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 an operating system's perspective, 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 manner. 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 reader-writer semaphore in the Linux kernel. Our evaluation results show that the new locks can improve the application performance by 1.2–1.6×, and some of the file system operations by as much as 4.7×, in both under- and over-subscribed scenarios. These new locks use 1.5–10× less memory than state-of-the-art NUMA-aware locks on 120-core machine.

1 Introduction

Over the last decade, microprocessor vendors have been pursuing the direction of bigger multi-core and multi-socket machines [19, 33]. For example, a single system can have up to 4096 hardware threads that are organized into sockets, known as NUMA (Non-Uniform Memory Access) domains [33]. They address a key problem of removing the memory access latency bottleneck by directly attaching multiple CPUs to a large chunk of memory (DRAM). Furthermore, these machines have become a norm to further scale applications such as large in-memory databases (Microsoft SQL server [28]) and processing engines [34, 41].

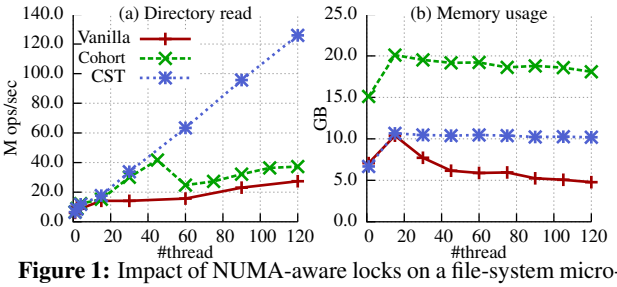

Figure 1: Impact of NUMA-aware locks on a file-system micro-benchmark that spawns threads to enumerate files in a shared directory [29], which stresses the reader side of the reader-writer semaphore (`rwsem`). Figure (a) shows the impact of locks on the throughput till 120 threads on a 120 core machine, where *Vanilla* is Linux's native version, Cohort is an in-kernel ported version of NUMA-aware lock [17], and our NUMA-aware lock implementation (CST). Figure (b) shows memory usage that use these locks before and after the experiments.

A NUMA machine consists of multiple sockets, where each node has a locally attached memory, a last-level cache and multiple CPUs. It only exposes a flat cache-coherent architecture to software by hiding the underlying hardware topology from the applications. Unfortunately, this flat architecture hinders the scalability of applications as the applications may either suffer remote memory access or from the contended memory access from multiple CPUs, thereby degrading their performance [2, 4].

To achieve scalability in NUMA machines, various applications such as databases, and OS rely on NUMA partitioning to mitigate the cost of remote memory access either by data placement or via task placement to achieve high performance. However, this approach does not solve the problem of how to efficiently modify shared data structures such as inodes, dentry cache or even the structures of memory allocator that are shared among multiple threads. As a result, synchronization primitives become the basic building blocks of such multi-threaded applications, and are critical in determining their scalability [4]. Hence, the state-of-the-art NUMA-aware locks [6, 7, 12, 13, 17, 25] are the apt choice to efficiently exploit the NUMA behavior, and also achieve scalability on these architectures. Unfortunately, they are difficult to adopt in practice due to their inherent memory overhead.

For non-blocking locks, Wickizier et al. [4] showed that a Ticket lock suffers from cache-line contention with increasing core count. They replace it with the MCS lock to mitigate such effect, improving the system performance. However, its adoption faced several challenges due to the memory constraint of the spinlock [23]. Similarly, for blocking synchronization primitives, there are various

problems: first, OS developers believe that there is not much contention on these primitives as evidenced by the use of their test-and-set lock or its variant [16, 21, 39]. But this does not hold true for machines with large core count (Figure 1). Secondly, the existing blocking synchronization primitives [13, 36] are NUMA-oblivious, and also suffer from high memory management cost for every lock acquisition. This can severely degrade the performance of the system running on machines with large core count [11, 20]. Thirdly, NUMA-aware locks suffer from memory bloat issue as they statically allocate memory for all sockets for each lock instance, which is a serious concern in an OS [5].

In this work, we design and implement two scalable blocking synchronization primitives, namely CST-`mutex` and CST-`rwsem`, which are variants of `mutex` and a reader-writer semaphore (`rwsem`) in the Linux kernel. 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). To support the blocking behavior, we incorporate the timeout capability for waiters, including readers and writers, in which waiters can park and wake-up without hurting the system's performance. We rely on three key ideas to implement a scalable blocking synchronization primitive. First, we consciously allocate memory by maintaining a dynamic list of per-socket structures that are a basic building block of NUMA-aware locks. Second, instead of passing the lock to a particular waiter, we pass it to a not yet parked waiter (still spinning). This removes the scheduler intervention while passing the lock to a waiter. Last, we keep track of parked waiters in a separate, per-socket list without manipulating the actual queue maintained by the lock protocol. Thus, our blocking primitives improve the application performance by as much 1.2–1.6×, and is 10× faster than existing blocking primitives in over-subscribed scenarios for various micro-benchmarks. Moreover, our approach uses 1.5–10× less memory compared to the state-of-the-art NUMA-aware locks.

