Scalable NUMA-aware Blocking Synchronization Primitives

Sanidhya Kashyap Changwoo Min Taesoo Kim
Georgia Institute of Technology

Abstract

Application scalability is a critical aspect to efficiently use NUMA machines with many cores. To achieve that, various techniques ranging from task placement to data sharding are used in practice. However, from the perspective of an operating system, these techniques often do not work as expected because various subsystems in the OS interact and share data structures among themselves, resulting in scalability bottlenecks. Although current OSes attempt to tackle this problem by introducing a wide range of synchronization primitives such as spinlock and mutex, the widely used synchronization mechanisms are not designed to handle both under- and over-subscribed scenarios in a scalable fashion. In particular, the current blocking synchronization primitives that are designed to address both scenarios are NUMA oblivious, meaning that they suffer from cache-line contention in an under-subscribed situation, and even worse, inherently spur long scheduler intervention, which leads to sub-optimal performance in an over-subscribed situation.

In this work, we present several design choices to implement scalable blocking synchronization primitives that can address both under- and over-subscribed scenarios. Such design decisions include memory-efficient NUMA-aware locks (favorable for deployment) and scheduling-aware, scalable parking and wake-up strategies. To validate our design choices, we implement two new blocking synchronization primitives, which are variants of mutex and read-write semaphore in the Linux kernel. Our evaluation shows that these locks can scale real-world applications by 1.2–1.6× and some of the file system operations up to 4.7× in both under- and over-subscribed scenarios. Moreover, they use 1.5–10× less memory than the state-of-the-art NUMA-aware locks on a 120-core machine.

1 Introduction

Over the last decade, microprocessor vendors have been pursuing the direction of bigger multi-core and multi-socket (NUMA) machines [16, 31] to provide large chunks of memory, which is accessible by multiple CPUs. Nowadays, these machines are a norm to further scale applications such as large in-memory databases (Microsoft SQL server [26]) and processing engines [34, 41]. Thus, achieving application scalability is critical for efficiently using these NUMA machines, which today can have up to 4096 hardware threads organized into sockets. To achieve high performance, various applications such as databases [26], processing engines [34, 41], and operating systems (OS) often rely on NUMA partitioning to mitigate the cost of remote memory access either by data or task placement. However, these approaches do not address how to efficiently modify shared data structures such as inodes, dentry cache, or even the structures of the memory allocator that span multiple sockets in a large multi-core machine. As a result, synchronization primitives are inevitably the basic building blocks for such multi-threaded applications and are critical in determining their scalability [2]. Hence, the state-of-the-art locks [4, 5, 10, 11, 23, 24], which are NUMA-aware, are the apt choice to efficiently exploit the NUMA behavior for achieving scalability on these multi-core machines.

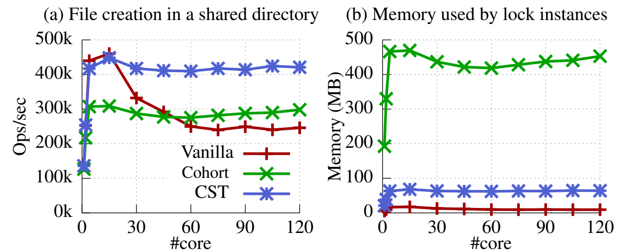

Figure 1: Impact of NUMA-aware locks on a file-system micro-benchmark that spawns processes to create new files in a shared directory (`MWCM` in [27]). It stresses either the `mutex` or the writer side of the read-write semaphore and memory allocation. Figure (a) presents the results up to 120 threads on a 120-core machine and Figure (b) shows the memory utilized by locks during the experiment. Here, *Vanilla* is Linux's native version, and Cohort is an in-kernel ported version of NUMA-aware locks [4, 11], and our NUMA-aware lock (CST).

NUMA-aware locks do improve application scalability, but they are difficult to adopt in practice. They either require application modification [5, 11, 23] or statically allocate a considerable amount of memory that can bloat shared data structures [4, 5, 11], as thousands to millions of lock instances can be instantiated in a large multi-core machine. For instance, a similar issue of adopting non-blocking queue-based locks occurred with Linux. Wickizier et al. [2] showed that a ticket lock suffers from cache-line contention with increasing core count. They replace it with the MCS lock to mitigate such an effect, which improved the system performance. Unfortunately, its adoption faced several challenges due to the change in the structure size and the lock function API [21].

We observe a similar trend in the case of blocking synchronization primitives, which suffer from numerous problems: 1) OS developers rely on TTAS locks or their variant [12, 18, 39], as they are simple and cache-line contention is not evident at smaller core count. However, they deter scalability on large multi-core machines (Figure 1 (a)). 2) The proposed blocking synchronization primitives [35, 36] are NUMA-oblivious and suffer from

high memory management cost for every lock acquisition, which impedes scalability. 3) NUMA-aware locks (Cohort locks) suffer from memory bloat as they statically allocate memory for all sockets, which is a serious issue in an OS [3] (Figure 1 (b)) and are non-blocking. 4) Finally, current blocking primitives severely suffer from the poor parking strategy because of cache-line contention, use of a global parking list, inefficient scheduling decisions, and inefficient system load estimation.

In this work, we design and implement two scalable blocking synchronization primitives, namely `CST-mutex` and `CST-rwsem`, from an OS perspective. Our primitives are memory-efficient, support blocking synchronization, and are tightly coupled with the scheduler, thereby resulting in better scalability beyond 100 physical cores for both under- and over-subscribed situations (tested up to 5× over-subscription). CST locks support blocking, as they incorporate a timeout capability for waiters, including readers and writers, in which waiters can park and wake-up without hurting the performance of the system. We use four key ideas to implement a scalable blocking synchronization primitive: First, we consciously allocate memory by maintaining a dynamic list of per-socket structures that is a basic building block of NUMA-aware locks. Second, instead of passing the lock to the very next waiter, we pass it to a not-yet-parked (still spinning) waiter, which removes the scheduler intervention while passing the lock to a waiter. Third, we keep track of the parked waiters in a separate, per-socket list without manipulating the actual waiting list maintained by the lock protocol. Lastly, we maintain a per-core scheduling information to efficiently estimate the system load. Thus, our blocking primitives improve the application performance by 1.2–1.6×, and they are 10× faster than existing blocking primitives in over-subscribed scenarios for various micro-benchmarks. Moreover, our approach uses 1.5–10× less memory compared with the state-of-the-art NUMA-aware locks.

In summary, we make the following contributions:

- **Two blocking synchronization primitives.** We design and implement two blocking synchronization primitives (`CST-mutex` and `CST-rwsem`) that efficiently scale beyond 100 physical cores.
- **Memory-efficient data structure.** We maintain a dynamically allocated list of per-socket structures that address the issue of memory bloat.
- **Scheduling-aware parking/wake-up strategy.** Our approach mitigates the scheduler interaction by passing the lock to a spinning waiter and batching the wake-up operation.
- **Lightweight schedule information.** We extend the scheduler to estimate the system load to efficiently handle both over- and under-subscription cases.

2 Background and Motivation

We first classify prior research directions into two categories: NUMA-aware locks and runtime contention management. We later give a primer on blocking synchronization primitives used in Linux.

NUMA-aware locks. NUMA-aware locks address the limitation of NUMA-oblivious locks [25] by amortizing the cost of accessing the remote memory. Most of the locks are hierarchical in nature such that they maintain multiple levels of lock [6, 10, 11, 14, 24] in the form of a tree. Inspired by prior hierarchical locks [10, 24], Cohort locks [6, 11] generalized the design of any two types of locks in a hierarchical fashion for two-level NUMA machines and later extended them for the read-write locks [4]. However, neither of them addresses the memory utilization issue nor supports blocking synchronization, which leads to sub-optimal performance when multiple instances of locks are used or when the system is overloaded. Besides Cohort locks, another category of locking mechanism is based on combining [13, 32] and the remote core execution approach [23] in which a thread executes several critical sections without any synchronization. Although it outperforms Cohort locks [23], the mechanism requires application modification, which is not practical for applications with a large code base.

