Scalable and Practical Locking with Shuffling

Sanidhya Kashyap   Irina Calciu*   Xiaohe Cheng‡   Changwoo Min†   Taesoo Kim

Georgia Institute of Technology   **VMware Research*   *‡HKUST*   *†Virginia Tech*

Abstract

Locks are an essential building block for high-performance multicore system software. To meet performance goals, lock algorithms have evolved towards specialized solutions for architectural characteristics (*e.g.*, NUMA). However, in practice, applications run on different server platforms and exhibit widely diverse behaviors that evolve with time (*e.g.*, number of threads, number of locks). This creates performance and scalability problems for locks optimized for a single scenario and platform. For example, popular spinlocks suffer from excessive cache-line bouncing in NUMA systems, while scalable, NUMA-aware locks exhibit sub-par single-thread performance.

In this paper, we identify four dominating factors that impact the performance of lock algorithms. We then propose a new technique, *shuffling*, that can dynamically accommodate all these factors, without slowing down the critical path of the lock. The key idea of shuffling is to re-order the queue of threads waiting to acquire the lock in accordance with some pre-established policy. For best performance, this work is done off the critical path, by the waiter threads. Using shuffling, we demonstrate how to achieve NUMA-awareness and implement an efficient parking/wake-up strategy, without any auxiliary data structure, mostly off the critical path. The evaluation shows that our family of locks based on shuffling improves the throughput of real-world applications up to 12.5×, with impressive memory footprint reduction compared with the recent lock algorithms.

CCS Concepts • **Software and its engineering → Mutual exclusion**.

Keywords mutual exclusion, memory footprint, Linux.

ACM Reference Format:
Sanidhya Kashyap Irina Calciu Xiaohe Cheng Changwoo Min Taesoo Kim. 2019. Scalable and Practical Locking with Shuffling. In *ACM SIGOPS 27th Symposium on Operating Systems and Principles (SOSP'19)*. ACM, New York, NY, USA. 14 pages. https://doi.org/10.1145/3341301.3359629

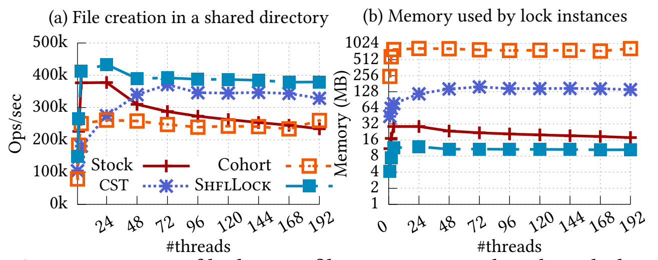

Figure 1. Impact of locks on a file-system micro-benchmark that spawns threads to create new files in a shared directory (`MWCM` [39]). A process stresses the writer side of the readers-writer lock. We evaluate the Linux baseline version (*Stock*), CST [27], Cohort lock [18], and our proposed ShflLock. (a) File creation throughput on an 8-socket 192-core machine. (b) Total memory consumed by locks that are part of the inode structure.

1 Introduction

The introduction of multicore machines marked the end of the "free lunch"[47], making concurrent programming, especially lock-based mutual exclusion, a critical approach to improve the performance of applications. Lock algorithms determine the scalability of applications in multicore machines [3, 5, 21].

Since the invention of concurrent programming, lock design has been influenced by hardware evolution. For instance, MCS [37] was proposed to address excessive cache-line traffic resulting from an increasing number of threads trying to acquire the lock at the same time, while Cohort locks [18] were proposed in response to the emergence of the non-uniform memory access (NUMA) architecture. NUMA machines consist of multiple nodes (or sockets), each with multiple cores, locally attached memories, and fast caches. In such machines, the access from a socket to its local memory is faster than remote access to memory on a different socket [44] and each socket has a shared last-level-cache. Cohort locks exploit this characteristic to improve application throughput.

Unfortunately, the influence of hardware evolution on lock design has resulted in a tight coupling between hardware characteristics and lock algorithms. Meanwhile, other factors have been neglected, such as memory footprint [10], low thread counts, and core over-subscription. For example, Cohort locks can achieve high throughput at high core counts, but also require memory proportional to the number of sockets. The extra memory is unacceptable for some applications, such as databases and OSes, which can have millions of locks. Moreover, Cohort locks have sub-optimal single-thread performance due to using multiple atomic instructions. Figure 1 shows an example of such a scenario: benchmark throughput is affected at lower thread count due to multiple atomic operations (Cohort and CST), and at higher thread count (from four threads) due to the bloated file structure (inode) caused by the large lock memory footprint because inode allocation starts stressing the kernel memory allocator. For example, the size of the inode structure grows by 3.4× with the Cohort lock.

In this paper, we investigate the main dominating factors that impact the scalability of locks and their adoption: 1) cache-line movement between different caches, 2) level of thread contention, 3) core over-subscription, and 4) memory footprint. We find that none of the existing locks perform well on all factors. We propose a new lock design technique called **shuffling** that decouples the lock-acquire/release phases from the lock order policy, and uses lock waiters (*i.e.*, threads waiting to acquire the lock) to enforce those policies, mostly off the critical path. In shuffling, a waiter in the waiting queue takes the role of a shuffler and re-orders the queue of waiters based on the specified policy. This technique gives us the freedom to design and multiplex new policies based not only on the characteristics of fast-evolving hardware, but also on software characteristics. Our new family of locks, called ShflLocks, augment existing locks (TAS and MCS) and use shuffling. Our first lock algorithm is a non-blocking lock that implements NUMA-awareness as a shuffling policy to implement a compact NUMA-aware lock. We further add a core over-subscription policy to implement a blocking lock. We also implement a readers-writer lock on top of our blocking lock. We evaluate our locks in both kernel space and in userspace, and find that our lock algorithms maintain the best throughput regardless of the number of threads contending for the lock. In particular, ShflLocks improve application throughput by 1.2–12.5×, while reducing the memory footprint up to 35.4% and 98.8%, against the currently used Linux kernel locks and against state-of-the-art locks, respectively.

This paper makes the following contributions:

- **Technique.** We propose shuffling, a technique that provides a new mechanism to implement locks with different policies, without increasing lock acquire/release overhead.
- **New lock family.** Based on the shuffling mechanism, we propose a family of ShflLocks: non-blocking, blocking, and blocking readers-writer locks. They are NUMA-aware, have a small memory footprint, and maintain the best performance for varying contention levels.
- **Performance evaluation.** Our evaluation shows that ShflLocks improve application throughput up to 12.5× relative to simple locks, while incurring 13× lower memory overheads compared with state-of-the-art blocking locks.

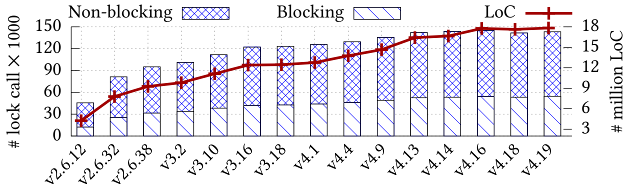

Figure 2. Indirect metric of the growing complexity of lock usage: the number of lock() API calls in the Linux kernel source code.

2 Background and Motivation

We first describe the general evolution of locks and later contrast it with lock evolution inside the Linux kernel.

Lock design. Since the dawn of concurrent programming, hardware has been the dominant factor [13] in the evolution of lock algorithms. For instance, queue-based locks [37] reduce cache traffic relative to test-and-set (TAS) and ticket locks. On NUMA architectures [25, 44], hierarchical locks improve throughput [8, 9, 17, 18, 34] as they amortize remote access cost by physically partitioning a lock into a global lock and per-node locks. Unfortunately, hierarchical locks have two issues: degraded performance for small numbers of cores, and, most importantly, memory overhead. AHMCS [9] and CST [27] partially address one of these issues, but not both concurrently. Our approach is closest to that of CNA [16] and Malthusian locks [14] that reorder an MCS-like queue of waiting threads to improve NUMA performance (CNA) or to block surplus threads (Malthusian). Compared to these locks, shuffling introduces two important innovations. First, in shuffling, queues are reordered by *waiting* threads, rather than lock-holding threads; this keeps the critical path fast and supports a wider range of ordering policies. Second, only waiting threads must maintain queue nodes—lock-holding threads can deallocate them. This simplifies lock deployment and supports important optimizations, such as lock stealing.

We observe a similar evolution in designing readers-writer locks. Mellor-Crummey and Scott [38] proposed variants of readers-writer locks on top of the queue-based locks. However, these locks create coherence traffic in NUMA machines. Calciu *et al.* [6] proposed a per-socket read indicator on top of Cohort locks to localize the reader contention within a socket, but both per-socket or per-CPU [29] approaches require extra memory and are beneficial only in particular cases [11, 42, 51].