In summary, this paper makes three following contributions:

- **Memory-efficient NUMA-awareness.** We maintain a dynamically-allocated list of per-socket structures that address the memory bloat problem.
- **Scheduling-aware parking strategy.** Our approach removes the scheduler interaction by passing the lock to an spinning waiter and batching the wake-up operation.
- **Two scalable blocking synchronizations.** We design and implement two synchronization primitives, namely `CST-mutex` and `CST-rwsem`, that efficiently scale beyond 100 physical cores.

2 Related Work

We classify prior research directions into three categories: NUMA-aware locks, timeout-based locks, and runtime contention management that addresses the over-subscription scenario.

NUMA-aware locks. NUMA-aware locks address the limitation of NUMA-oblivious locks [27], by amortizing the cost of accessing the remote memory. These locks are hierarchical in nature, i.e. they maintain multiple levels of lock [8, 12, 13, 17, 25] in the form of a tree. Cohort locks [13] introduced the cohort principle, which enables combining of any two types of locks (similar or different) to design a hierarchical lock for two-level NUMA machines, and later extended them for the reader-writer locks [6]. Before this, there have been prior works [12, 25] that separately presented the same design principle. HMCS further generalized the cohort locks to design an *N* level hierarchical version of MCS lock to support bigger machines like SGI UV [37] or HP Superdome machine [15] that have more than two levels of NUMA hierarchy. But, none of the introduced locks address the problem of memory utilization as the higher level of locks are pre-allocated. This results in sub-optimal performance if multiple instances of locks are used (Figure 1).

Our design of NUMA-aware locks is memory conscious as we defer the allocation of per-socket locks until required. The global lock is an MCS [27] lock, whereas the per-socket lock is a variant of MCS lock (K42). All prior hierarchical locks do not consider the memory usage as they statically allocate the memory for all sockets [17]. This is a major concern while designing these primitives specifically for an OS. As of this writing, we do not know any of the NUMA-aware reader-writer locks that support blocking readers. Other NUMA oblivious solutions that support blocking readers are Linux's `percpu-rwsem` [32], `rwsem` [32], and recently proposed `prwlock` [22]. Both, `percpu-rwsem` and `prwlock` that address the problem of read mostly operations like RCU [26].

Timeout-based locks. Locks with timeout capability address the problem of tolerating preemption of the threads, aborting transactions in databases or even meeting the deadline in real-time systems by abandoning their attempt to acquire the lock. Scott et al. implemented a timeout based locks [35, 36] that either modify the queue and status maintained by the lock or explicitly allocated memory for each lock acquisition. These locks are inefficient in terms of space complexity as well as do not address the cache line bouncing problem of NUMA machines. Moreover, the memory management will become a critical bottleneck for these locks with increasing core count. Cohort locks [13] also present two timeout capable locks but they implement variant of CLH lock [35], which still suffers from explicit memory management.

CST locks are the family of NUMA-aware blocking synchronization primitives that support timeout capability for both readers and writers. CST maintains the locality awareness like the other non-blocking NUMA-aware locks. Unlike the prior locks [35], our design does not require any explicit memory management during the lock acquisition.

Contention management. The interaction between lock contention and thread scheduling is the determinant of application scalability. Johnson et al. [20] addressed this problem by separating the 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 kind of studies and solutions have been proposed for runtimes [9] and task placement strategies inside the kernel without considering the lock subsystem [42]. In the area of locks, the Malthusian lock [11] is a NUMA-oblivious lock that handles thread over subscription by randomly moving a waiter from an active list to a passive list (concurrency culling), which is inspired from Johnson et al.

CST locks are blocking synchronization primitives that efficiently handle over-subscription, in which waiters independently add themselves to a separate list, which holds the timed-out waiters. This is different from the Malthusian lock design, where the lock holder is responsible for moving the waiters. This severely lengthens the unlock phase, since it has to either wake-up or park the waiters. The use separate parking list is not new as various blocking synchronization primitives [30, 31] use it in the form of wait queues [38], which creates contention on the list in an over-subscribed scenario. To remove the global contention, CST locks maintain a per-NUMA, separate parking list, for both readers and writers. This also removes the remote memory accesses, which we later demonstrate (§7).

3 Challenges and Approaches

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

C1. NUMA awareness. A synchronization primitive should perform well under high contention even in NUMA machines. However, the existing locks used in practice [21, 30, 31] address the cache line contention by using queue-based locks [24] for high contention, but do not address the cache line bouncing—accessing the remote socket—introduced in NUMA machines. The remote access is at least 1.6× slower than the local access within a socket, which can severely degrade the performance.