Our design of NUMA-aware locks is memory conscious, as we defer the allocation of per-socket locks until required, unlike prior ones. In addition, CST locks are blocking, meaning that they support timeout capability while maintaining the locality awareness, unlike the existing NUMA-oblivious locks that allocate memory for each lock acquisition [35, 36]. Moreover, none of the NUMA-aware read-write locks support blocking readers, but the ones that do support [19, 28, 30] are NUMA oblivious and are designed specifically for read-mostly operations.

Contention management. The interaction between lock contention and thread scheduling determines application scalability, which is an important criterion to decide whether to spin or park a thread in an under- or over-subscribed scenario. Johnson et al. [17] addressed this problem by separating contention management and scheduling in the user space. They use admission control to handle the number of spinning threads by running a system-wide daemon that globally measures the load on the system. Similar approaches have been used by runtimes [7] and task placement strategies inside the kernel without considering the lock subsystem [42]. Along these lines, the Malthusian lock [9], a NUMA-oblivious lock, handles thread over-subscription by randomly moving a waiter from an active list to a passive list (concurrency culling), which is inspired by Johnson et al.

CST locks handle the over-subscription by maintaining a separate list in which waiters independently add them-

selves to a separate list after timing out. Our approach is different from the Malthusian lock and does not lengthen the unlock phase because wake-up and parking strategies are independent. Moreover, CST locks adopt the idea of a separate parking list from existing synchronization primitives [28, 29] or wait queues [38], but remove the cache-line bouncing by maintaining a per-socket, separate parking list for both readers and writers.

Design of Linux's `mutex` and `rwsem`. Many OSes, including Linux, do not allow nested critical sections for any blocking locks. The current design of `mutex` is based on the TTAS lock, which is coupled with a global queue-based instance [22] and a parking list per-lock instance. The algorithm works by first trying to atomically update the lock variable, called fast path; on failure, the mid-path phase (optimistic spinning) begins in which only a single waiter is queued up if there is no spinning waiter and optimistically spins until its schedule quota expires. If the waiter still does not acquire the lock, it goes to the slow-path phase in which it acquires a lock on the parking list (parking lock), adds itself, and schedules out after releasing the parking lock. During the unlock phase, the lock holder first resets the TTAS variable and wakes up a waiter from the parking list while holding the parking lock. Meanwhile, it is possible that either a new waiter can acquire the lock in the fast path or a spinning waiter in the mid path. Now, once a waiter is scheduled in, it again acquires the parking lock and tries to acquire the TTAS lock. If successful, it removes itself from the parking list and enters the critical section; otherwise, it schedules itself out again and sleeps until a lock holder wakes it up. The current algorithm is unfair because of the TTAS lock; even starves its waiters in the slow-path phase. Moreover, the algorithm also suffers from cache-line contention because of the TTAS lock and waiters maintenance, and even worse is the scheduling overhead in the slow-path phase and the unlock phase for parking and wake up.

The read-write semaphore is an extension of `mutex`, with a writer-preferred version. Both the write lock and the reader count are encoded in a word to decide readers, writer, and waiting readers. Moreover, `rwsem` maintains a single parking list in which both readers and writers are added. Thus, in addition to inheriting the issues of `mutex`, `rwsem` also suffers from reader starvation due to the writer-preferred version. Interestingly, developers found that the neutral algorithm suffers from scheduler overhead [20], while the writer-preferred version mitigated this overhead and improved the performance by 50% [37].

3 Challenges and Approaches

We present challenges and our approaches in designing practical synchronization primitives that can scale beyond 100 physical cores.

C1. NUMA awareness. A synchronization primitive should scale under high contention even in NUMA machines. Although locks that are used in practice [18, 28, 29] address cache-line contention by using queue-based locks [22] for high contention, they do not address the cache-line bouncing (remote socket access) introduced in NUMA machines. The remote access is at least $1.6\times$ slower than the local access within a socket, which is a deterrent to the scalability of an application.

> **Approach:** To achieve scalability in NUMA machines, hierarchical locks (e.g., Cohort lock) are an apt choice. They mitigate the cache-line bouncing by passing a lock within a socket, which relaxes the strict fairness guarantee of FIFO locks for throughput.

C2. Memory-efficient data structures. Unfortunately, current hierarchical locks severely bloat the memory due to their large structure size (e.g., a Cohort lock requires 1,600 bytes in an eight-socket machine[1]), which they statically allocate for all sockets that may be unused. Memory bloat is a serious concern because it stresses the memory allocator and is alarming for synchronization primitives as they statically allocate the memory. For example, the size of the `XFS` inode structure increased by 4% after adding 16 bytes to the `rwsem` structure, which had an impact on the footprint and performance, as there can be millions of inodes cached on a system [8]. Thus, existing hierarchical locks are difficult to adopt in practice because they statically allocate per-socket structures during initialization.

> **Approach:** A hierarchical lock should dynamically allocate per-socket structure only when it is being used to avoid the memory bloat problem and reduce the memory pressure on a system.

C3. Effective contention management for both over- and under-subscribed scenarios. Designing synchronization primitives that perform equally well for both over- and under-subscribed situations is challenging. Non-blocking synchronization primitives, such as spinlocks including Cohort locks, work well when a system is under-loaded. However, for an over-loaded system, they perform poorly because spinning waiters and a lock holder contend each other, which deters the progress. On the other hand, blocking synchronization primitives such as `mutex` and `rwsem` are designed to handle an over-loaded system. Instead of spinning, waiting threads sleep until a lock holder wakes one up upon lock release. However, this procedure imposes the overhead of waking up in every unlock operation, which increases the length of the critical section. Also, frequent sleep and wake-up operations impose additional overhead on the scheduler, which can result in scalability collapse, especially when multiple lock instances are involved. To mitigate this issue, many blocking synchronization primitives [18, 28, 29] employ

[1] 64-byte cache line size $\times$ (3 cache lines for the socket lock $\times$ 8 sockets + 1 cache line for the top lock).

the *spin-then-park* strategy: a waiter spins for a while, and then parks itself out. Unfortunately, this approach is agnostic of system-wide contention, which leads to suboptimal performance when multiple locks are contending. Ryan et al. [17] addressed the problem by designing a system-wide load controller, but its centralized design has memory hot spots for its control variables (e.g., the number of ever-slept threads) to decide whether a thread should sleep or spin.

Approach: To work equally well in both over- and under-loaded cases, we must address the system-wide load that allows waiters to optimistically spin in under-loaded cases and park themselves out in over-loaded cases. In addition, such a decision should be taken in a distributed way to keep the contention management from becoming a scalability bottleneck.

C4. Scalable parking and wake-up strategy. To implement an efficient blocking synchronization primitive, the most important aspects are how and when to park (schedule out) and wake up waiters with minimal overhead. The current approach [28, 29] maintains a global parking list to keep track of parked waiters and a lock holder wakes one of the parked waiters at the unlock operation. However, this design has several drawbacks: The frequent updating of a global parking list becomes a single point of contention in an over-loaded system, which leads to severe performance degradation because a lock holder has to wake up each sleeping waiter during the unlock phase, which adds extra pressure on the scheduler subsystem and lengthens the critical section: the cost of waking up varies from 2,000–8,000 cycles in the kernel-space or from 5,000–50,000 cycles in the user-space (`futex()` overhead). Thus, according to Amdahl's Law, an increased sequential part can significantly affect the scalability, especially in a large multi-core machine.