Locks in the kernel space. Over the past decade, the Linux kernel has been striving for more concurrency by switching to finer granularity locks. Figure 2 shows the increase in the number of locks as well as their use. One of the most significant goals is to maintain optimal single-thread performance. In addition, lock design must consider: 1) the interaction with the scheduler, 2) the size of the lock structure, and 3) avoiding any explicit memory allocation. These factors have led to sophisticated optimizations. The `spinlock` is the primary locking construct in Linux; it has evolved from TAS to

			Non-blocking locks				Blocking locks		
	Factors		**TAS**	**MCS** [37]	**Cohort** [18]	**CNA** [16]	**`mutex`** [40]	**CST** [27]	**SHFLLOCK**
F1.	*Cache-line movement*		Very high	High	Low	Low	High	Low	Low
F2.	*Thread contention*		🟩🟥🟥	🟧🟩🟥	🟥🟩🟩	🟧🟩🟩	🟩🟩🟥	🟥🟧🟩	🟩🟩🟩
F3.	*Core subscription (U/O)*		-	–	–	–	×/ ✓	×/ ✓	✓/ ✓
F4.	*Memory footprint*	Per lock:	1	8	1,152‡	8	40	1,056‡	12
	(bytes)	Per waiter:	0	12	24	28	32	24	28
		Per lock-holder:	0	12	24	28	0	0	0
Atomic instructions per acquire and release:			1/∞	2/1	4/≈2	2/≈2	1/≈4	≈6/≈3	1/≈2

U: Under-subscribed; O: Over-subscribed. ‡For CST and Cohort locks, we use an eight-socket machine as a reference.

▪ → 1–2 threads ▮ → 1 socket ■ → > 1 socket 🟩 → Optimal throughput 🟧 → Sub-optimal throughput 🟥 → Worst throughput

Table 1. Dominant factors affecting locks that are in use in the Linux kernel or are the state-of-the-art for NUMA architecture. Cache-line movement refers to data movement inside a critical section. Boxes represent the scalability of locks with increasing thread count from one thread to threads within a socket to all threads among multiple sockets. Core subscription is only applicable to blocking locks and denotes the best throughput for a varying number of threads. Both `mutex` and CST are sub-optimal when under-subscribed but maintain good throughput once they are over-subscribed. Memory footprint is the memory allocation for locks: the size of each lock instance (per lock), a queue node required by each waiting thread before entering the critical section (per waiter), and a queue node retained by a lock holder within the critical section (per lock-holder). If the lock holder uses the queue node, which happens for MCS, CNA, and Cohort locks, the thread must keep track of the node, as it can acquire multiple locks: a common scenario in Linux. Note that queue nodes can be allocated on the stack for each algorithm. However, in practice, a lock user needs to explicitly allocate it on the stack for MCS, CNA, and Cohort locks, while `mutex`, CST, and SHFLLOCKS avoid this complexity. We also summarize the number of atomic instructions in the non-contended/contended scenarios.

ticket locks to an MCS variant [32]. The current design is an amalgamation of two locks: a TAS lock in the fast path and an MCS lock in the slow path. The second most widely used lock is the `mutex`, which incorporates a fast path comprising of TAS, an abortable queue-based spinning in mid-path [33], and a parking list per-lock instance in the slow path. Because of the mid-path, along with optimized hand-over-hand locking, `mutex` ensures long-term fairness [33]. The readers-writer semaphore (`rwsem`) is an extension of `mutex` that encodes readers, writers, and waiting readers in an indicator. `rwsem` maintains a single parking list in which both readers and writers are added in the slow path. However, it suffers from severe cache-line movement both when cores are over-subscribed and when they are under-subscribed.

3 Dominating Factors in Lock Design

Locks not only serialize data access, but also add their overhead, directly impacting application scalability. Looking at the evolution of locks and their use, we identify four main factors that any practical lock algorithm should consider. These factors are critical in achieving good performance in current architectures, but their relative importance can vary not only across architectures, but also across applications with varying requirements. Therefore, we should holistically consider all four factors when designing a lock algorithm. Table 1 shows how these factors impact state-of-the-art locks.

F_1. Avoid data movement. Memory bandwidth and the interconnect bandwidth between NUMA sockets are limited, leading to performance bottlenecks when applications incur remote cache traffic or remote memory accesses. Thus, every lock algorithm should minimize cache-line movement and remote memory accesses for both lock structures and data inside the critical section. This movement is quite expensive in NUMA machines: the cost of accessing a remote cache line can be 3× higher than local access [13]. Moreover, for future architectures, even L1/L2 cache-line movements will further exacerbate this cost [41]. Similarly, for readers-writer locks, their readers indicator incurs cache-line movement. *A lock algorithm should amortize data movement from both the lock structure and the data inside the critical section, to hide non-uniform latency and minimize coherence traffic.*

F_2. Adapt to different levels of thread contention. Most multi-threaded applications use fine-grained locking to improve scalability. For example, Dedup and fluidanimate [1] create 266K and 500K locks, respectively. Similarly, Linux has also adopted fine-grained locking over time (Figure 2) and only a subset of locks heavily contend based on the workload [3]. Generally, lock designs optimize either for low contention or for high contention: TAS results in better performance when contention is low, while Cohort locks are a better choice for high contention. Similarly, the scalability of a readers-writer lock is determined by its low-level design and choices, such as using a centralized readers indicator vs. per-socket indicators vs. per-core indicators impact scalability depending on the ratio of readers and writers. *For the best performance in all scenarios, a lock algorithm should adapt to varying thread contention.*

F_3. Adapt to over- or under-subscribed cores. Applications can instantiate more threads than available cores to parallelize tasks, to improve hardware utilization, or to efficiently deal with I/O. In these scenarios, **blocking locks** need to efficiently choose between spinning or sleeping, based on the thread scheduling. Spinning results in the lowest latency, but can waste CPU cycles and underutilize resources while starving other threads, leading to lock-holder preemption [26]. In contrast, sleeping enables threads to run and utilize the hardware resources more efficiently. However, this can result in latency as high as 10ms to wake up a sleeping thread. *Thus, a lock algorithm should consider the mapping between threads and cores and whether cores are over-subscribed.*

F_4. Decrease memory footprint. The memory footprint of a lock not only affects its adoption, but also indirectly affects application scalability. Generally, the structures of a lock are not allocated inside the critical section or on the critical path, so many algorithms do not consider these allocations as a performance overhead. However, in practical applications, locks are embedded inside other structures, which can be instantiated on the critical path. In such scenarios, this allocation aggravates the memory footprint, which stresses the memory allocator, leading to performance degradation. For example, Exim, a mail server, creates three files for each message it delivers. Locks are part of the file structure (inode), so large locks can slow down allocation and directly affect performance [10]. This is even worse for locks that dynamically allocate their structure before entering the critical section [27]. The memory allocation can fail, leading to an application crash. Extra per-task or per-CPU allocations can further exacerbate the issue, *e.g.*, for queue-based locks [12, 24]. Memory footprint also affects readers-writer scalability because the memory consumption dramatically increases for the readers indicators from centralized (8 bytes) to per-socket (1 KB) to per-CPU (24KB) for each lock instance.[1] *Thus, a lock algorithm should consider memory footprint, as it affects both the adoption of the lock and applications performance.*

4 ShflLocks

To adapt to such a diverse set of factors, we propose a new lock design technique, called *shuffling*. Shuffling enables the decoupling of lock operations from a lock policy enforcement, which happens off the critical path. Policies can include NUMA-awareness and efficient parking/wakeup strategies. Using shuffling, we design and implement a family of lock algorithms called ShflLocks. At its core, a ShflLock uses a combination of TAS as a top-level lock and a queue of waiters (similar to MCS). We rely on the shuffling mechanism to enable NUMA-awareness that minimizes cache-line movement (F_1). ShflLocks work well under high contention due to their NUMA-awareness, while maintaining good performance for low contention due to their TAS lock (F_2). Besides NUMA-awareness, we also add a parking/wakeup policy to design an efficient blocking ShflLock (F_3). ShflLocks requires a constant, minimal data structure and does not require additional allocations within the critical section, thereby reducing memory footprint (F_4).

4.1 The Shuffling Mechanism

Shuffling is a new technique for designing locks in which a thread waiting for the lock (the shuffler) re-orders the queue of waiters (shuffles) based on a policy specified by the lock developer. Shuffling is similar to sorting a list with a user-defined comparison function. Here, the list represents a set of waiters and the comparison function is a set of policies, such as NUMA-awareness or a wakeup/parking strategy. This shuffling mechanism is mostly off the critical path because a thread handles the task of policy enforcement while waiting to acquire the lock. Thus, shuffling enables the decoupling of lock-acquire/release operations from policy enforcement, and allows lock developers to easily optimize for particular design factors (§3) or architectures. In this paper, we use a policy designed to optimize for NUMA architectures. Moreover, shuffling can group multiple policies together to devise complex lock algorithms. For example, in the blocking ShflLock we combine the NUMA-aware policy with an efficient parking/wakeup strategy: the shuffler groups waiters based on their NUMA socket and wakes up a nearby sleeping waiter. This approach solves the lock-waiter preemption problem by removing the wake-up overhead from the lock-holder critical path, a well-known issue for queue-based locks [7, 27, 45].

4.2 ShflLocks Design

We now present a family of ShflLock protocols, both non-blocking (§4.2.1) and blocking (§4.2.2). We further augment our blocking lock with a read indicator to design a blocking readers-writer lock (§4.2.3). We first enumerate a set of design decisions and later focus on the design of these locks.