Approach: To achieve high performance in NUMA machines, hierarchical locks (e.g., Cohort lock) are suitable. They mitigate the cross-socket access (cache line bouncing) by passing the lock within a socket, which relaxes the strict fairness guarantee of queue-based locks for throughput.

C2. Memory-efficient data structures. Unfortunately, current hierarchical locks severely bloat the memory because of their large structure size (e.g., 1600 bytes for the Cohort lock on our eight-socket machine: 64×3×8+64), as they statically allocate memory for all sockets, that may be unused. Memory bloat is a serious concern, as it puts the system under memory pressure. This is even more alarming for synchronization primitives as they statically allocate the memory. For example, the structure of `XFS` inode increases by 4% after adding 16 bytes to the `rwsem` structure. This had a huge impact on the footprint and performance, as there can be millions of inodes cached on a system [10]. Hence, existing hierarchical locks are difficult to adopt in practice because they statically allocate per-NUMA structures (e.g., 3 cache lines for a Cohort lock) at initialization.

Approach: A hierarchical lock should dynamically allocate per-NUMA structure only when it is being used; so that we can avoid the memory bloat problem and reduce the memory pressure on a system.

C3. Effective contention management for both over- and under-subscribed situations. Designing synchronization primitives that equally work 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, when a system is over-loaded, they performs poorly because spinning waiters contend each other. On the other hand, blocking synchronization primitives, such as mutex and read-write semaphore, are designed to work well under an over-loaded system. Instead of spinning, waiting threads sleep until a lock holder wakes one up upon lock release. However, it imposes the overhead of wakeup in every unlock operation, which increases the length of the critical section. Also, frequent sleep and wake-up operations impose additional overhead on scheduler, which will result in a scalability bottleneck. To mitigate this problem, many blocking synchronization primitives [21, 30, 31], including mutex in pthread and Linux kernel, employ a *spin-then-park* strategy: a waiter spins for a while, and then parks itself out. But this approach does not consider system-wide contention; hence, its behavior is sub-optimal when multiple locks are contending. A system-wide load controller is presented by Ryan et al. [20] 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 sleeps or

spins.

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

C4. Scalable parking and wake-up strategy. To implement an efficient blocking synchronization primitive, the most important aspect is how and when to park (schedule out) and wake up waiters with minimal performance overhead. The current approach [30, 31] maintain a global parking list to keep track of parked waiters, and a lock holder wakes one of the parked waiters at unlock operation. However, this design has several drawbacks. Frequent updating of a global parking list becomes a single point of contention (even cache line bouncing) in an over-loaded system. This can lead to severe performance degradation, as the lock holder has to wake up each sleeping waiter during the unlock phase, which adds an extra pressure on the scheduler subsystem for waking up the waiters. This can lead to convoy effect [1], in which all the waiters may go to sleep. 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` handling overhead). According to Amdahl's Law, increased sequential part can cause significant degradation of the scalability.

Approach: Instead of waking up the very next waiter, a lock holder passes the lock to a non-sleeping waiter, if any. This avoids waking up other threads under high contention, and the access of the parking list and scheduler interactions is minimized. Also, we maintains a per-NUMA parking list to remove costly cache line bouncing among NUMA domains to access the parking list.

4 Design Principles

We present two scalable NUMA-aware blocking synchronization primitives, a mutex (`CST-mutex`) and a reader-writer 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 [27] and a per-NUMA local lock is a K42 lock [18] (see Figure 2). The first level localizes the cache line contention within a socket, whereas the second mitigates the cache line bouncing among sockets. To enter a critical section, a thread first acquires the per-NUMA local lock, and then the global lock. During the release phase, it first releases the global lock, and then the local lock. We maintain per-NUMA structure (`snode`), which is dynamically allocated when a thread on that NUMA domain first tries to acquire the lock to avoid the memory bloat problem,

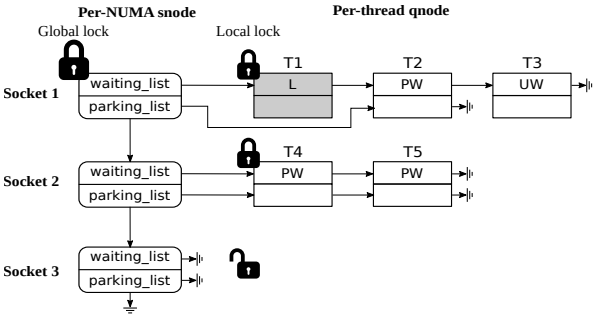

Figure 2: A `CST-mutex` serving for socket 1, 2, and 3. Currently, socket 1 is being served. `T1` now holds the lock (`L`); `T3` is spinning for its turn (`UW`); `T2`, `T4`, and `T5` are sleeping (`PW`) until a lock holder wakes them up. A lock holder, `T1`, will pass the lock to `T3`, which is spinning,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