Approach: Instead of waking up the very next waiter, a lock holder passes the lock to a non-sleeping waiter, if any. Thus, this approach not only avoids waking up other threads under high contention, but also minimizes the access of the parking list and scheduler interactions. Furthermore, we maintain a per-socket parking list to remove costly cache-line bouncing among NUMA domains for accessing the parking list.

4 Design Principles

We present two scalable NUMA-aware blocking synchronization primitives, a mutex (`CST-mutex`) and a read-write semaphore (`CST-rwsem`), that can scale beyond 100 physical cores. At a high level, our lock is a two-level NUMA-aware lock, where a global lock is an MCS lock [25] and a per-socket local lock is a K42 lock [15] (see Figure 2). While the first level localizes the cache-line contention within a socket, the second one mitigates the cache-line bouncing among sockets. To enter a critical section, a thread first acquires the per-socket local lock and then the global lock. During the release phase, it first releases the global lock followed by the local lock. To mitigate memory bloating, we dynamically allocate the per-socket structure (`snode`) when a thread first tries to acquire the lock on a specific NUMA domain, and maintain it until the life-cycle of the lock. Each `snode` maintains a per-thread `qnode` in two lists: `waiting_list`—a K42-style list of waiters and `parking_list`—a list of parked (or sleeping) waiters. To acquire the lock, a thread first appends its `qnode` to the `waiting_list` of the corresponding `snode` in a `UW` (unparked waiting or spinning) status (`T3` in Figure 2) and spins until its schedule quota is over. On timing out, `T3` parks itself by changing its status to `PW` (parked waiting) and adds itself to the `parking_list` (`T2`). A lock holder (`T1`) that acquires its local and the global lock, passes the lock in the same NUMA domain by traversing the `waiting_list` during the release phase. It skips the parked waiter (`T2`) and passes the lock to an active waiter (`T3`). If there is no active waiter, the lock holder wakes up parked waiters in the same or other NUMA nodes to pass the lock. Our `rwsem` additionally maintains a separate reader parking list, besides writer parking list, to handle the over-subscription of the readers.

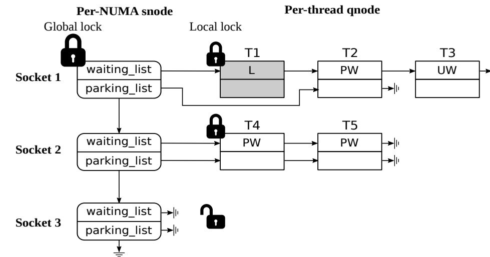

Figure 2: A `CST-mutex` is active on sockets: 1, 2, and 3. Currently, socket 1 is being served. `T1` now holds the lock (`L`); `T3` is spinning for its turn (`UW`: unparked waiting); `T2`, `T4`, and `T5` are sleeping (`PW`: parked waiting) until a lock holder wakes them up. A lock holder, `T1`, will pass the lock to `T3`, which is spinning, skipping the sleeping `T2`, to minimize the overhead of wake-up.

We explain our design principles on efficient memory usage (**C1** and **C2** in §4.1) and parking/wake-up strategy (**C3** and **C4** in §4.2). We later show how to apply our approaches to design blocking synchronization primitives: `CST-mutex` (§5.1) and `CST-rwsem` (§5.2).

4.1 Memory-efficient NUMA-aware Lock

Unlike other hierarchical locks that statically allocate per-NUMA structures for all sockets during the initialization, CST defers the `snode` allocation until the moment it is accessed first. The allocated `snodes` are active until the lock is destroyed. Our dynamic allocation of `snode` is especially beneficial in two cases: 1) when the number of objects is unbounded, such as `inode` and `mm_struct` in Linux kernel,[2] and 2) when threads are restricted to access

[2]The static allocation of all `snodes` increases the `inode` structure size by 3.8× and `mm_struct` size by 2.6× in an eight-socket machine.

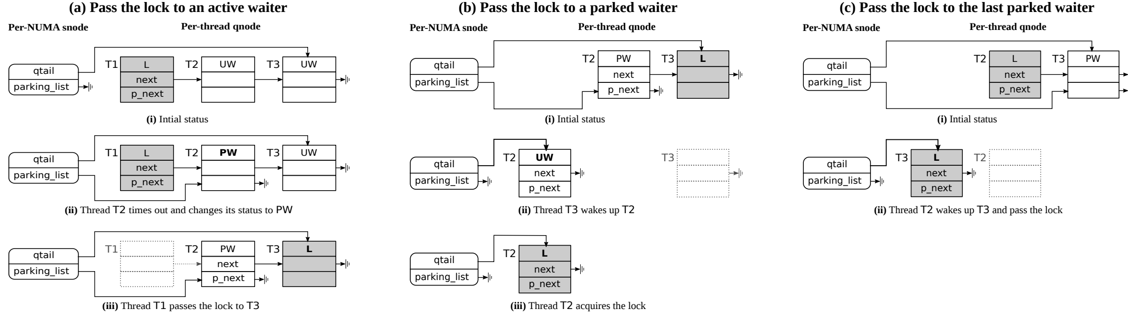

Figure 3: Figure (a) shows the passing of a lock to a spinning waiter inside a per-socket structure (snode). (i) T1 is the current lock holder, and T2 and T3 are in the waiting_list, and qtail points to the qnode of T3. (ii) T2 times out, successfully CASes its state from UW to PW, and adds itself to the parking_list. (iii) T1 exits the critical section. It tries to pass the lock to T2, but fails to CAS the state of T2 from UW to L. T1 goes to T3 via next pointer of T2, successfully CASes the state of T3 from UW to L, and leaves the unlock phase. Figure (b) shows the passing of the lock to a parked waiter in the parking_list. (i) T3 (lock holder) is in the unlock phase. It finds that waiting_list is empty as T2 is in the parking_list. T3 successfully CASes qtail to NULL. (ii) Now, T3 checks for parked waiters in parking_list, finds T2, and updates the state of T2 from PW to R. (iii) Since stail is NULL and there are no prior waiters, T2 sets its state to L and acquires the local lock, and later goes to acquire the global lock. Figure (c) illustrates the passing of the lock to a parked waiter at the end of the waiting_list. (i) On exiting the critical section, T2 fails to CAS the state of T3 to L, since it is parked. (ii) T2 then explicitly SWAPs the state of T3 to L and wakes it up. T3 now holds the local lock and goes to acquire the global lock.

a subset of sockets such as running a multi-core virtual machine on a subset of sockets in a cloud environment.

For every lock operation, we first check whether a corresponding snode is present, and then get the snode to acquire the local lock. To efficiently determine whether an snode is present, a lock maintains a global bit vector in which each bit denotes the presence of a particular snode. Hence, each thread relies on the bit vector for determining the presence of an snode. We use CAS to atomically update the bit vector, but the number of CAS operations is bounded to the number of sockets in a system during the lifetime of a lock. A lock maintains allocated snodes in snode_list, which is traversed by a thread to find the corresponding snode. We separate the snode into two cache lines, almost-read-only for snode traversal and read-write for the local lock operation, which prevents snode traversal from incurring cache-line bouncing among sockets.

4.2 Scheduling-aware Parking/Wake-up Strategy

As discussed in the previous section, the most widely used *spin-then-park* policy fails to address the issue of scalability in NUMA machines. It works by maintaining a single, global parking list to account for the sleeping waiters, and wakes one or some waiters to pass the lock at the time of release. Hence, this approach is not scalable because it incurs contention on the parking list and suffers from scheduler interaction as it passes the lock to a potentially sleeping waiter in an over-subscribed condition.

To address these issues, the CST lock uses two key ideas: it maintains a per-socket parking_list, which minimizes costly cross-socket cache-line bouncing and passes the lock to a spinning waiter, whose time quota is not over yet, to minimize costly wake-up operations. We wake up a set of skipped sleeping waiters in bulk when there are no active waiters in the serving snode or pass the global lock to the other waiting snode. Thus, by relaxing the strict FIFO guarantee, we mitigate the lock-waiter preemption problem.