Lock state decoupling. Unlike the MCS protocol, we decouple the lock acquisition state from the waiter queue. We achieve decoupling by introducing two levels of locks: a TAS lock for handling non-contended cases and a queue-based lock to handle moderate contention at the socket level. This approach is similar to the Linux spinlock and has several foundational benefits for building practical and scalable locks: a) ShflLocks remove the complexity of node allocation and tracking for the waiters queue because a queue node is only maintained within the acquire phase. This contrasts with conventional MCS/CNA locks, which maintain the node until the release phase. This prevents the lock-holder from reusing the node for a nested acquisition; b) ShflLocks use waiters for shuffling, moving work from the critical path to threads that are waiting; c) ShflLocks provide a fast trylock method with a single atomic compare-and-swap instruction; and d) ShflLocks mitigate the lock-waiter preemption problem through two mechanisms. First, the shuffler wakes up the next thread to acquire the lock proactively (§4.1). Second, ShflLocks allow lock stealing using the internal TAS lock.

Scheduling-aware parking strategy. We use the conventional *spin-then-park* strategy for blocking locks implemented for kernel space, but with the help of the task scheduler. For instance, waiters spin only for a duration allowed by the kernel thread scheduler. The scheduler notifies[2] a task if

[1]Per-socket: 8 sockets × 128 bytes; per-CPU: 192 cores × 128 bytes.

[2]The Linux scheduler provides a `need_resched()` method inside the kernel for yielding to the scheduler.

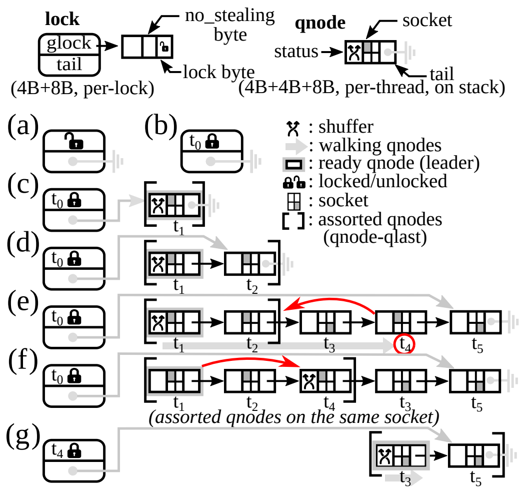

Figure 3. ShflLock^{NB} example. The lock structure consists of a state (`glock`) and the queue tail. The first byte of `glock` is the lock/unlock state, while the second byte denotes whether stealing is allowed. We encode multiple information in the qnode structure. (a) Initially, there is no lock holder. (b) $\mathtt{t}_0$ successfully acquires the lock via `CAS` and enters the critical section. (c) $\mathtt{t}_1$, of socket 1, executes `SWAP` on the lock's tail after the `CAS` failure on TAS. (d) Similarly, $\mathtt{t}_2$ from socket 1, also joins the queue. (e) Now, there are five waiters ($\mathtt{t}_1$–$\mathtt{t}_5$) waiting for the lock. $\mathtt{t}_1$ is the very first waiter, so it becomes the shuffler and traverses the queue to find waiters from the same socket. $\mathtt{t}_1$ then moves $\mathtt{t}_4$ (same socket) after $\mathtt{t}_2$. (f) After the traversal, $\mathtt{t}_1$ selects $\mathtt{t}_4$ as the next shuffler. (g) $\mathtt{t}_4$ acquires the lock after $\mathtt{t}_1$ and $\mathtt{t}_2$ have executed their critical sections. At this point, $\mathtt{t}_3$ becomes the shuffler.

it has utilized its current quota. Similar to CST locks [27], a waiter only parks if the system is overloaded. To know that, a waiter peeks at the number of active tasks on the current CPU scheduling queue, which is regularly updated by the scheduler. Otherwise, it yields to the scheduler, knowing that the scheduler will reschedule the task after some bookkeeping.

4.2.1 Non-Blocking Version: ShflLock^{NB}

ShflLock^{NB} uses a TAS and MCS combination, and maintains queue nodes on the stack [12, 24, 27]. However, we do extra bookkeeping for the shuffling process by extending the thread's qnode structure with socket ID, shuffler status, and batch count (to limit batching too many waiters from the same socket, which might cause starvation or break long-term fairness). Figure 3 shows the lock structure and the qnode structure. Our current design of the shuffling phase enforces the following four invariants for implementing any policy: 1) The successor of the lock holder, if it exists, always keeps its position intact in the queue. 2) Only one waiter can be an active shuffler, as shuffling is single threaded. 3) Only the head of the queue can start the shuffling process. 4) A shuffler may pass the shuffling role to one of its successors.

Figure 3 presents a running example of our lock algorithm. (a) A thread first tries to acquire the TAS lock; (b) it enters the critical section on success; otherwise, it joins the waiting queue ((c)–(e)). Now, the very next lock waiter, *i.e.*, $\mathtt{t}_1$, becomes the shuffler and groups waiters belonging to the same socket, *e.g.*, $\mathtt{t}_4$ (Figure 3 (e)). Once a shuffler iterates the whole waiting queue, it selects the last moved waiter as the next shuffler to start the process: $\mathtt{t}_1$ selects $\mathtt{t}_4$ (f). The shuffler keeps retrying to find a waiter from the same socket and leaves the shuffling phase after finding a successor from the local socket (f) or becoming the lock holder (g). The passing of a shuffler status, within a socket, lasts until the batching quota is exceeded.

Figure 4 presents the pseudo-code of our non-blocking version. The lock structure is 12 bytes (Figure 3): 4 bytes for the lock state (`glock`), and 8 bytes for the MCS tail. The algorithm works as follows: A thread `t` first tries to steal the TAS lock (line 6). On failure, `t` initiates the MCS protocol by first initializing a queue-node (`qnode`) on the stack, and then adding itself to the waiting queue by atomically swapping the tail with the `qnode`'s address (line 11–13). After joining the queue, `t` waits until it is at the head of the queue. To do that, `t` checks for its predecessor. If `t` is the first one in the queue, it disables lock stealing by setting the second byte to 1 to avoid TAS lock contention and waiter starvation (line 17). On the other hand, if waiters are present, `t` starts to spin locally until it becomes the leader in the waiting queue, *i.e.*, until its `qnode`'s status changes from `S_WAITING` to `S_READY` (line 47). Here, `t` also checks for the `is_shuffler` status. If the value is set, then `t` becomes the shuffler and enters the shuffling phase (line 51), which we explain later.

On reaching the head of the queue, `t` checks whether it can be a shuffler to group its successors based on the socket ID, meanwhile trying to acquire the TAS lock via the `CAS` operation (lines 20–30). Note that only the head of the queue can start the shuffling process if the `qnode`'s batch is set to 0. Otherwise, `t` can only shuffle waiters if the `is_shuffler` status is set to 1, which might be set by a previous shuffler.

The moment `t` becomes the lock holder, *i.e.*, `t` acquires the TAS lock, it follows the MCS unlock protocol (lines 33–40). `t` checks for the next successor (`qnode.next`). If the successor is present, `t` updates the successor's `qnode` status to `S_READY`. Otherwise, it tries to reset the queue's tail and enables lock stealing, which enables a new thread to get the lock via TAS if the queue is empty. The unlock phase is a conventional TAS unlock in which the first byte is reset to 0 (line 54).