and which is freed when the lock is destroyed. Each `snode` maintains a per-thread `qnode` in two lists: `waiting_list` is a K42-style list of waiters, and `parking_list` is 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). If its time quantum is over, it will park itself by changing its status to `PW` (parked waiting), and then adds itself to the `parking_list` (`T2`). A lock holder (`T1`) that acquires its local lock and the global lock, passes the lock in the same NUMA domain by traversing the `waiting_list`. It skips the parked waiter (`T2`) and passes the lock to an active waiter (`T3`). If there is no active waiter, a lock holder wakes up parked waiters in the same or other NUMA node to pass the lock. Our reader-writer semaphore additionally maintains a separate list for parked readers, and writers, called a `parking_list`, to handle an over-subscribed system.

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 in designing 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 that it is first accessed. The allocated `snodes` remain until the lock is freed. 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,[1] and 2) when threads are designed to access a subset of sockets—which is common in many applications designed for scalability.

At every lock operation, we first check whether or

[1] The static allocation of all `snodes` increases the `inode` structure size by 3.8× and `mm_struct` size by 2.6×.

not a corresponding snode is already present or not, and then get an snode to acquire the local lock. This process should be efficient and scalable. To determine whether an snode is present, a lock maintains a global bit vector, where each bit denotes whether a corresponding snode is present or not. So a thread check whether an snode is present by simply checking the bit vector. 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. A lock maintains allocated snodes in snode_list so a thread traverses snodes through the snode_list to find the corresponding snode. We separate the snode into two parts— almost-read-only for snode traversal and read-write for local lock operation—to prevent 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 scalability problem in a NUMA architecture; it typically maintains a single, global list (parking_list) to account the sleeping waiters and wake one or some waiters during unlocking to pass the lock. This approach is not scalable as it incurs contention on a single, global list on the NUMA architecture, and is not performant as it passes the lock to the potentially sleeping waiter under over-subscribed conditions.

To address these issues, CST uses two key idea: it maintains a per-NUMA parking_list, which minimizes costly cross-socket cache line bouncing, and it passes the lock to an actively spinning waiter, whose time quota is not over yet, to minimize costly wake-up operations. The skipped sleeping waiters are woken up in bulk when there are no active waiters in the snode or when the global lock is passed to the other snode. By relaxing the strict FIFO guarantee, we can mitigate the convoy effect [1].

4.2.1 Low-contending list management

In CST, each snode maintains the K42-style waiting list, which maintains its tail pointer, qtail. For parked waiters, the snode also maintains a per-socket parking_list to account for the parked waiters. Thus, we can avoid the costly cache line bounding while manipulating per-socket parking_list. For the reader-writer semaphore, we maintain a separate readers and writers parking_list. This design simplifies the list processing in the unlock phase as the lock holder can pass the lock to all the 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 the readers in an over-subscribed scenario.

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 for 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 make the parking decision in a distributed way while considering system load. We discuss three key ideas to address the problem of whom to pass the lock to, when to park itself, and how to make the parking decisions for the 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. This guarantees complete fairness, but this, unfortunately, causes severe performance degradation in an over-subscribed system, as it stresses the scheduler to always issue a call to wake up the parked waiter. To mitigate this issue, we modify this invariant of succeeding lock holder from the next waiter to a nearest *active* waiter, which is still spinning for lock acquisition. Hence, the waiting_list encompasses 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, as 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, in common cases it re-queues itself back at the end of the waiting_list for active spinning again. This approach is effective because we can avoid scheduler intervention under high contention by passing the lock to an active waiter. In addition, a batch wake-up strategy amortizes the cost of the wake-up phase.

Scheduling-aware spinning. The current design of hierarchical locks [6, 8, 13] does not consider the amount of time a waiter should spin before parking itself out. Thus, in an over-loaded system, waiting threads will contend each other, hindering the forward progress of the system. Instead, in CST, waiting threads park themselves as soon as their time quota is about to finish. Checking the time quota of a task is trivial in Linux kernel using need_resched(). 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 not preempting the current lock holder, as it may get more CPU cycles to do some useful task; 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 keeps the scheduling decisions proposed by the scheduler to always be fairer.