4.2.1 Low-contending List Management

In a CST lock, each snode maintains the K42-style waiting list that comprises its own tail pointer: qtail. For parked waiters, the snode also maintains a per-socket parking_list to account for the parked waiters, which avoids the costly cache-line bouncing while manipulating the parking_list. For a rwsem, we maintain a separate readers and writers parking_list, which simplifies the list processing in the unlock phase, as the lock holder can pass the lock to all parked readers or to one of the writers. Moreover, this approach enables a distributed parallel waking of readers at a socket level, which can improve the throughput of readers in an over-subscribed scenario (refer §5.2).

4.2.2 Scheduling-aware Parking/wake-up Decision

For a blocking synchronization primitive, the most important question is how to efficiently pass the lock or wake up a waiter, while maintaining an on-par performance in both the under- and over-subscribed cases. For the scalable parking/wake-up decision, we remove costly scheduler operations (i.e., wake-up) from the common, critical path and employ a distributed parking decision while considering the load on a system. We discuss three key ideas to address the problem of 1) whom to pass the lock to, 2) when to park oneself, and 3) how to take the parking decisions for blocking synchronization primitives.

Passing lock to an active spinning waiter. In queue-based locks (e.g., MCS, K42, and CLH), the successor of a lock holder always acquires the lock, which guarantees complete fairness, but, unfortunately, causes severe perfor-

mance degradation in an over-subscribed system, as this invariant stresses the scheduler to always issue a call to wake up the parked waiter. To mitigate this issue, we modify the invariant of a succeeding lock holder from the next waiter to a nearest *active* waiter, which is still spinning for the lock acquisition. Hence, the `waiting_list` comprises both active and parked waiters in its queue, and the parked waiters are added to a separate list: `parking_list`. Figure 3 (a) illustrates this scenario, where `T1` passes the lock to `T3` instead of `T2`, since `T2` is parked. Later, parked waiters are woken up in batches up to the number of physical cores in a socket once there is no active waiter in the `waiting_list`. When a parked waiter is woken up, it generally re-queues itself back at the end of the `waiting_list`, and again actively spins for the lock. This approach is effective because we can avoid scheduler intervention under high contention by passing the lock to an active waiter. In addition, a batched wake-up strategy amortizes the cost of the wake-up phase.

Scheduling-aware spinning. Current hierarchical locks [4, 6, 11] do not consider the amount of time a waiter should spin before parking itself out. Thus, in an over-loaded system, waiting threads and a lock holder will contend with each other, which deters the system progress. Instead, in CST locks, waiting threads park themselves as soon as their time quota is about to cease. To check the quota, we rely on the scheduler and its APIs for this information. Specifically in the Linux kernel, the scheduler exposes `need_resched()` to know whether the task should run, and preemption APIs (`preempt_disable() / preempt_enable()`) to explicitly disable or enable the task preemption. These APIs work with both preemptive and non-preemptive kernels. Limiting the duration of spinning up to the time quota proposed by the scheduler has several advantages: 1) It guarantees the forward progress of the system in an over-loaded system by allowing the current lock holder to do useful work while mitigating its preemption. 2) It allows other tasks to do some useful work rather than wasting the CPU cycles. 3) By only spinning for the specified duration, the primitive respects the fair scheduling decision of the scheduler.

Scheduling-aware parking. The current blocking synchronization primitives [28, 29] do not efficiently account for the system load; thus, they naively park waiters even in under-loaded scenarios. Hence, a naive use of the spin-then-park approach results in scheduler intervention, as the waiters park themselves as soon as their time quota ceases, and the lock holder has to do an extra operation of waking them up, which severely degrades the performance of the system in an under-loaded scenario [27]. Also, previous research [17] has shown that estimating system load is critical to the spin-then-park approach because it not only removes the scheduler interaction from the parking phase, but also improves the latency of the lock/unlock phase.

We gauge the system load by peeking at the number of running tasks on a CPU (i.e., the length of scheduling run queue for a CPU). Checking the number of running tasks is almost free because a modern OS kernel, including Linux, has a per-CPU scheduler queue, which already maintains an up-to-date per-CPU active task information. On the other hand, maintaining system-wide, central information, like the approach used by Johnson et al. [17], is costly because the cost of collecting the total number of active tasks increases with increasing core count, which may not catch the load imbalance due to the new incoming tasks or the rescheduling of periodic tasks.

5 Scalable Blocking Synchronizations

We now discuss the design and implementation of the two types of NUMA-aware blocking synchronization primitives (`mutex` and `rwsem`) using our design decisions. We first present the design of `mutex` (`CST-mutex`) along with the parking strategy and later extend it to `rwsem` (`CST-rwsem`). Figure 4 presents their pseudo-code.

5.1 Mutex (`CST-mutex`)

`CST-mutex` is a two-level hierarchical lock, which is extended to support blocking behavior by adding several design choices, such as scheduling-awareness, efficient spinning and parking strategy, and passing of the lock to the spinning waiter. The global lock employs an MCS lock, whereas the local lock is a K42 lock [15], a variant of the MCS lock. We choose the K42 lock because it does not require an extra argument in the function call as it maintains a `qnode` structure on the stack, but we can use any queue-based lock for the local lock. The top level lock maintains a dynamically allocated per-socket structure (`snode`) to keep track of the global lock and local lock information such as its `waiting_list` and the next waiter (for the K42 lock), and also `parking_list` information for the parked waiters. The MCS lock protocol has two status values: waiting (lock waiter) and locked state (lock holder). To support the blocking behavior, we keep the locked state (denoted as `L`) intact and extend the waiting state to the spinning/unparked (`UW`) and parked (`PW`) state. We also introduce a special state, called re-queue (`R`), that notifies the waiter to re-acquire the local lock.