Shuffling. Our shuffling algorithm moves a waiter's `qnode` from an arbitrary position to the end of the shuffled nodes in the waiting queue. Based on the specified policy, *i.e.*, socket-ID-based grouping, the shuffler (`S`) either updates the batch count or further manipulates the `next` pointer of waiting qnodes (line 84–100). We consider `S` as the first shuffled node. The algorithm is as follows: `S` first resets its `is_shuffler` to 0 and checks its quota of the maximum allowed shufflings to avoid starvation for remote socket waiters (line 71–73).

```
1  S_WAITING = 0 # Waiting on the node status
2  S_READY   = 1 # The waiter is at the head of the queue
3
4  def spin_lock(lock):
5    # Try to steal/acquire the lock if there is no lock holder
6    if lock.glock == UNLOCK && CAS(&lock.glock, UNLOCK, LOCKED):
7      return
8
9    # Did not get the node, time to join the queue; initialize node states
10   qnode = init_qnode(status=S_WAITING, batch=0,
11                      is_shuffler=False, next=None, skt=numa_id())
12
13   qprev = SWAP(&lock.tail, &qnode) # Atomically adding to the queue tail
14   if qprev is not None:  # There are waiters ahead
15     spin_until_very_next_waiter(lock, qprev, &qnode)
16   else: # Disable stealing to maintain the FIFO property
17     SWAP(&lock.no_stealing, True) # no_stealing is the second byte of glock
18
19   # qnode is at the head of the queue; time to get the TAS lock
20   while True:
21     # Only the very first qnode of the queue becomes the shuffler (line 16)
22     # or the one whose socket ID is different from the predecessor
23     if qnode.batch == 0 or qnode.is_shuffler:
24       shuffle_waiters(lock, &qnode, True)
25     # Wait until the lock holder exits the critical section
26     while lock.glock_first_byte == LOCKED:
27       continue
28     # Try to atomically get the lock
29     if CAS(&lock.glock_first_byte, UNLOCK, LOCKED):
30       break
31
32   # MCS unlock phase is moved here
33   qnext = qnode.next
34   if qnext is None: # qnode is the last one / next pointer is being updated
35     if CAS(&lock.tail, &qnode, None): # Last one in the queue, reset the tail
36       CAS(&lock.no_stealing, True, False) # Try resetting, else someone joined
37       return
38     while qnode.next is None: # Failed on the CAS, wait for the next waiter
39       continue
40     qnext = qnode.next
41   # Notify the very next waiter
42   qnext.status = S_READY
43
44 def spin_until_very_next_waiter(lock, qprev, qcurr):
45   qprev.next = qcurr
46   while True:
47     if qcurr.status == S_READY:  # Be ready to hold the lock
48       return
49     # One of the previous shufflers assigned qcurr as a shuffler
50     if qcurr.is_shuffler:
51         shuffle_waiters(lock, qcurr, False)
52
53 def spin_unlock(lock):
54   lock.glock_first_byte = UNLOCK # no_stealing is not overwritten
55 MAX_SHUFFLES = 1024
56
57 # A shuffler traverses the queue of waiters (single threaded)
58 # and shuffles the queue by bringing the same socket qnodes together
59 def shuffle_waiters(lock, qnode, vnext_waiter):
60   qlast = qnode # Keeps track of shuffled nodes
61   # Used for queue traversal
62   qprev = qnode
63   qcurr = qnext = None
64
65   # batch → batching within a socket
66   batch = qnode.batch
67   if batch == 0:
68     qnode.batch = ++batch
69
70   # Shuffler is decided at the end, so clear the value
71   qnode.is_shuffler = False
72   # No more batching to avoid starvation
73   if batch >= MAX_SHUFFLES:
74     return
75
76   while True: # Walking the linked list in sequence
77     qcurr = qprev.next
78     if qcurr is None:
79       break
80     if qcurr == lock.tail:  # Do not shuffle if at the end
81       break
82
83     # NUMA-awareness policy: Group by socket ID
84     if qcurr.skt == qnode.skt: # Found one waiting on the same socket
85       if qprev.skt == qnode.skt: # No shuffling required
86         qcurr.batch = ++batch
87         qlast = qprev = qcurr
88
89       else: # Other socket waiters exist between qcurr and qlast
90         qnext = qcurr.next
91         if qnext is None:
92           break
93         # Move qcurr after qlast and point qprev.next to qnext
94         qcurr.batch = ++batch
95         qprev.next = qnext
96         qcurr.next = qlast.next
97         qlast.next = qcurr
98         qlast = qcurr  # Update qlast to point to qcurr now
99     else: # Move on to the next qnode
100      qprev = qcurr
101
102    # Exit → 1) If the very next waiter can acquire the lock
103    #        2) A waiter is at the head of the waiting queue
104    if (vnext_waiter is True and lock.glock_first_byte == UNLOCK) or
105          (vnext_waiter is False and qnode.status == S_READY):
106      break
107
108  qlast.is_shuffler = True
```

Figure 4. Pseudo-code of the non-blocking version of ShflLocks and the shuffling mechanism.

Similar to CNA, we can also use a random generator to mitigate starvation. Now, S iterates over qnodes in the queue while keeping track of the last shuffled qnode (`qlast`). While traversing, S always marks the nodes that belong to its socket by increasing the batch count. It only does pointer manipulations when there are waiters between the last shuffled node and the node belonging to S's socket (lines 89–98). Finally, S always exits the shuffling phase if either the TAS lock is unlocked or S becomes the head of the queue (line 104–105). Before exiting the shuffling phase, S assigns the next shuffler: the last marked node (line 108). S can stop traversing the queue for two more reasons: 1) if successors are absent (line 78, 91), as S wants to avoid the locking delay because it might soon acquire the lock; 2) if S reaches the queue tail, as there might be waiters joining at the end of the tail, which it cannot move (line 80).

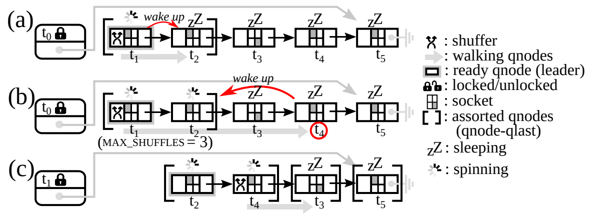

Figure 5. A running example of how a shuffler shuffles waiters with the same socket ID and wakes them up. (a) t_0 is the lock holder; t_1 is the shuffler and is traversing the queue. t_2 is sleeping, but t_1 wakes it up. (b) t_2 becomes active, while t_1 continues shuffling and reaches t_4, t_1 first moves t_4 after t_2, and wakes up t_4 to mitigate the wakeup latency. (c) When t_0 releases the lock, t_1 acquires it; t_2 and t_4 are actively spinning for their turn; t_4 is the shuffler.

Optimization. Our shuffling algorithm has unnecessary pointer chasing when a newly selected shuffler, assigned by the previous S, has to traverse the queue. We avoid this issue by further encoding extra information about the qnode where S stopped traversal in the next shuffler's qnode structure. This leads to traversing mostly from the near end of the tail, thereby better utilizing the time of waiters.

```
1  + S_PARKED   = 2 # Parked state (used by lock waiter for sleeping)
2  + S_SPINNING = 3 # Spinning state (used by shuffler for waking up)
3
4    def mutex_lock(lock):
5    ...
6      # Notify the very next waiter
7  -   qnext.status = S_READY
8  +   # Atomically SWAP the qnode status
9  +   prev_status = SWAP(&qnext.status, S_READY)
10 +   if node_pstate == S_PARKED: # Required for avoiding lost wakeup
11 +     wake_up_task(qnext.task) # Explicitly wake up the very next waiter
12
13   def spin_until_very_next_waiter(lock, qprev, qcurr):
14   ...
15       if curr.status == S_READY:
16         return
17 +     if task_timed_out(qcurr.task): # Running quota is up! Give up
18 +       park_waiter(qcurr) # Will try to park myself
19
20   def shuffle_waiters(lock, qnode, next_flag):
21   ...
22     if batch >= MAX_SHUFFLINGS:
23       return
24 +   SWAP(&qnode.status, S_SPINNING) # Don't sleep, will soon acquire the lock
25
26     while True:
27   ...
28       # NUMA-awareness and wakeup policy
29       if qcurr.skt == qnode.skt:
30         if qprev.skt == qnode.skt: # No shuffling required
31 +         update_node_state(qcurr) # Disable sleeping
32           qnode.batch = ++batch
33           qlast = qcurr
34           qprev = qcurr
35
36         else:
37 +         update_node_state(qcurr) # Disable sleeping
38           qnode.batch = ++batch
39           qprev.next = qnext
40
41 + def update_node_state(qnode):
42 +   # If the task is waiting, then make it spinning
43 +   if CAS(&qnode.status, S_WAITING, S_SPINNING):
44 +     return
45 +   # If the task is sleeping, then wake it up for spinning
46 +   if CAS(&qnode.status, S_PARKED, S_SPINNING):
47 +     wake_up_task(qnode.task) # Wakeup task (off the critical path)
48 +
49 + def park_waiter(qnode):
50 +   # Park it when the task is waiting
51 +   if CAS(&qnode.status, S_WAITING, S_PARKED):
52 +     park_task(qnode.task)
```

Figure 6. The extra modification required to convert our non-blocking version of SHFLLOCK to a blocking one.

```
  def mutex_lock(lock):
  ...
+   qnext = qnode.next  # Try to get the successor before acquiring TAS
+   if qnext is not None:
+     if SWAP(&qnext.status, S_SPINING) == S_PARKED:
+       wake_up_task(qnext.task)

    # qnode is at the head of the queue; time to get the TAS lock
    while True:
  ...
```

Figure 7. An optimization for avoiding a waiter wakeup issue in the critical path with an extra state update before the TAS lock.

4.2.2 Blocking Version: SHFLLOCK^B

We augment SHFLLOCK^{NB} to incorporate an effective parking/wakeup policy. Our lock algorithm departs from the scalable queue-based blocking designs as we do not have a separate parking list [14, 27, 40]. This allows us to save up to 16–20 bytes per lock compared to existing separate parking list-based locks. We maintain both the active and passive waiters in the same queue, and utilize the TAS lock for lock stealing and shuffling to efficiently wake up parked waiters off the critical path. SHFLLOCK^B avoids the lock-waiter preemption by allowing the TAS lock to be unfair in the fast path [12, 27] as well as keeping the head of the waiting queue active, *i.e.*, not scheduled out. In addition, we modify the MCS protocol to support waiter parking and wakeup. We further extend our shuffling protocol to wake up the nearby sleeping waiters while shuffling the queue for NUMA-awareness in both under- and over-subscribed cases (Figure 5). To support efficient parking/wakeup, we extend our non-blocking version with two more states: 1) *parked* (`S_PARKED`), in which a waiter is scheduled out for handling core over-subscription and 2) *spinning* (`S_SPINNING`), in which a shuffled waiter is always spinning for mitigating the convoy effect.