Scheduling-aware parking. The current blocking synchronization primitives [30, 31] do not account for the load on the, system so they naively park waiters even

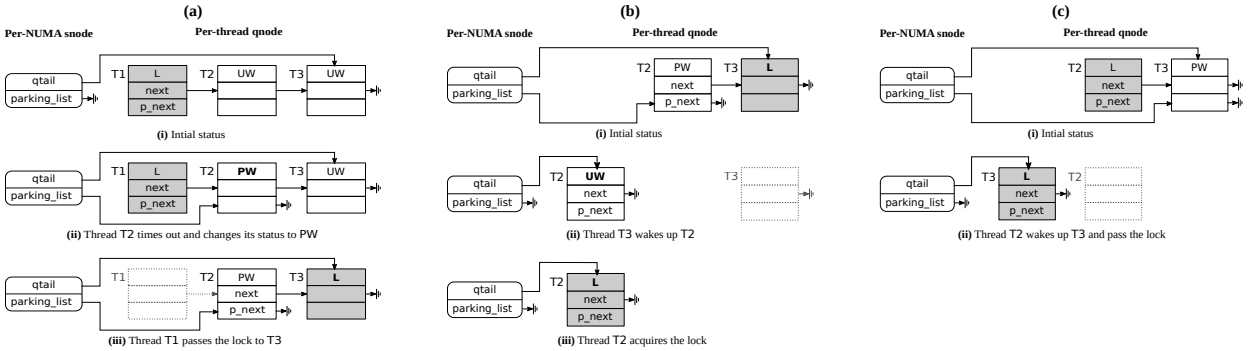

Figure 3: Figure (a) shows the passing of lock to a spinning waiter inside a `snode`. (i) `T1` is the current lock holder, and `T2` and `T3` join `waiting_list` and `qtail` points `qnode` of `T3`. (ii) `T2` times out updates and successfully `CAS`es its state from `UW` to `PW` and adds itself to the `parking_list`. (iii) `T1` will release the lock. 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` and successfully `CAS`es the state of `T3` from `UW` to `L` and then moves out of the unlock phase. Figure (b) shows the waking up of any parked waiters in the `parking_list`. (i) `T3` is in the unlock phase and is last in the `waiting_list`. It successfully `CAS`es `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 waiters, `T2` sets its state to `L` and acquires the local lock and will move on to acquire the global lock. Figure (c) illustrates the passing of lock to a parked waiter at the end of the `waiting_list`. (1) `T2` is going to release the lock, fails to `CAS` the state of `T3` to `L`, since it is parked. (2) `T2` then explicitly `SWAP`s the state of `T3` to `L` and wakes it up. `T3` now holds the local lock and will move on to acquire the global lock.

in the case of an under-loaded system. Hence, a naive use of the spin-then-park approach results in a convoy effect as the waiters will park themselves as soon as their time quota is up, and the lock holder has to do an extra operation of waking them up, thereby severely degrading the performance of the systems for an under-loaded scenario [29]. Also, previous research [20] shows that deciding the system load is critical for the spin-then-park approach, as it not only removes the scheduler interaction from the parking phase, but also improves the latency of the lock/unlock phase.

We decide 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 the per-CPU scheduler in an OS kernel already maintains up-to-date per-CPU active task information. Also, maintaining a system-wide, central information, like the approach used by Johnson et al. [20], is very costly, as the cost of collecting the total number of active tasks increases with increasing core count; the global load does not match the local load, as it cannot catch the load imbalance introduced either because of new incoming tasks or because of rescheduling of the periodic tasks.

5 Scalable Blocking Synchronizations

Based on the aforementioned approaches, we design and implement two different types of queue-based NUMA-aware blocking synchronization primitives: a mutex and a reader-writer semaphore. We first present the design of our `mutex` along with the parking strategy and later extend it to a reader-writer semaphore. The pseudo-code is presented in Figure 4.