Extended Cohort lock/unlock protocol. A thread starts by trying to acquire a local lock inside a socket. If there are no predecessors during the lock acquisition, it acquires the global lock, thereby becoming the lock holder, and enters the critical section (CS). The other threads that do not acquire the local lock are the local waiters, and the ones waiting for the global lock are the socket leaders. They wait for their respective predecessor to pass the lock. In the release phase, the lock holder locally tries to pass the lock to a successor. Thus, on success, the successor

```
1   def mutex_lock(lock):
2       snode = find_or_add_snode(lock) # Find or allocate snode once
3       while True:
4           lock_status = acquire_local_lock(snode)
5           if lock_status & ACQUIRE_GLOLBAL_LOCK is True: # Acquire global lock?
6               acquire_global_lock(lock, snode)
7               return
8
9   def acquire_local_lock(snode):
10      cur_qnode = init_qnode(status=UW, next=None) # Initialize qnode on the function stack
11      pred_qnode = SWAP(&snode.qtail, &cur_qnode) # Add to snode's waiting list
12      if pred_qnode is None: # Check for predecessor
13          cur_qnode.status = L|ACQUIRE_GLOLBAL_LOCK # Should acquire global lock
14          return cur_qnode.status
15      pred_qnode.next = &cur_qnode # Update predecessor next pointer
16      cur_qnode.task = current_task
17      while cur_qnode.status == UW: # Spinning for the local lock
18          if task_timed_out(cur_qnode.task): # Time quota is over
19              if park_write_qnode(snode, cur_qnode) == REQUEUE: # Check for requeue state
20                  if cur_qnode.status == L: # Local lock acquired
21                      break
22                  else:
23                      return R # Restart the local lock acquisition
24      update_next_qnode(snode, cur_qnode) # Update the next qnode (k42 protocol)
25      return cur_qnode.status
26
27  def acquire_global_lock(lock, snode):
28      snode = init_snode(snode, status=UW, next=None) # Initialize snode
29      pred_snode = SWAP(&lock.stail, &snode) # Add to global lock's waiting list
30      if pred_snode is None:
31          snode.status = L  # Acquired global lock
32          return
33      pred_snode.next_snode = snode # Update predecessor next pointer
34      snode.leader_task = current_task
35      while snode.status == UW: # Spin till the global lock holder passes the lock
36          if task_timed_out(current_task): # Leader time quota is over
37              if CAS(&snode.status, UW, PW): # Modify the state to PW
38                  schedule_out(snode.leader_task) # Schedule out the task
39      lock.current_serving_socket = snode
40
41  def mutex_unlock(lock):
42      snode = lock.current_serving_socket # Get the lock holder's snode #
43      if snode.local_batch_count < BATCH_COUNT: # local lock batching
44          snode.local_batch_count += 1
45          # Pass the lock to waiter with UW state and already has the global lock
46          if pass_local_lock(snode, acquire_global=False) is True:
47              return # Successfully found an active waiter
48      snode.local_batch_count = 0 # Reset the batch count
49      release_global_lock(lock, snode) # Release the global lock
50      release_local_lock(lock, snode) # Release the local lock
51      if snode_parking_list_is_not_empty(snode): # Remove parked waiter starvation
52          wake_up_parked_waiters(snode) # Wake up set of parked waiters
53
54  def release_local_lock(lock, snode):
55      if snode.qnext is None: # Check for next qnode, if any
56          if CAS(&snode.qtail, &snode.qnext, None) is True: # No qnode present
57              wake_up_parked_waiters(snode) # Wake up set of parked waiters
58          while snode.qnext is None: # qnode joined the qtail (waiting)
59              continue
60      if pass_local_lock(snode, acquire_global=True) is False:
61          with parking_list_lock(snode): # Acquire parking list lock to wake up a waiter
62              snode.qnext.status = L|ACQUIRE_GLOLBAL_LOCK # Update status
63              remove_from_parking_list(snode.qnext) # Update the parking list
64              schedule_in(snode.qnext.task) # Wake up the parked waiter
65
66  def release_global_lock(lock, snode):
67      if snode.next_snode is None: # Check for next snode, if any
68          if CAS(&lock.stail, snode, NULL) is True: # No snode present
69              return
70          while snode.next_snode is None: # Some snode joined the global lock stail
71              continue
72      if CAS(&snode.next_snode.status, UW, L) is False: # Check for parked snode
73          snode.next_snode.status = L # next snode is parked, still pass the lock
74          schedule_in(snode.next_snode.leader_task) # Wake it up for global lock acquisition
```

```
75  def park_write_qnode(snode, cur_qnode):
76      park_flag = False # Denotes whether waiter is parked
77      with parking_list_lock(snode): # Acquire parking list lock
78          if CAS(&cur_qnode.status, UW, PW) is True: # Try to update the state
79              add_to_parking_list(snode, cur_qnode) # Update parking list
80              park_flag = True # Parking was successful
81      if park_flag is True:
82          schedule_out(cur_qnode.task) # Schedule the task out
83          # cur_qnode.task is now awake, the task now returns REQUEUE
84          return REQUEUE # Should check for requeue phase
85      else:
86          return DO_NOT_REQUEUE # Acquired the lock
87
88  def pass_local_lock(snode, acquire_global):
89      qnode = snode.qnext # Search from snode.next
90      while True:  # Search for an active waiter
91          if CAS(&qnode.status, UW, L) is True:
92              if acquire_global is True: # Need to acquire the global lock
93                  L = L|ACQUIRE_GLOLBAL_LOCK # Update L status bit
94              return True
95          if qnode.next is None:
96              break
97          qnode = qnode.next # Find next qnode
98      snode.qnext = qnode # Found no one, updating qnext with tail
99      return False
100
101 def wake_up_parked_waiters(snode):
102     with parking_list_lock(snode): # Acquire the parking list
103         for qnode in parking_list(snode): # Iterate over stored parked waiters
104             qnode.status = R # All waiter should requeue right now
105             remove_from_parking_list(snode, qnode) # Update parking list
106             schedule_in(qnode.task)  # Schedule in the waiter
107
108 def write_lock(lock):
109     mutex_lock(lock)  # Acquire mutex first
110     for s in snode_list(lock):  # Check for active readers
111         while s.active_readers is not 0:
112             if task_timed_out(current_task):
113                 schedule() # Only schedule, will come back
114
115 def write_unlock(lock):
116     mutex_unlock(lock)  # Release the mutex
117     if lock.stail is None:  # There is no waiting snode
118         for s in snode_list(lock): # Traverse the snode
119             wake_up_first_read_waiter(s.reader_parking_list) # Wake-up a reader
120
121 def read_lock(lock):
122     snode = find_or_add_snode(lock) # Find or allocate the snode
123     ret = True
124     while True: # Spin, till acquired the lock
125         while lock.stail is not None: # Check for no waiters
126             if task_timed_out(current_task):
127                 ret = park_reader_task(lock, snode) # park the reader
128         if ret is True:
129             FAA(&snode.active_readers, 1)
130         if lock.stail is not None: # No one in the global lock tail
131             FAA(&snode.active_readers, -1)
132             ret = True
133             continue
134         break
135
136 def read_unlock(lock):
137     snode = find_or_add_snode(lock)
138     FAA(&snode.active_readers, -1) # Update the snode readers count.
139
140 def park_reader_task(lock, snode):
141     # Wait and park youself until global lock tail is NULL
142     park_and_wait_on_event(&snode.reader_parking_list, (lock.stail is not None))
143     if CAS(&snode.reader_is_parked_leader, False, True) is True:
144         FAA(&snode.active_readers, 1) # Decrease the active reader count
145         wake_up_all_read_waiters(&snode.reader_parking_list) # Wake up all readers
146         snode.reader_is_parked_leader = False
147         return False
148     return True
```

Figure 4: Pseudo-code of `CST-mutex` (lines 1 – 74), `CST-rwsem` (lines 108 – 148), and their parking/wake up (lines 75 – 106). We use three atomic instructions: `CAS(addr,new,old)` atomically updates the value at `addr` to `new` and returns `True` if the value at `addr` is `old`. Otherwise, it returns `False` without updating `addr`. `SWAP(addr,val)` atomically writes `val` to `addr` and returns the old value at `addr`. `FAA(addr,val)` atomically increases the value at `addr` by `val`.

does not acquire the global lock and immediately enters the critical section. To prevent starvation, a lock holder later passes the global lock to a globally waiting successor (socket leader) after a bounded number of local acquisitions. We now describe the `CST-mutex` protocol in detail, which is an extension of the aforementioned steps.

Acquire local lock: A thread `T` starts by first finding (or adding if not present) its `snode` (line 2). Unlike the Cohort lock protocol, `T` tries to acquire the local lock (line 4) in an infinite `for` loop because it may restart the protocol after being parked. In the local lock phase, `T` initializes its `qnode` (line 10) and then `SWAP`s the `qtail` of `snode` with `qnode`. It then acquires the global lock when no waiters are present. Otherwise, `T` spins on its status, which changes to either the `L` or `R` state (line 17). While waiting, `T` initiates the parking protocol (lines 75 – 86) on timing out, where it tries to `CAS` the status of `qnode` from `UW` to `PW`. `T` returns back on failure; otherwise, it adds itself to the `parking_list` and schedules out. Later, when a lock holder wakes it up, it resumes (line 83) and either acquires the local lock or restarts the protocol, depending on its updated status. If `T` has `L` status after being woken up, it goes on to acquire the global lock as the previous lock holder releases the global lock before waking up sleeping waiters. To mitigate cache-line bouncing, `T` checks for the global lock flag (line 5). If not set, `T` already holds the

global lock, or else it goes to acquire it.