Figure 6 shows the modifications on top of SHFLLOCK^{NB}. While spinning locally on its status, a waiter `t` checks if the time quota is up (line 17). In that case, `t` tries to atomically change its qnode status from `S_WAITING` to `S_PARKED` (line 51). On success, `t` parks itself out (line 52); otherwise, `t` goes back to spinning. In the shuffling phase, a shuffler `S` also wakes up the shuffled sleeping waiters (lines 31, 37). Note that this is a best effort strategy, in which an `S` first tries to atomically `CAS` the `qnode`'s status from `S_WAITING` to `S_SPINNING`, hoping that the waiter is still waiting locally; if the operation fails, then `S` does another explicit `CAS` from `S_PARKED` to `S_SPINNING` and wakes up the sleeping waiter if successful (line 47). The last notable change to the algorithm is notifying the head of the queue. There is a possibility that the very next waiter might be sleeping. We atomically swap the `qnext`'s state to `S_READY` (line 9) and wake up the waiter at the head of the queue if the return value of the atomic `SWAP` operation is `S_PARKED` (line 11).

Optimizations. Our first optimization is to enable lock stealing by not setting the second byte when the queue begins. The reason is that waking up a waiter ranges from $1\mu s$–10ms, which adds overhead in the acquire phase. The second optimization regards the waiter wakeup. Our current design leads to waking up the queue head inside the critical section, even though it is rare (see §6). As shown in Figure 7, we explicitly set the successor status to `S_SPINNING` and wake it up if parked. This approach further removes the rare occurrence of the waiter preemption problem at the cost of an extra atomic operation, which is acceptable, as the atomic operation is only between two qnodes. It is not a part of the critical section, as other joining threads can steal the lock (TAS) to ensure the forward progress of the system.

4.2.3 Readers-Writer Blocking SHFLLOCK

Linux uses a readers-writer spinlock [31], which combines a readers indicator with a queue-based lock. This lock queues waiting readers and writers to avoid cache-line contention and bouncing. We use a similar design on top of our blocking SHFLLOCK. Thus, our readers-writer lock inherently becomes a blocking lock, and at most only one reader or a writer can spin to acquire, while others spin locally. Our lock design provides only long-term fairness due to the NUMA-awareness of the SHFLLOCK. This is acceptable because even the Linux's `rwsem` is writer-preferred to enhance throughput

Kernel space		
Locks	**Replaced**	**Selection criteria**
ShflLocks	All	–
CNA [16]	`qspinlock`	Compact NUMA-aware lock (NB)
CST [27]	`mmap_sem` / `i_rwsem` / †	Hierarchical + dynamic allocation (B)
Cohort [18]	`s_vfs_rename_mutex` †	Hierarchical + static allocation (NB)

Userspace	
Locks	**Selection criteria**
MCS [37]	Queue-based lock (NB)
HMCS [8]	Representative cohort lock (NB)
CNA [16]	Compact version of NUMA-aware MCS (NB)
MCSTP [23]	Preemption adaptive MCS for over-subscription (NB)
Pthread	Stock version used by everyone (B)
Mutexee [19]	Optimized version of Pthread (B)
Malthusian [14]	Culls extra thread deterministically (B)

B: Blocking; NB: Non-blocking † Both CST and Cohort replace all three locks.

Table 2. Locks evaluated in both the kernel space and the userspace. In the kernel space, we replace all locks with ShflLocks. We use `LD_PRELOAD` to replace all the mutex-based locks in the userspace.

over fairness [30, 46], similar to prior work [6]. Note that the shuffler only moves writers in the wait queue because all the contiguous readers can enter the critical section together, irrespective of NUMA socket.

Details. We augment a ShflLock, henceforth called `wlock`, with a read/write counter, which encodes: a readers count (`Rcount`), a writer waiting bit (`WWb`) indicating if a writer is waiting to acquire the lock, and a writer byte (`WB`), indicating if a writer is currently holding the lock. A writer enters the critical section on successfully setting `WB` from 0 to 1; otherwise, it enqueues itself to acquire the underlying blocking lock (`wlock`). After acquiring the `wlock`, the writer sets the waiting bit (`WWb`) to 1 to prohibit new readers from entering the critical section and waits for existing readers to leave. Once readers leave, the writer atomically resets `WWb` to 0 and sets `WB` to 1, releases `wlock`, and then enters the critical section. In the writer unlock phase, a writer resets `WB` to 0. A reader first atomically increments `Rcount` and enters the critical section if both `WB` and `WWb` are 0. If non-zero, the reader first decreases `Rcount` and starts acquiring `wlock`. Once it holds `wlock`, it first increments the `Rcount` to prevent writers from entering the critical section and waits for the existing writer to exit. When `WB` is 0, the reader enters the critical section after releasing `wlock`. In the unlock phase, a reader atomically decreases `Rcount`.

5 Implementation

We implement all ShflLocks in the Linux kernel v4.19-rc4 and entirely replace `mutex` and `rwsem` with ours. Our replacement results in adding 459 and 557 lines of code (LoC) for `mutex` and `rwsem`, respectively. We add our shuffling phase to the `qspinlock` in 150 LoC, without increasing the lock size. We have also tested ShflLocks with locktorture. Our code is publicly availabe at https://github.com/sslab-gatech/shfllock.

Lock type	Workload	Lock: Usage
Non-blocking	MWRL [39] lock1 [2]	**rename seqlock**: Rename files within a directory **files_struct.file_lock**: fd allocation / fcntl
Blocking	MWRM [39]	**sb->s_vfs_rename_mutex**: Rename a file across directory
RW blocking	MWCM [39] MRDM [39]	**inode->i_rwsem**: Create files in the directory (writer) **inode->i_rwsem**: Enumerate files in the directory (readers)

Table 3. Lock usage in various micro-benchmarks [2, 39].

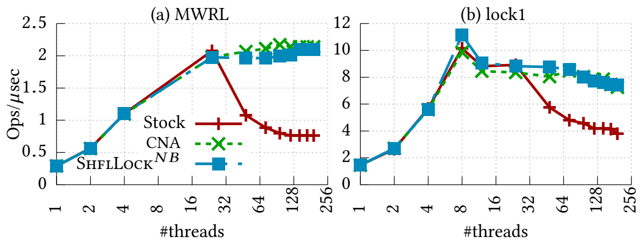

Figure 8. Impact of non-blocking locks on the scalability of micro-benchmarks [2, 39]. Refer to Table 3 for lock usage. Here, Stock refers to the default spinlock.

6 Evaluation

We evaluate ShflLocks by answering three questions:

Q1. How do ShflLocks, implemented in the kernel, impact micro-benchmarks (§6.1) and real applications (§6.2)?

Q2. How does each design decision affect ShflLocks performance and how fair are ShflLocks (§6.3)?

Q3. How do userspace ShflLocks impact applications' performance and memory footprint? (§6.4)

Evaluation setup. We use micro-benchmarks that stress a single lock [2, 39], and three workloads that heavily stress several kernel subsystems [4, 50]. We also use a hash-table nano-benchmark [48] to break down the performance characteristics of ShflLocks. Table 2 lists all the evaluated locks and the selection criteria. We evaluate on an 8-socket, 192-core machine with Intel Xeon E7-8890 v4 (hyperthreading disabled). We use `tmpfs` to minimize file system overhead.

6.1 ShflLock Performance Comparison

We evaluate the performance of all ShflLocks using a set of micro-benchmarks [2, 39]. Each micro-benchmark instantiates a set of threads and pins them to cores. These threads contend on a single lock while performing specific tasks (Table 3) for 30 seconds. We pin two threads on each core in the over-subscribed scenario for blocking locks.

Non-blocking ShflLock. Figure 8 shows that both CNA and ShflLock outperform the Linux version (Stock) by 2.8× and 2× on MWRL and lock1, respectively, while maintaining the same throughput under lower contention, *e.g.*, within a single socket. Similar to ShflLock, CNA maintains NUMA-awareness by using the lock holder to physically split the waiting queue into two, one for local threads and the other for remote threads. Meanwhile, ShflLock uses lock waiters to shuffle waiters around, mostly off the critical path.