5.1 Mutex (`CST-mutex`)

`CST-mutex` is a two-level Cohort lock, which we have extended to be a blocking synchronization primitive 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 [18], 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 any queue-based spinlocks can be used for the local lock. The top level lock maintains a dynamically-allocated socket structure—`snode`—to keep track of the global lock, maintain 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`), and extend the waiting state to 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 proceeds to acquire the global lock. After acquiring the global lock, it becomes 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, and they wait for their respective predecessor to pass the lock. In the release phase, the lock holder passes the lock locally

```
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_GLOBAL_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 on-stack qnode
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 parked or not
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 readeer
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 reder 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/wakeup (lines 75 – 106). We use three atomic instructions: `CAS(addr,old,new)` atomically updates the value at `addr` to `new` and return `True` if the value at `addr` is `old`. Otherwise, it simply 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`.

to its successor, if any. After this, the successor does not acquire the global lock and immediately enters the critical section. Later, a lock holder passes the global lock to a global waiting successor after a bounded number of times to prevent starvation. 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). Contrary to 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 goes to acquire the global lock, when no waiters are present; otherwise, spins on its status, which changes to either `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` goes back on failure; otherwise, it adds itself to the `parking_list` and schedules out. Later, when a lock holder wakes it up, it resumes back (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 the cache line

bouncing, T checks for the global lock flag (line 5). If not set, T already holds the global lock; otherwise, it goes to acquire it.

Acquire global lock: T initializes its snode (line 28) and adds itself to the waiting_list (line 29); then it acquires the global lock if there is no waiter (line 31); otherwise, 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. Even after being woken up, it goes on to acquire the global lock, without re-queueing.

Release local lock: T gets the current snode (line 42) and tries to locally pass the lock based on the batching threshold (line 43). To locally pass the lock, T first tries to CAS the status of its successor from UW to L. If it is successful, 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 the waiters are parked, (line 99), then T releases the global lock (line 49) and then the local one (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 R state to requeue 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, T2 wakes up parked T3 after updating its status from PW to L. 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 one, since T has already released that. If all, including the last waiter, are parked (lines 60–64), T passes the local lock to the last waiter and wakes it up. This is required because T cannot reset the qtail pointer, as there maybe some parked waiters; hence, passing the lock to the last thread is mandatory. Figure 3 (c) illustrates this scenario, where 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 a L state (not R), so that T3 can maintain the K42/MCS lock protocol.

Release global lock: The protocol varies from the MCS protocol while 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 snode leader (lines 72 – 74).

5.2 Reader-writer Semaphore (CST-rwsem)

Our reader-writer semaphore is a writer-preferred version of the Cohort reader-writer lock [6] (CST-rwsem) with two extensions: 1) application of our parking strategy to the readers, and 2) our own version of mutex algorithm (§5.1). CST-rwsem relaxes the condition of acquiring the critical section by multiple threads in *read mode*. Hence, CST-rwsem 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 snode to support the parking of readers by maintaining a separate parking_list for them, so that readers can separately park themselves without intervening with the writers.

Write lock: Thread T first acquires the CST-mutex (line 109). Then it traverses all snodes to check whether the value of active_readers is zero (line 111). Since our algorithm is a writer-preferred one, T blocks other readers from entering the CS, since 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: Thread 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 readers in the parking_list and wakes up the remaining sleeping waiters in its own socket. Lines 143 – 146 illustrate the waking up of the parked reader and subsequent readers.

Write unlock: Thread T first releases the writer lock (line 116). If there are no writers (line 117), then T checks for any sleeping waiters across all snode. If there are any, it only wakes up the very first waiter, which will subsequently wake up the remaining waiters to acquire the read lock (line 119). It provides 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 inhibits the writer, which is going to be the lock holder, from parking itself out.