Acquire global lock: T initializes its snode (line 28) and adds itself to the waiting_list (line 29). It then acquires the global lock if there is no waiter (line 31), or waits until its predecessor snode passes the lock (line 35). On timing out, while spinning (line 35), T CASes status of snode from UW to PW and schedules out (line 38); otherwise, it acquires the lock as the predecessor passed the lock. Note that even after being woken up, T always acquires the global lock without re-queueing itself.

Release local lock: T gets the current snode (line 42) and tries to locally pass the lock if it is within the batching threshold (line 43). To locally pass the lock, T first tries to CAS the status of its successor from UW to L. On success, the unlock phase is over; otherwise, it traverses the waiting_list to find an actively running waiter (lines 88 – 99). Figure 3 (a) illustrates this scenario, where T1 ends up passing the lock to T3 since T2 has PW state. Note that if all waiters are parked, (line 99), T releases the global lock (line 49) and then the local lock (line 50). T can also initiate both release phases when an snode exceeds the batching threshold. In the local unlock phase, T finds the snode qnext pointer to pass the lock. If qnext is NULL, T updates the qtail of snode with NULL (line 68) and wakes up waiters in the parking list to the R state to re-queue them back to the waiting_list (line 101). Figure 3 (b) illustrates the scenario, where T3 is the last one in the waiting_list. In the release phase, after resetting qtail to NULL, T3 wakes up parked T2 after updating its status from PW to UW. If there are waiters (lines 60 – 64), then T again tries to pass the lock to a spinning waiter in the waiting_list (line 88). If successful, a waiter acquires the local lock and then goes for the global lock since T has already released that one. If all, including the last waiter, are parked (lines 60–64), T passes the local lock to the last waiter and wakes it up because T cannot reset the qtail pointer, as there maybe some parked waiters; hence, passing the lock to the last waiter is mandatory. Figure 3 (c) shows this scenario in which T2 is about to release the local lock and finds that T3 is the last one and has PW status. T2 has to wake up T3 with an L state (not R), so that T3 can maintain the K42/MCS lock protocol.

Release global lock: The protocol differs from the MCS protocol for passing the lock. For an existing snode successor, thread T tries to CASes the status of its succeeding snode from UW to L. If successful, the lock is passed; otherwise, T explicitly updates the status to L and wakes up the succeeding socket leader (lines 72 – 74).

5.2 Read-write Semaphore (CST-rwsem)

CST-rwsem is a writer-preferred version of the Cohort read-write lock [4] (CST-rwsem) with two extensions: 1) application of our parking strategy to the readers and 2) our own version of the mutex algorithm (§5.1). It relaxes the condition of acquiring the CS by multiple threads in a *read mode*. Hence, it maintains an active reader count (active_readers) on *each* snode to localize the contention on each socket at the cost of increasing the latency for the writers. We further extend the snode to support the parking of readers by maintaining a separate parking list for them, which allows readers to separately park themselves without intervening with the writers.

Write lock: Thread T first acquires the CST-mutex (line 109). Then T traverses all snodes to check whether the value of active_readers is zero (line 111). Due to our writer-preferred algorithm, T blocks new readers from entering the CS because they can only proceed if there is no writer. Once the writer has acquired the mutex lock, it does not park itself, as this is a writer-preferred algorithm and the writer will soon enter the CS (lines 110 – 113).

Read lock: T first finds its snode (line 122) and waits until there are no writers (line 125). On timing out, while waiting, T adds itself to the parking_list and schedules itself out until there are no writers (line 142). The last writer wakes up the first reader in the parking_list, which wakes up remaining sleeping waiters in its own socket. Lines 143 – 146 present the waking up of the parked reader and subsequent readers.

Write unlock: T first releases the writer lock (line 116). If there are no writers (line 117), then T checks for any sleeping waiters across all snodes. If there are any, it wakes up the only very first waiter, which will subsequently wake up remaining waiters to acquire the read lock (line 119). This approach has two advantages: 1) it ensures distributed, parallel wake-up of the readers, and 2) it does not lengthen the writer unlock phase along with the least number of remote memory accesses.

Read unlock: Thread T searches for its snode from the list of existing sockets and atomically decreases the active_readers count by 1. T does not have to wake up any writer because our approach does not park the writer thread, which is going to be the next lock holder.

6 Implementation

We implemented CST locks on the Linux kernel v4.6 and v4.7. We also provide a destructor API to reclaim the snode memory while destroying a data structure (e.g., destroy_inode for inode). For our evaluation, we modified the inode structure to use our CST-rwsem in v4.7 and CST-mutex in v4.6 since mutex was replaced with rwsem from v4.7 [40]. We also modified the virtual memory subsystem (mmap_sem) that manipulates the virtual memory area of a process. We modified 650 and five calls for mmap_sem and inode, respectively. In total, our lock implementation comprises 1,100 lines of code and can substitute most of the lock instances in Linux.

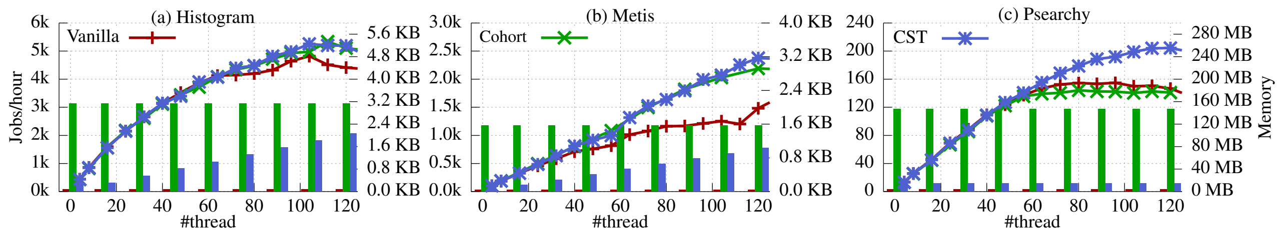

Figure 5: Impact of synchronization primitives on the scalability and memory utilization for three applications: (a) Histogram, (b) Metis, and (c) Psearchy with Linux's native `rwsem` (Vanilla), Cohort read-write lock, and `CST-rwsem`.

7 Evaluation

We evaluate the impact of CST locks by answering the following questions:

- How do locks affect the scalability and memory utilization of real-world applications? (§7.1)
- What is the impact of locks on operations provided by the OS in various scenarios? (§7.2)
- How does each design aspect help improve the performance? (§7.3, §7.4)

Evaluation setup. We evaluate CST locks on three workloads [1, 33] in an under-subscribed scenario, three micro-benchmarks from FxMark [27] that stress various file system components and the kernel memory allocator. Finally, we breakdown the performance implication of each design aspect using a hash table micro-benchmark. We evaluate on an eight-socket, 120-core machine with Intel Xeon E7-8870 v2 processors.

7.1 Application Benchmarks

We evaluate the scalability of `CST-rwsem` on three applications, namely Histogram [33], Metis [1], and Psearchy [1], that scale with increasing core count and stress the memory subsystem of the Linux kernel at varying levels. We compare our lock with the Linux's `rwsem` and an in-kernel port of Cohort locks [14]. For each benchmark results, we use *Vanilla* for the native Linux's `rwsem`, *Cohort* for the read-write Cohort lock, and CST for the `CST-rwsem` lock.

Histogram is a MapReduce application, which is page-fault intensive. It mmaps an 11 GB file at the beginning and keeps reading this file while each thread performs a simple computation. Figure 5 (a) shows that NUMA-aware Cohort and CST locks outperform the native implementation after 60 cores. They scale better because both locks localize the number of active readers within a socket, thereby having almost negligible contention across the sockets. Moreover, both locks have 2% idle time because the Cohort lock is non-blocking by design and the `CST-rwsem` effectively behaves as a non-blocking lock. On the other hand, the vanilla version is idle 10.5% of the time because of its ineffective parking strategy even in the under-subscribed situation. In summary, both locks outperform the native `rwsem` by 1.2× at 120 cores.

