml/linux-kernel/20200921163845.769861942@infradead.org/. [Accessed on 30/04/2023].

[10] List of mips registers. https://en.wikibooks.org/wiki/MIPS_Assembly/Register_File. [Accessed on 22/04/2023].

[11] Non-maskable interrupt. https://en.wikipedia.org/wiki/Non-maskable_interrupt. [Accessed on 22/04/2023].

[12] Dentry cache spinlock unlocked out-of-order, . https://elixir.bootlin.com/linux/v6.0/source/fs/dcache.c#L3022.

[13] Pipe mutex unlocked out-of-order, . https://elixir.bootlin.com/linux/v6.0/source/fs/splice.c#L1552.

[14] Per-cpu variables. https://docs.kernel.org/core-api/this_cpu_ops.html#inner-working-of-this-cpu-operations, . [Accessed on 22/04/2023].

[15] Credentials in Linux. https://www.kernel.org/doc/Documentation/security/credentials.txt, . [Accessed on 30/04/2023].

[16] Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe. https://www.kernel.org/doc/Documentation/preempt-locking.txt. [Accessed on 30/04/2023].

[17] raw_spinlock_t: Lock types and their rules — The Linux Kernel documentation. https://docs.kernel.org/locking/locktypes.html#raw-spinlock-t. [Accessed on 30/04/2023].

[18] What is RCU? – "Read, Copy, Update" — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html. [Accessed on 30/04/2023].

[19] spinlock_t and PREEMPT-RT: Lock types and their rules — The Linux Kernel documentation. https://docs.kernel.org/locking/locktypes.html#spinlock-t-and-preempt-rt. [Accessed on 30/04/2023].

[20] Lock ordering for directory rename. https://docs.kernel.org/filesystems/directory-locking.html. [Accessed on 22/04/2023].

[21] Control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. [Accessed on 30/04/2023].

[22] CFS Scheduler — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/scheduler/sched-design-CFS.html#few-implementation-details. [Accessed on 30/04/2023].

[23] Softirq. https://www.kernel.org/doc/htmldocs/kernel-hacking/basics-softirqs.html, . [Accessed on 22/04/2023].

[24] Locking Between User Context and Softirqs: Unreliable Guide To Locking — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html#locking-between-user-context-and-softirqs, . [Accessed on 30/04/2023].

[25] Task context. https://www.kernel.org/doc/htmldocs/kernel-hacking/basic-players.html#basics-usercontext. [Accessed on 22/04/2023].

[26] CFS Scheduler — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/scheduler/sched-design-CFS.html. [Accessed on 30/04/2023].

[27] User Context: Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.16/kernel-hacking/hacking.html#user-context. [Accessed on 30/04/2023].

[28] Lock ordering in memory management subsystem. https://elixir.bootlin.com/linux/v6.1/source/mm/rmap.c#L20. [Accessed on 22/04/2023].

[29] Windows fibers. https://learn.microsoft.com/en-us/windows/win32/procthread/fibers.

[30] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[31] A. Blanchard. will-it-scale. https://github.com/antonblanchard/will-it-scale. [Accessed on 22/04/2023].

[32] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[33] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, Oct. 2010.

[34] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[35] B. B. Brandenburg and J. H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 184–193. IEEE, 2009.

[36] B. B. Brandenburg and J. H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. In *Real Time Systems*, pages 184–193, 2011. doi: 10.1109/ECRTS.2009.14.

[37] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97. Springer, 2013.

[38] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 157–166, Shenzhen, China, Feb. 2013.

[39] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Francisco, CA, Feb. 2015.

[40] J. Corbet. Big reader locks, 2010. https://lwn.net/Articles/378911/, [Accessed on 30/04/2023].

[41] J. Corbet. MCS locks and qspinlocks, 2014. https://lwn.net/Articles/590243/, [Accessed on 30/04/2023].

[42] T. Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, Department of Computer Science, University of Washington, 1993.

[43] D. Dice and A. Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6281-8.

[44] D. Dice and A. Kogan. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 173–183, 2021.

[45] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, 2011.

[46] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–256, New Orleans, LA, Feb. 2012.

[47] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 257–266, New Orleans, LA, Feb. 2012.

[48] S. Ghemawat and J. Dean. LevelDB, 2019. URL https://github.com/google/leveldb. [Accessed on 30/04/2023].

[49] R. Guerraoui, H. Guiroux, R. Lachaize, V. Quéma, and V. Trigonakis. Lock—Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.*, 36(1):1:1–1:149, Mar. 2019. doi: 10.1145/3301501. URL http://doi.acm.org/10.1145/3301501.

[50] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 649–662, Denver, CO, June 2016.

[51] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[52] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[53] X. Leroy. The open group base specifications issue 7, 2016. http://pubs.opengroup.org/onlinepubs/9699919799/, [Accessed on 30/04/2023].

[54] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, 2009.

[55] Linux. Lock ordering, 2013. URL https://elixir.bootlin.com/linux/latest/source/mm/filemap.c#L66. [Accessed on 30/04/2023].

[56] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, Jan. 2016.

[57] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, 2006.

[58] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE, 1994.

[59] C. D. Marlin. *Coroutines: a programming methodology, a language design and an implementation.* Number 95. Springer Science & Business Media, 1980.

[60] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. pages 21–65, Feb. 1991.

[61] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-writer Synchronization for Shared-memory Multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 106–113, 1991.

[62] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[63] O. Nesterov. Linux percpu-rwsem, 2012. http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h, [Accessed on 30/04/2023].

[64] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, jul 1999.

[65] S. Park, D. Zhou, Y. Qian, I. Calciu, T. Kim, and S. Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.

[66] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1871-0.

[67] S. Roghanchi, J. Eriksson, and N. Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 342–358, 2017.

[68] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, Salt Lake City, UT, Feb. 2001.

[69] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 11–11, Portland, OR, June 2011.

[70] A. Viro. parallel lookups, 2016. https://lwn.net/Articles/684089/, [Accessed on 30/04/2023].

[71] P. Zijlstra. percpu rwsem -v2, 2010. https://lwn.net/Articles/648914/, [Accessed on 30/04/2023].