6 Implementation

We have implemented our locks on Linux Kernel v4.6 and v4.7. Since our interface requires dynamic memory management, we further provide a destructor API to reclaim the snode memory as part of the termination of data structures (e.g., destroy_inode for inode). For our evaluation, we modify the inode structure to our CST-rwsem

in v4.7 and `CST-mutex` in v4.6 since inode `mutex` was replaced to inode `rwsem` at v4.7 [40]. We also modify the virtual memory subsystem of the Linux kernel: `mmap_sem` and their replacement required changes to 650 calls to `mmap_sem` and five calls for the inode. In total, our lock implementations comprises 1,100 lines of code (LoC) and can easily replace the Linux's `mutex`

7 Evaluation

We evaluate CST by answering the following questions:

- What is the impact of locks on real-world applications in terms of performance and memory utilization? (§7.1)
- What is the impact of locks on the kernel-provided operations in various scenarios? (§7.2)
- How does each design aspect help in improving the performance? (§7.3, §7.4)

Evaluation setup. We use three workloads from MOSBENCH [3] that scale well with increasing core count and use the memory subsystem intensively: Histogram, Metis, and Psearchy. They represent different usage of the VM subsystem, where the ratios between write (memory mapping) and read (page-fault) operations are small, medium and large. Furthermore, to evaluate the performance of handling the over-subscribed situations, we choose two microbenchmarks from FxMark [29] that are dependent on the inode structure: they stress various file system operations such as writing in a shared file (`DWOM`) and enumerating files (`MRDM`). Last, we break down the performance implications of our each design aspect by using a simple hash table microbenchmark. We perform all of our experiments on an 8-socket, 120-core machine with Intel Xeon E7-8870 v2 processors.

7.1 Application Benchmarks

We evaluate the performance and scalability implications of `CST-rwsem` on various applications, when we adopt it in the memory subsystem in Linux. For comparison, we also evaluate the current implementation of `rwsem` in the Linux kernel, as well as a Cohort lock implementation [17] ported in the kernel. For each benchmark results, we use *Vanilla* for the native Linux's `rwsem`, *Cohort* for a Cohort reader-writer lock, and *CST* for our `CST-rwsem` implementation.

Histogram. This is a MapReduce application, which is page-fault intensive. It mmaps an 11GB file at the beginning and keeps reading this file while each thread performs a simple computation. As a result, the NUMA-aware Cohort and CST locks outperform the native implementation beyond more than 60 cores and maintain similar performance trends up to 120 cores, as shown in Figure 5. The main reason for showing better performance in both locks is that they localize the number of active readers for `mmap_sem` within a socket, thus spending less time for lock contention: both locks only show 2% idle time as the Cohort lock is non-blocking by design and CST effectively behaves as a non-blocking lock. On the other hand, the vanilla version shows 10.5% of idle time as its ineffective parking strategy kicks in under an under-subscribed situation. In summary, both locks outperform the native `rwsem` by 1.2× at 120 cores.

Metis. This workload runs one worker thread per core and mmaps 12GB of anonymous memory for generating tables for map, reduce, and merge phases, so its performance is bounded by the page-fault operations. Figure 5 (b) shows that both Cohort and CST locks outperform the original version by 1.6× as soon as the frequency of 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 system. On the other hand, the original version has 39% of idle time. This is due to its naive parking strategy, which keeps on parking the readers and writers.

Psearchy. This workload is a parallel version of searchy that does text indexing, and is mmap-intensive, which stresses the memory subsystem from multiple userspace threads. It does around 100,000 small and large mmap/munmap operations, which taxes the writer side of the `rwsem`. Figure 5 (c) shows that both, Cohort and CST, locks maintain a almost similar performance and outperform the original `rwsem` by 1.4×. They spend only around 11.4% of idle time, whereas the original one shows around 53.4% of idle time. Hence, it pays the cost of waking up waiters, whereas our approach mitigates the scheduler intervention by efficiently spinning on the waiters. Even though CST has an equivalent time spent in the kernel (30%), this time is mostly spent by the waiters spinning on their own status.

Summary. Figure 5 shows that as the contention between the readers and writers starts increasing, the original `rwsem` idle time increases. This happens because most of the waiters park themselves as they run out of their time quota. The original `rwsem` suffers from cache line contention since it only employs a spin-based approach, severely degrading the performance, even though it incorporates an optimization [24]. With our efficient spinning strategy that checks its local load, CST locks have the same benefit as the Cohort locks in case of a highly-contended but under-subscribed system.