Blocking ShflLock. We compare ShflLock with Linux `mutex` and `rwsem` (Stock), Cohort non-blocking lock, and CST lock (Table 2). We test these locks in both under- and

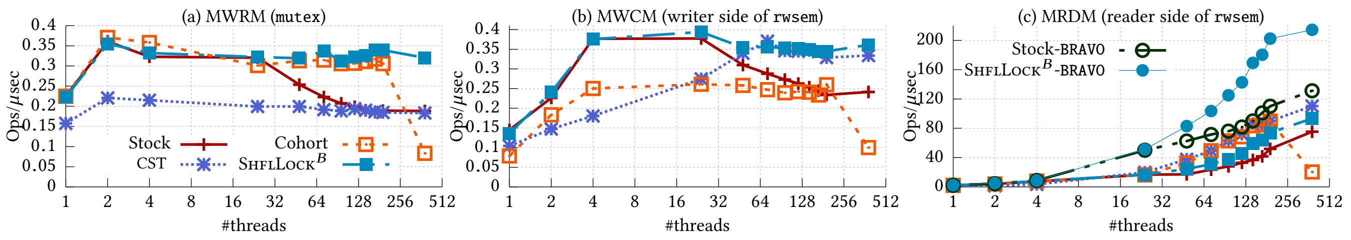

Figure 9. Impact of blocking locks on the scalability of micro-benchmarks with up to 2× over-subscription (384 threads: we pin two threads on each core). Cohort and CST are the non-blocking and blocking hierarchical locks, respectively. Refer to Table 3 for lock usage.

over-subscribed cases, *i.e.*, up to 384 threads by pinning two threads on a core in a round-robin manner. Figure 9 (a) shows the results for the MWRM benchmark, which renames files across directories. MWRM first pre-allocates a set of empty files in per-thread directories; then, each thread moves a file from its directory to a shared directory, which stresses the super-block's mutex (Table 3). ShflLock maintains the best throughput in both under- and over-subscribed scenarios and is 1.8× faster than both CST and Stock. The stock suffers from cache-line bouncing at high core count but maintains the throughput in the over-subscribed case. Cohort is a non-blocking lock, which performs well up to the total number of cores (192 threads), but significantly degrades MWRM's throughput in the over-subscribed case (384 threads), as waiters waste CPU cycles. CST does not scale because it dynamically allocates its socket structure before each critical section, which results in excessive allocations with elongated critical section length. In contrast, Cohort pre-allocates its socket structure, and does not extend the critical section.

Readers-Writer Blocking ShflLock. Figure 9 (b) shows the impact of ShflLock when stressing the writer lock of rwsem. We use the MWCM benchmark, in which each worker creates 4KB files in a shared directory to stress inode allocation. We observe that ShflLock maintains the best throughput at all core counts, due to its ability to better adapt to the workload. For example, ShflLock is 1.8–2× faster than hierarchical locks within a socket and 1.5× faster than Stock at 192 threads. Cohort can only scale up to four cores (almost 55% slower than ShflLock) because memory allocation becomes an issue as the inode size increases by 3.4×. Meanwhile, CST avoids this scenario, as it only allocates the memory for one socket initially, but its performance only scales to reach that of ShflLock after 2 NUMA nodes.

Figure 9 (c) shows the impact of ShflLock when stressing the readers side of the rwsem. We use MRDM, in which each thread enumerates files in a directory. We also include a recently proposed approach, called BRAVO [15], that tries to mitigate the centralized reader overhead by using a global readers table. We observe that both hierarchical locks are faster than ShflLock and rwsem because of their per-socket readers indicator, which localizes the contention within a socket. ShflLock is still faster than stock rwsem by 1.2–1.5× because the stock version suffers from the spurious sleeping of waiters, which results in extra cache-line contention on the reader indicator, thereby impacting the throughput. We also observe that the BRAVO approach improves the throughput for both Stock and ShflLock up to 2.3× compared to Cohort and CST locks at 192 threads. However, due to the extra cache-line contention in the stock version, ShflLock-BRAVO still outperforms Stock-BRAVO by 1.6× at 384 threads.

6.2 Improving Application Performance

We evaluate three applications that extensively stress various subsystems of the Linux kernel. Figure 10 reports the throughput of applications and the memory used by locks, which are mostly blocking and are present in several data structures such as inodes, task structures, and memory mappings. Table 2 shows the locks modified for the evaluation. Note that CNA only modifies the spinlock, but does not affect the size of blocking locks.

AFL [50], a fuzzer, is an embarrassingly parallel workload that heavily uses fork() to execute test cases and scan directories created by the fuzzing instances. AFL suffers from the following overheads: process forking, repeatedly creating and unlinking files in a private directory, and scanning other instances' directories [49]. In addition, AFL suffers from the gettimeofday() syscall, as each instance issues this syscall to log information. Figure 10 (a) shows AFL throughput and memory usage with various locks. We observe that all the existing versions of NUMA-aware locks improve throughput compared with the stock version. For instance, CNA decreases the qspinlock overhead due to process forking and gettimeofday() from 48% to 32%. Meanwhile, both CST and Cohort locks improve the file system performance, as these locks scale as well as ShflLocks. However, their large memory footprint starts stressing the memory allocator at higher core count, as the bottleneck completely shifts to process forking (30%). Finally, ShflLocks improve performance on two fronts: they improve throughput by 1.2–1.6× while reducing the lock overhead by 35.4–95.8% at 192 threads. The significant overhead now comes from the gettimeofday() syscall, as perf shows almost 20% of CPU cycles.

Exim [4] is a process-intensive workload that forks a new process for every connection. Each connection then forks twice to handle messages and file system operations [39], which heavily over-subscribe the system. Exim creates about 3× copies for each message and heavily stresses the kernel in three subsystems: memory management, file systems, and

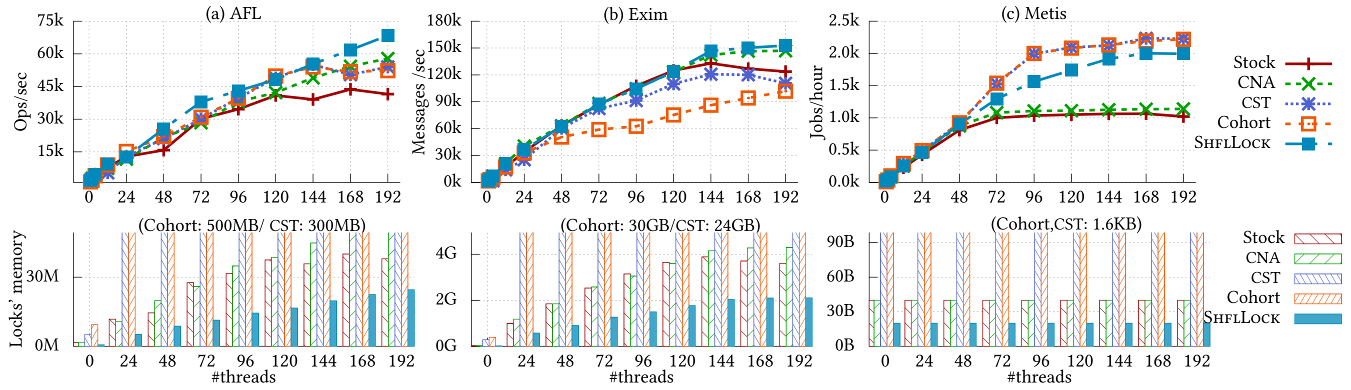

Figure 10. Impact of locks on application scalability and on memory footprint, while running three applications with ShflLocks, Linux stock version (Stock), CNA, CST, and Cohort. Refer to Table 2 for specific changes. ShflLock reduces the memory footprint because of the blocking locks that are embedded in inodes, task structure, and memory management structures.

network connections. On average, about 50% of the time is spent in the process forking/exiting and interrupts. Figure 10 (b) shows Exim throughput and memory usage with various locks. Both ShflLocks and CNA improve throughput as they decrease the CPU idle time by 50% compared with CST, Cohort, and Stock while improving the useful work by ≈2%. The improvement is a result of a decrease in lock contention by 10% (relative to Stock) in the cleaning up of reverse mappings [36]. The throughput of the CST and Cohort locks decreases because these locks stress the memory allocator (see §3), as the benchmark generates about 8M files in 20 secs. In summary, ShflLocks improve the throughput by 1.5× compared with hierarchical locks as well as decrease the memory footprint by 40.8–92.9% among all existing versions.

Metis is an in-memory map-reduce framework, representing a page-fault-intensive workload that stresses a single lock in the kernel: the reader side of the `mmap_sem`. Figure 10 (c) shows that both Cohort and CST locks outperform all the centralized counter-based locks because of localizing the contention within a socket but at the cost of ≈80× extra memory. However, our readers-writer blocking lock is still faster than Stock because the stock version also encodes the sleeping waiters in its count indicator, as it has almost 3.4× higher atomic instructions compared with ShflLock when measured with perf [35]. This workload also shows the efficiency of our under-subscribed scenario. Compared to `rwsem`, that has 33% more idle time due to its naive parking strategy, ShflLock's readers do not park themselves. This results in less idle time (1.2%) and higher throughput (2.4×) than the original `rwsem`.

Summary. Figure 10 shows the impact of scheduling interaction, the overhead of memory allocation with respect to locks in both under- and over-subscribed cases, with varying contention levels. Our holistic design of ShflLocks accommodates NUMA-awareness at high core count and shows that memory overhead (whether dynamic or static) heavily influences the scalability of applications. Compared to all locks, ShflLocks reduce the memory footprint overhead up to 98.8% and 35.4% when compared with the hierarchical locks, and and Stock, respectively.

6.3 Performance Breakdown

We now do an in-depth analysis of ShflLocks using a hash-table benchmark in the kernel [48]. A global lock guards the hash table. For ShflLock^{NB} and ShflLock^{B}, we use 1% writes, and for ShflLock^{RW} (readers-writer blocking lock), we generate 1% and 50% writes on the hash table. Figure 11 shows the results as well as the factor analysis of ShflLocks.