Metis is a mix of page-fault and mmap operation workload. It runs a worker thread on each core and mmaps 12 GB of anonymous memory for generating tables for map, reduce, and merge phases. Figure 5 (b) shows that both Cohort and CST locks outperform the original version by 1.6× as soon as the frequency of the write operation increases. Since the Cohort lock is non-blocking, it does not sleep, whereas the CST lock efficiently handles the under-subscribed case by not parking the threads, resulting in only 0.5% of idle time. Moreover, both locks batch readers, which improves the throughput of the workload. On the other hand, the original `rwsem` has 39% of the idle time because of its naive parking strategy and is 1.6× slower than the others at 120 cores.

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 small and large mmap/munmap operations from 96,000 files with multiple threads, which taxes the writer side of the `rwsem` in the memory subsystem as well as the allocation of the inodes for those files in the virtual file system layer. Figure 5 (c) shows that `CST-rwsem` outperform both the Cohort and native locks by 1.4× at 120 cores. Cohort locks suffer from the static allocation because the kernel has to allocate 96,000 inodes for reading files into a per-core hash table of Psearchy, which not only stresses the memory allocator with large objects, but also suffers from ineffective scheduling because of the involvement of multiple instances of locks. Like prior workloads, the native lock suffers from the scheduler intervention after 45 cores, as it spends up 54.4% being idle, whereas the `CST-rwsem` is only idle for 11.4% of the time.

Summary. Figure 5 shows the impact of scheduler intervention with increasing contention between readers and writers. With our efficient spinning strategy that checks its local load, CST locks have the same benefit as Cohort locks in the case of a highly contended but under-subscribed system. While Cohort locks improve the scalability of applications in highly contended and under-subscribed scenario, they hamper the scalability of applications that allocate multiple instances of locks (Figure 5 (c)). Unlike Cohort locks, CST locks consciously allocate memory with increasing *socket* count, which saves up to 10× of memory for each workload on a single socket, and 1.5 – 9.1× at 120 cores. Thus, CST locks show that dynamic allocation is beneficial to real applications, while mitigating the memory bloat issue and maintaining an on-par performance.

7.2 Over- And Under-subscribed Cases

We compare the performance of CST locks with the kernel and Cohort locks in both an over- and under-subscribed

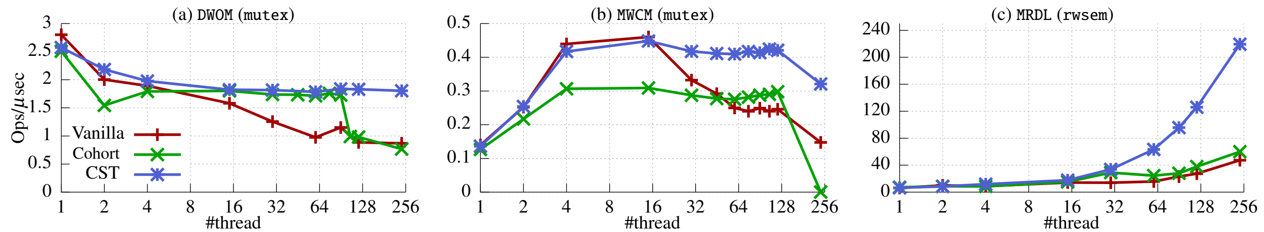

Figure 6: Impact of synchronization primitives on the scalability of micro-benchmarks [27] that affect file system operations such as (a) overwriting a block in a shared file, (b) creating, and (c) enumerating files in a shared directory.

system, where multiple instances of locks are in use. We run FxMark because it stresses various file system operations by only stressing various kernel components that interact with the virtual file system layer, without any user-space computation. We use three micro-benchmarks from FxMark [27] to show how multiple instances—static lock size allocation, contention, and scheduler—affect the scalability of file system operations: `DWOM` updates a shared file in which threads overwrite a block. It represents a log in I/O workloads such as databases that multiple threads share and manipulate. `MWCM` creates multiple files in a shared directory. Both stress the writer lock of `rwsem`, and `mutex`. Finally, `MRDM` enumerates all files in a shared directory and stresses the reader lock.

Block overwrite. Figure 6 (a) shows the impact of various locks on the scalability of block overwriting that stresses the `mutex`. We observe that the CST lock outperforms the Cohort lock by 1.6× and 2.3×, and the Linux one by 2.6× and 2.5× for 120 and 240 threads, respectively. Its efficient parking design maintains an on-par performance even in the over-subscribed scenario (i.e., 2× more threads). Cohort locks suffer from scheduler interaction because tasks get frequently rescheduled, which consume 54.4% of the time because of no scheduling information. The native `mutex` suffers from cache-line bouncing until 60 cores, but starts to suffer from scheduler intervention since the threads start parking themselves as the system is 98% and 90% idle at 120 and 240 threads, respectively.

File creation. Figure 6 (b) shows the impact of various locks on file creation. `CST-mutex` outperforms the Cohort lock by 1.4× and 1614.7×, and the Linux `mutex` by 1.7× and 2.2× for 120 and 240 threads, respectively. At 240 cores, `CST-mutex` suffers from a bottleneck imposed on the memory allocator because of the over-subscription, which also happens with the Linux `mutex`. The Cohort lock, stresses both the scheduler and the memory allocator, as each operation allocates a new inode, whose size is 3.8× larger than the normal inode structure. Moreover, at 240 cores, its performance severely degrades because of its non-blocking nature, and is 743.0× slower than the Linux `mutex`. The Linux version again suffers from the cache-line contention after 30 cores and then from scheduler intervention after 60 cores.

File enumeration. Figure 6 (c) shows the impact of reading a directory. `CST-rwsem` achieves almost linear scalability with increasing threads up to 120 cores and further scales in the over-subscribed case. It outperforms the Cohort lock by 3.3× and 3.7×, and the Linux one by 4.6× and 4.7× for 120 and 240 threads, respectively. The Cohort lock still suffers from scheduler interaction, whereas the Linux version suffers from cache-line contention because of the global count of readers compared with the per-socket storage by both hierarchical locks.

7.3 Performance Breakdown

We evaluate how each component of CST contributes to the overall performance improvement by using an in-kernel hash table that is protected by a single lock. To quantify the impacts of NUMA awareness and parking strategy, we keep the read-write ratio at 90/10%. We vary the thread count from 1 to 600 threads on 120 cores to show the effectiveness of our blocking strategy even in the over-subscribed scenario. Figure 7 (a) shows the throughput of readers with increasing thread count. We evaluate three variants of the reader-side parking strategy: 1) global wake-up of parked readers (CST-Wake) and 2) distributed wake-up (CST-DWake). In an under-subscribed system, CST variants outperform both Cohort and Linux by 4.6× and 10×, respectively, as Cohort locks suffers from scheduler intervention (86.4%) and `mutex` is contending on the global reader count value. Beyond 120 threads, both the Cohort and CST-Spin approaches perform poorly compared with Linux because they are non-blocking. On the other hand, CST-Wake and CST-DWake scale up to 600 threads, thereby showing the importance of blocking behavior. CST-DWake, a distributed wake-up scheme for readers, wakes up more readers in parallel, thereby improving their performance by 1.2× over the global wake-up strategy and outperforming the Linux version by 9.1×.