Non-blocking ShflLock^{NB}. Figure 11 shows (a) throughput and (b) fairness. We calculate the fairness factor described by Dice *et al.* [16], in which we sort the number of operations performed by each thread, and divide the sum of the second half of threads' operations (sorted in increasing order) by the total number of operations. Thus, the resulting fairness factor is a number between 0.5 and 1, with a strictly fair lock yielding a factor of 0.5 and an unfair lock yielding a factor close to 1. We observe that both CNA and ShflLock are the best performing, while the performance of Cohort locks is affected because of bloating of the critical section in the case of one socket. Although NUMA-aware locks impact the fairness of locks, they still maintain long-term fairness, as the fairness factor is close to 0.5.

Figure 11 (e) shows the improvement at 192 threads due to the various optimizations in ShflLocks. Here, `Base` represents no shuffling, which behaves as the NUMA-oblivious spinlock. `+Shuffler` represents a version of ShflLocks where only the very first waiter shuffles, but doesn't pass the role to other threads. This version improves the throughput by 16% over `Base`. `+Shufflers` represents the algorithm we describe in Figure 4, in which a shuffler passes the role to any waiter in the local socket. This approach results in almost a 10% improvement over `+Shuffler`. Finally, the `+qlast` optimization avoids the unnecessary pointer chasing done by the shuffler to determine where to insert a rellocated qnode by saving the last qnode of the threads with the same socket ID. This optimization improves throughput by 30% .

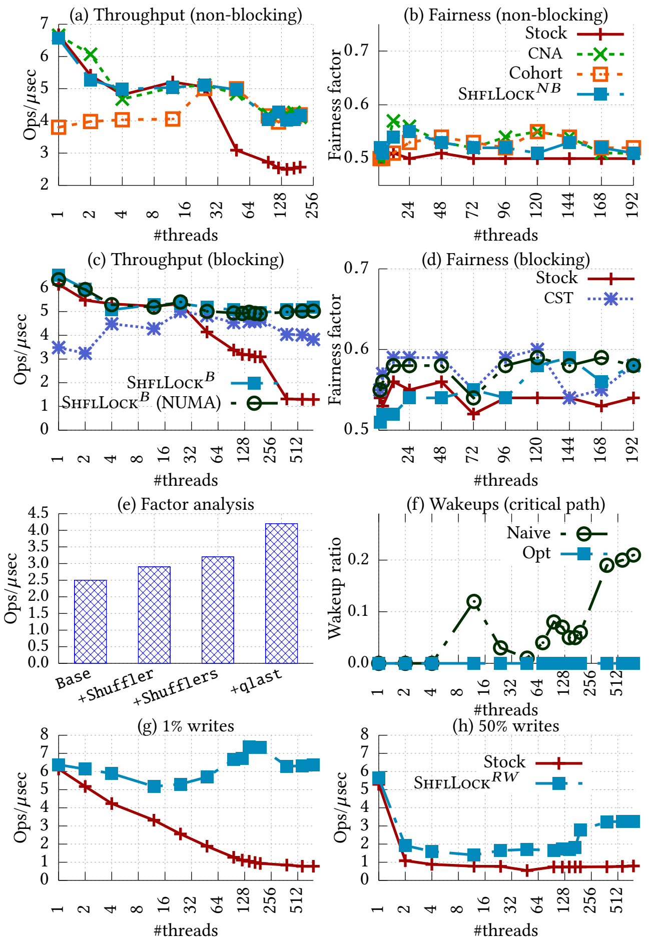

Figure 11. Impact on throughput and long-term fairness of non-blocking and blocking locks on the hash-table benchmark. For blocking locks, we over-subscribe the system by 4×. We also include the factor analysis of several phases introduced by SHFLLOCK^{NB}, and the number of wakeups in the critical path for SHFLLOCK^{B}. Later, we show the impact on throughput with centralized readers-writer locks: Stock and SHFLLOCK^{RW} for 1% and 50% writes up to 4× over-subscription.

Blocking SHFLLOCK^{B}. Figure 11 shows (c) throughput, and (d) the fairness factor for SHFLLOCK^{B}. We see that SHFLLOCK maintains the best throughput even up to 4× over-subscription because it aggressively steals the lock in the over-subscribed case. Our lock stealing is inherently NUMA-aware because most of the remote waiters join the queue; meanwhile, the local active waiters only steal the lock if the very-next waiter (shuffler) is busy waking up its successor. We further confirm this result by only allowing the stealing from the local NUMA-socket, which shows the same throughput, as shown by SHFLLOCK (NUMA) in the figure. Meanwhile, even in the under-subscribed scenario, we observe that the fairness factor reaches up to 0.6 because of lock stealing but does not starve waiters (d). Besides, the shuffler proactively wakes up threads that will acquire the lock soon, which completely removes the waking-up overhead from the critical path, even in the over-subscribed scenario (refer to Figure 11 (f)).

Blocking SHFLLOCK^{RW}. Figure 11 ((g) and (h)) show that the SHFLLOCK^{RW} has higher throughput than the stock version by 8.1× and 3.7× for 1% and 50% writes, respectively. This happens because the stock version is very inefficient, as most of the threads are blocked, resulting in idling of the CPU (99%). Meanwhile, SHFLLOCK^{RW} maintains consistent performance regardless of the contention on the lock, even further batching readers together at a higher count to maintain good throughput. One point to note is that in the case of over-subscription, SHFLLOCK^{RW} aggressively batches readers and writers, which slightly improves the throughput.

6.4 Performance With Userspace SHFLLOCK

We now evaluate SHFLLOCKS on three benchmarks: LevelDB for high contention, Streamcluster for the trylock interface, and Dedup for memory allocation [21]. We integrate both SHFLLOCK and CNA into LiTL [22] for evaluation.[3] We use a set of blocking and non-blocking locks that have the best performance for the selected workloads (refer to Table 2).

LevelDB is an open-source key-value store [20]. We use the `readrandom` benchmark that contends on the global database lock. Figure 12 (a) and (b) show the throughput with non-blocking and blocking locks, respectively, with up to 4× over-subscription for the blocking ones after running for 60 seconds. We keep Pthread as a reference for the comparison. We find that SHFLLOCK is almost as fast as the existing NUMA-aware locks with increasing core count and is 2.4× faster than MCS locks with 192 threads. We also observe that Pthread only scales up to eight threads because it starts parking threads. The throughput of blocking locks is better than non-blocking ones because fewer threads are contending on locks. SHFLLOCK^{B} outperforms others by 1.7–3.8× at 192 threads. Moreover, we see that SHFLLOCK maintains almost the same throughput even at 768 threads, and achieves 1.6–12.5× higher throughput. This happens for two reasons: efficient waking up of waiters and aggressive lock stealing, as there are still active waiters that acquire the lock.

Streamcluster is a data mining workload [1], which uses a custom barrier implementation to synchronize threads between the different phases of the application. The barrier implementation uses a mix of trylock and lock operations, as well as condition variables, which amount to almost 30% of the execution time [21]. Figure 12 (c) shows the execution time of streamcluster. Guerraoui *et al.* pointed out that the contention-hardened trylock interface results in better execution of this workload, which we observe for HMCS as well as for MCSTP (slightly better than MCS and CNA). However, we find that SHFLLOCKS has almost similar execution time as that of HMCS and is 1.3–4.4× faster than other locks. This happens because of our main design choice: decoupling the lock state from the waiting queue. Even though CNA is

[3]Similar to Pthread, we use `futex()` system call to implement SHFLLOCK^{B}. The waiter spins for a constant duration and then parks itself.

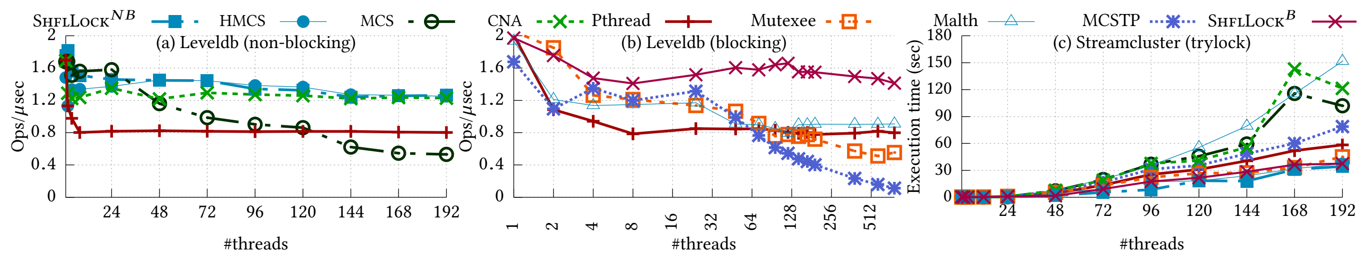

Figure 12. Total throughput of LevelDB benchmark and the streamcluster benchmark with various blocking and non-blocking locks. We further over-subscribe the cores for levelDB (b) to test the impact of blocking locks with 4× the number of cores.

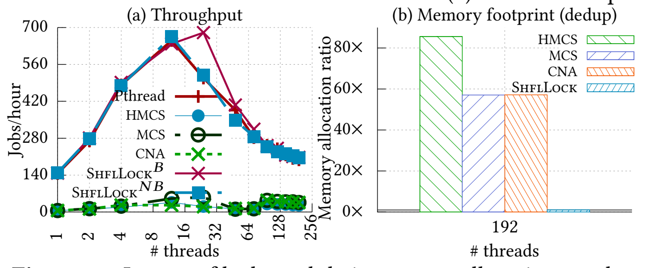

Figure 13. Impact of locks and their memory allocation overhead on the scalability of Dedup. We report the overall memory allocation overhead that is used during the entire run, with respect to Pthread.