Figure 7 (b) presents another micro-benchmark results in which we update a single cache line by multiple threads from 120 to 600. We compare the Linux's `mutex` with the Cohort lock and two CST locks: 1) CST-WA is the blocking lock that modifies the status invariant and wakes up all parked waiters in a socket, and 2) CST-WS is also blocking but wakes up the selected number of parked waiters in which the number of wake-ups is equal to the number of hardware threads in a socket. At 120 threads, the native `mutex` suffers from cache-line bouncing and later from

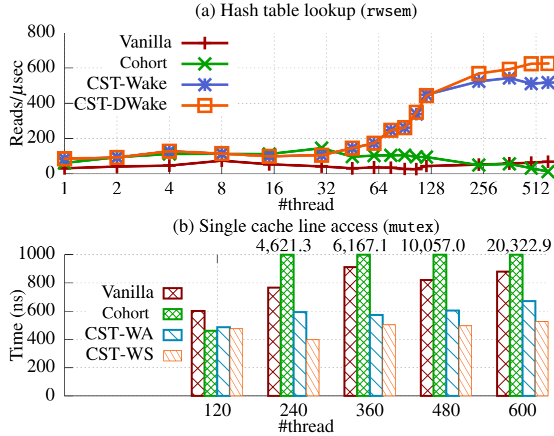

Figure 7: Two micro-benchmarks to illustrate the performance impact of various techniques employed by `CST-rwsem` and `CST-mutex`. Figure (a) represents the lookup performance of a concurrent hash table for 10% writes, which uses `rwsem`. Figure (b) shows the time taken to update a single cache line by holding a `mutex` with increasing thread count.

contention on its global `parking_list` while still maintaining a permissible performance beyond 120 threads. On the other hand, all CST variants address the cache-line bouncing issue for 120 threads. However, the Cohort lock suffers from spinning at higher core count since the waiters preempt the lock-holder after 120 threads. CST-WA and CST-WS address the limitation of the Cohort lock and maintain on-par performance even beyond 120 cores. CST-WS further mitigates the lock-holder preemption problem, since it does not wake up all waiters in one shot inside a socket, which has slightly higher throughput than CST-WA. In summary, CST-WS outperforms the Linux version by 1.7× at 6× over-subscription.

7.4 Critical Section Latency

We evaluate the lock/unlock pair latency of `rwsem` to gauge the effectiveness of CST against the Linux version. Table 1 shows that while NUMA-aware lock is a better fit for multiple readers/writers, it suffers in the low contention scenario because of the costly operation of finding the `snode` for readers and multiple atomic operations to obtain the lock, which we can improve with the hysteresis-based technique [5].

8 Discussion and Limitations

The current design of CST locks can introduce starvation in two cases: 1) re-queueing of the waiters after they are parked, and 2) the writer-preferred version of the `rwsem`. Although, in theory, we can devise a non-blocking algorithm that mitigates the overhead of costly scheduler interaction for the first case, we have not come across such an algorithm in practice and the CST lock is a better alternative than the current `mutex` that also suffers from the same starvation issue. We believe that this can be a plausible future research direction both in the terms of synchronization primitives and lightweight scheduling. For

Latency	RW-lock (ns)		
	Kernel	**Cohort**	**CST**
Reader (1 reader)	30.4	36.4	37.6
Reader (120 readers)	20,062.2	1,973.2	1,925.3
Writer (0 reader)	31.3	140.3	75.0
Writer (119 readers)	28,545.2	11,314.4	4,252.2

Table 1: Empty critical section latency for `rwsem`.

`CST-rwsem`, we choose a writer-preferred version because it batches readers, thereby improving the throughput of the application, which is similar to the design ideology of the Linux `rwsem` [37]. We can address this limitation by exactly adopting the writer-preferred version of the read-write Cohort lock [4].

Even though CST locks outperform both Cohort locks and the Linux `mutex`, we can further scale applications by using combining [13, 32] or the remote-core locking approach [23]. However, the only caveat with these approaches is that we need to rewrite some parts of the OS, which is not easy due to the large code base and complicated lock usage. Another area in which we can improve the performance of CST locks is the latency in low contention (Table 1). We are investigating the use of hardware transactional memory (TSX) to acquire and release the locks in a transaction as in prior work [5]. Although CST locks cannot completely replace all of the locks, they are beneficial to a few data structures that are critical and contend as much as `inode`, `mm`, `dentry`, etc.

9 Conclusion

Synchronization primitives are the basic building blocks of any parallel application, out of which the blocking synchronization primitives are designed to handle both over- and under-subscribed scenarios. We find that the existing primitives have sub-optimal performance for machines with large core count. They suffer either from cache-line contention or scheduler intervention in both scenarios, and are oblivious to the existing NUMA machines. In this work, we present scalable NUMA-aware, memory-efficient blocking primitives that exploit the NUMA hardware topology along with scheduling-aware parking and wake-up strategies. We implement `CST-mutex` and `CST-rwsem`, which provide the same benefit of existing non-blocking NUMA-aware locks in under-subscribed scenario while maintaining similar peak performance in over-subscribed cases. Our code is available here: https://github.com/sslab-gatech/cst-locks.

10 Acknowledgment

We thank the anonymous reviewers and our shepherd, Jean-Pierre Lozi, for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851, ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] 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 Conference on Operating Systems Design and Implementation*, OSDI, 2010.

[2] 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.

[3] D. Bueso and S. Norton. An Overview of Kernel Lock Improvements, 2014. https://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.

[4] 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.

[5] M. Chabbi and J. Mellor-Crummey. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 22:1–22:14, Barcelona, Spain, Mar. 2016.

[6] 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.

[7] G. Chadha, S. Mahlke, and S. Narayanasamy. When Less is More (LIMO):Controlled Parallelism For improved Efficiency. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, 2012.

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

[9] D. Dice. Malthusian Locks. *CoRR*, abs/1511.06035, 2015. URL http://arxiv.org/abs/1511.06035.

[10] 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.

[11] 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.

[12] Facebook. A persistent key-value store for fast storage environments, 2012. http://rocksdb.org/.

[13] 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.

[14] 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.

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

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

[17] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, New York, NY, Mar. 2010.

[18] X. Leroy. The open group base specifications issue 7, 2016. http://pubs.opengroup.org/onlinepubs/9699919799/.

[19] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 219–230, Philadelphia, PA, June 2014.

[20] Y. Liu. aim7 performance regression by commit 5a50508 report from LKP, 2014. https://lkml.org/lkml/2013/1/29/84.

[21] W. Long. qspinlock: Introducing a 4-byte queue spinlock, 2014. https://lwn.net/Articles/582897/.

[22] W. Long. locking/mutex: Enable optimistic spinning of lock waiter, 2016. https://lwn.net/Articles/696952/.

[23] 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.

[24] 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.

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

[26] Microsoft. SQL Server 2014, 2014. http://www.microsoft.com/en-us/server-cloud/products/sql-server/features.aspx.

[27] 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.

[28] I. Molnar. Linux rwsem, 2006. http://www.makelinux.net/ldd3/chp-5-sect-3.

[29] I. Molnar and D. Bueso. Generic Mutex Subsystem, 2016. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

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

[31] *Data Sheet: SPARC M7-16 Server*. Oracle, 2015. http://www.oracle.com/us/products/servers-storage/sparc-m7-16-ds-2687045.pdf.

[32] 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.

[33] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, 2007.

[34] SAP. SAP HANA 2: the transformer, 2015. http://hana.sap.com/abouthana.html.

[35] M. L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 31–40, New York, NY, USA, 2002. ISBN 1-58113-485-1.

[36] 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, Snowbird, Utah, June 2001.

[37] A. Shi. [PATCH] rwsem: steal writing sem for better performance,

2013. https://lkml.org/lkml/2013/2/5/309.

[38] L. Torvalds. Linux Wait Queues, 2005. http://www.tldp.org/LDP/tlk/kernel/kernel.html#wait-queue-struct.

[39] L. Torvalds. The Linux Kernel Archives, 2017. https://www.kernel.org/.

[40] A. Viro. parallel lookups, 2016. https://lwn.net/Articles/684089/.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

[42] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.*, 45(1), Dec. 2012.