NUMA-aware, its performance is similar to MCS because the lock state and the queue tail are coupled. On further analysis, we find that queue-based locks, such as HMCS, CNA, and MCS, spend 4× extra time (failed and succeeded trylock time) and 15× excessive trylock operations than ShflLock, which improves ShflLock's throughput over MCS and CNA. Despite HMCS spends extra time in the trylock operation, it spends 4× less time in the lock operation than ShflLock because waiters statically partition the list, which results in the most efficient NUMA-aware lock. In summary, tail and state decoupling provides a window of opportunity that allows the trylock operation to succeed.

Dedup represents an enterprise storage workload [1], which allocates up to 266K locks throughout its lifetime and heavily stresses the memory allocator as well as creates almost 3× the number of threads for several application phases. Figure 13 shows (a) the number of jobs per hour and (b) the ratio of the overall memory allocated during the application's lifetime with respect to Pthread. We observe that the benchmark is not scalable after 48 cores (2 sockets) because of huge over-subscription and memory allocation. Both versions of ShflLocks have the same scalability as that of a light lock, as Pthread, and the blocking version is even 5% faster at 48 cores by avoiding the lock-waiter preemption.

ShflLock adds no memory overhead over Pthread, but other queue-based lock add the overhead of per-thread queue nodes allocated on the heap. While these locks could theoretically allocate queue nodes on the stack, it would require application-wide changes to the Dedup code and Pthread API; ShflLock's queue node design is easier to deploy. In addition, the hierarchical locks also allocate per-socket structures. This leads to more than 90% of the time being spent in memory allocations. For instance, the ratio of extra memory allocation is 58–87× higher for existing queue-based locks.

7 Discussion

Policies. Our shuffling mechanism opens new opportunities to implement different policies based on the hardware behavior or the requirements of the application. For example, we can devise policies to support non-inclusive caches [41] or a multi-level NUMA hierarchy [43]. In this case, the shuffler optimizes the waiting list according to the NUMA node, but it also keeps track of the number of hops. In addition, shuffling can also be used to avoid the priority inversion issue [28] or to devise approaches for applications that require occupancy-aware scheduling, (*i.e.*, prioritize lock-acquire based on the time spent inside the critical section). In addition, shuffling can also be beneficial in designing an adaptive readers-writer lock, in which a waiter switches among centralized, per-socket or per-CPU reader indicators, depending on workload and thread contention.

8 Conclusion

Locks are still the preferred style of synchronization. However, a considerable discrepancy exists in practice and design. We classify such issues into four dominating factors that impact the performance and scalability of lock algorithms and find that none of the locks meets all the required criteria. To that end, we propose a new technique, called *shuffling*, that enables the decoupling of lock design from policy enforcement, such as NUMA-awareness or parking/wakeup strategies. Moreover, these policies are enforced entirely off the critical path by the waiters. We then propose a family of locking protocols, called ShflLocks, that respects all of the factors and shows that we can indeed achieve performance without additional memory overheads.

9 Acknowledgments

We thank Dave Dice, Alex Kogan, Jean-Pierre Lozi, the anonymous reviewers, and our shepherd, Eddie Kohler, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851, and CNS-1749711; ONR under grant N00014-18-1-2662, N00014-15-1-2162, and N00014-17-1-2895; DARPA TC (No. DARPA FA8650-15-C-7556); ETRI IITP/KEIT [B0101-17-0644]; and gifts from Facebook, Mozilla, Intel, VMware, and Google.

References

[1] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton, NJ, USA. Advisor(s) Li, Kai. AAI3445564.

[2] Anton Blanchard. 2013. will-it-scale. (2013). https://github.com/antonblanchard/will-it-scale.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Vancouver, Canada, 1–16.

[4] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Vancouver, Canada, 1–16.

[5] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium.* Ottawa, Canada.

[6] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, Shenzhen, China, 157–166.

[7] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. 2017. An Efficient Abortable-locking Protocol for Multi-level NUMA Systems. In *Proceedings of the 22nd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, Austin, TX, 14.

[8] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, San Francisco, CA, 12.

[9] Milind Chabbi and John Mellor-Crummey. 2016. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, Barcelona, Spain, 22:1–22:14.

[10] Dave Chinner. 2014. Re: [regression, 3.16-rc] rwsem: optimistic spinning causing performance degradation. (2014). https://lkml.org/lkml/2014/7/3/25.

[11] Jonathan Corbet. 2010. Big reader locks. (2010). https://lwn.net/Articles/378911/.

[12] Jonathon Corbet. 2014. MCS locks and qspinlocks. (2014). https://lwn.net/Articles/590243/.

[13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP).* ACM, Farmington, PA, 33–48.

[14] Dave Dice. 2015. Malthusian Locks. *CoRR* abs/1511.06035 (2015). http://arxiv.org/abs/1511.06035

[15] Dave Dice and Alex Kogan. 2019. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC).* USENIX Association, Renton, WA, 315–328.

[16] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19).* ACM, New York, NY, USA, Article 12, 15 pages.

[17] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11).* 65–74.

[18] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, New Orleans, LA, 247–256.

[19] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking Energy. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC).* USENIX Association, Denver, CO, 393–406.

[20] Sanjay Ghemawat and Jeff Dean. 2019. LevelDB. (2019). https://github.com/google/leveldb

[21] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. 2019. Lock—Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.* 36, 1, Article 1 (March 2019), 149 pages. https://doi.org/10.1145/3301501

[22] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC).* USENIX Association, Denver, CO, 649–662.

[23] Bijun He, William N. Scherer, and Michael L. Scott. 2005. Preemption Adaptivity in Time-published Queue-based Spin Locks. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05).* 7–18.

[24] IBM. 2016. IBM K42 Group. (2016). http://researcher.watson.ibm.com/researcher/view_group.php?id=2078.

[25] Intel 2016. *Xeon Processor E7-8890 v4 (60M Cache, 2.20 GHz).* Intel. http://ark.intel.com/products/93790/Intel-Xeon-Processor-E7-8890-v4-60M-Cache-2_20-GHz.

[26] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. 1991. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP).* ACM, New York, NY, USA, 41–55.

[27] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC).* USENIX Association, Santa Clara, CA.

[28] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17).* USENIX Association, Berkeley, CA, USA, 345–358.

[29] Ran Liu, Heng Zhang, and Haibo Chen. 2014. Scalable Read-mostly Synchronization Using Passive Reader-writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC).* USENIX Association, Philadelphia, PA, 219–230.

[30] Yuanhan Liu. 2014. aim7 performance regression by commit 5a50508 report from LKP. (2014). https://lkml.org/lkml/2013/1/29/84.

[31] Waiman Long. 2014. qrwlock: Introducing a queue read/write lock implementation. (2014). https://lwn.net/Articles/579729/

[32] Waiman Long. 2014. qspinlock: Introducing a 4-byte queue spinlock. (2014). https://lwn.net/Articles/582897/.

[33] Waiman Long. 2016. locking/mutex: Enable optimistic spinning of lock waiter. (2016). https://lwn.net/Articles/696952/.

[34] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06).* 801–810.

[35] Joe Mario. 2016. C2C - False Sharing Detection in Linux Perf. (2016). https://joemario.github.io/blog/2016/09/01/c2c-blog/.

[36] Dave McCracken. 2004. Object-based Reverse Mapping. In *Proceedings of the Ottawa Linux Symposium*. OLS, 6.

[37] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65.

[38] John M. Mellor-Crummey and Michael L. Scott. 1991. 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)*. 106–113.

[39] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. USENIX Association, Denver, CO.

[40] Ingo Molnar and Davidlohr Bueso. 2017. Generic Mutex Subsystem. (2017). https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[41] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. (2017). https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[42] Oleg Nesterov. 2012. Linux percpu-rwsem. (2012). http://lxr.free-electrons.com/source/include/linux/percpu-rwsem.h.

[43] Andy Patrizio. 2017. HPE refreshes its Superdome servers with SGI technology. (2017). https://www.networkworld.com/article/3236789/hpe-refreshes-its-superdome-servers-with-sgi-technology.html.

[44] Zoran Radovic and Erik Hagersten. 2003. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 241–252.

[45] Michael L. Scott. 2002. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing (PODC '02)*. New York, NY, USA, 31–40.

[46] Alex Shi. 2013. [PATCH] rwsem: steal writing sem for better performance. (2013). https://lkml.org/lkml/2013/2/5/309.

[47] Herb Stutter. 2005. The Free Lunch is Over. (2005). http://www.gotw.ca/publications/concurrency-ddj.htm.

[48] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*. USENIX Association, Portland, OR, 11–11.

[49] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, Texas.

[50] Michal Zalewski. 2017. american fuzzy lop (2.41b). (2017). http://lcamtuf.coredump.cx/afl/.

[51] Peter Zijlstra. 2010. percpu rwsem -v2. (2010). https://lwn.net/Articles/648914/.