In general, both Cohort and CST outperform the Linux native one in terms of performance. However, Cohort locks use the same memory with each core count as it statically allocates the memory. On the other hand, CST efficiently allocates memory (Figure 5) which only increases with increasing socket count. Hence, CST saves

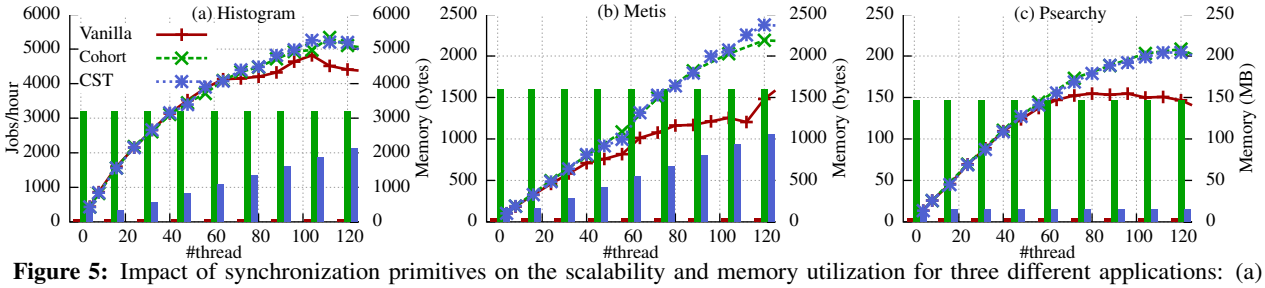

Figure 5: Impact of synchronization primitives on the scalability and memory utilization for three different applications: (a) Histogram, (b) Metis, and (c) Psearchy—with Linux's native reader-writer semaphore (Vanilla), Cohort reader-writer lock, and our `CST-rwsem`.

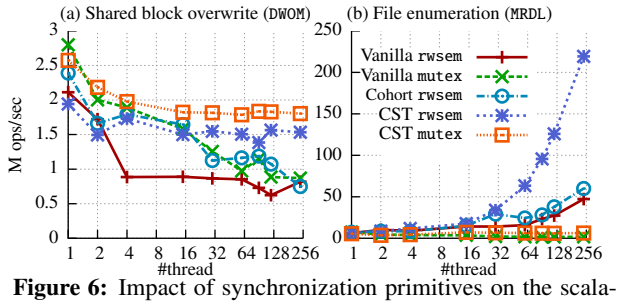

Figure 6: Impact of synchronization primitives on the scalability of the two microbenchmarks of FxMark [29] that affect the file system operations such as (a) overwriting a block in a shared file, and (b) enumerating files in a shared directory.

up to 10× of memory for each workload when running on a single socket, and up to 1.5 – 9.1× at 120 cores.

7.2 Over- And Under-subscribed Cases

We compare the performance of CST locks with the kernel and the Cohort locks, in both a over- and under-subscribed system, where multiple instances of locks are in use. We chose to run FxMark because it stresses the file-system operations by only stressing various kernel components that interact with the virtual file-system layer, without having any actual user-space computation. We use two micro-benchmarks from FxMark [29] to show how multiple instances, static lock size allocation, contention, and convoy effect affect the scalability of file-system operations. The first one overwrites a block in which threads overwrite a block (`DWOM`). This is representative of I/O workloads in databases, where a log is maintained and shared by multiple processes/threads. It stresses the write part of `rwsem` and `mutex`. The other one is read-intensive and enumerates all the files in a shared directory (`MRDM`). It stresses the readers' throughput. However, the earlier Linux versions relied on `mutex` (till v4.6), which serialized all the readers [40].

Block overwrite. In this micro-benchmark, each thread opens a shared file and overwrites its own block for a specified duration (30 seconds). Hence, this workload stresses the writer part of the lock as a thread acquires the write lock to update the inode information. Figure 6 (a) shows the impact of various locks on the performance of block overwriting. `mutex` performs better than any of the `rwsem` algorithms, since they are built on top of `mutex` and pay the processing cost to distinguish between readers and a writer (e.g. `CST-mutex` is 20% faster than `CST-rwsem`). Furthermore, the efficient parking design maintains the performance even in the over-subscribed scenario (i.e., 2× more threads) for the CST locks. For the Cohort locks, scheduler interaction becomes the bottleneck as the tasks get frequently rescheduled, which takes up 54.4% of the time since it is a non-blocking primitive. The Linux versions (both `mutex` and `rwsem`) suffer from cache line bouncing till 60 cores, but starts to suffer from scheduler intervention since the threads start parking themselves. This increases the idle time to 98% for 120 threads and 90% for 240 threads. Hence, CST locks outperform Cohort locks by 1.6× and 2.3×, and Linux one by 2.6× and 2.5× for 120 and 240 threads, respectively.

File enumeration. This is a read-dominated micro-benchmark that creates a specified number of threads that enumerate files in a shared directory. Figure 6 (b) shows the impact of various locks. CST-`rwsem` achieves almost linear scalability with increasing threads till 120 cores and further maintains its performance in an over-subscribed case. Our `rwsem` outperforms Cohort 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 the scheduler interaction, whereas the Linux version suffers from cache-line contention, as it maintains a global count of the readers compared to the per-socket storage by both of the hierarchical locks. The mutex is not the right choice since it serializes the parallel lookup. Even then, both Cohort and CST locks outperform the Linux one by 3.7× for 120 threads, but CST outperforms the Cohort lock by 1.4×.

7.3 Performance Breakdown

We evaluate how each component of CST contributes to the overall performance improvement by using a hash table that is protected by a single lock and is running in Linux kernel. To quantify the impacts of NUMA awareness and parking strategy, we keep the read and write ratio at 10% in this benchmark. We vary the thread count from 1–600 threads on 120 cores to show the effectiveness of our blocking lock even in the over-subscribed scenario. Figure 7 (a) shows the readers' throughput with increasing thread count. We evaluate three variants of the reader-

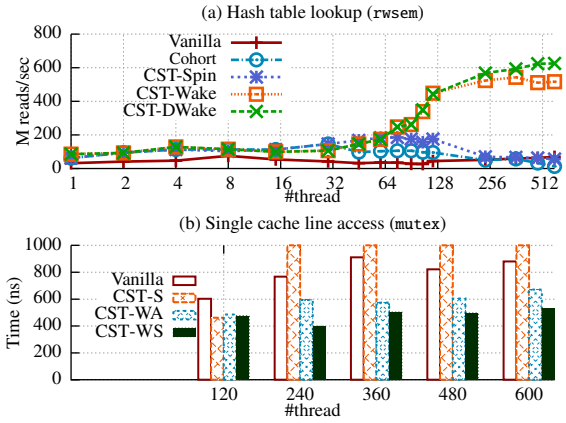

Figure 7: Two micro-benchmarks to illustrate the performance impact of various techniques employed by `CST-rwsem` and `CST-mutex`. Figure (a) represents the lookup performance of a concurrent hash table for 10% writes, which uses `rwsem`. Figure (b) shows the time taken to update a single cache line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

side parking strategy: 1) no reader parking (CST-Spin), 2) global wakeup of parked readers (CST-Wake), and 3) distributed wakeup (CST-DWake). For an under-subscribed system, CST variants outperform both Cohort and Linux by 4.6× and 10×, respectively, as Cohort 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 to Linux, as they are non-blocking, thereby loosing their effectiveness of NUMA-awareness. On the contrary, CST-Wake and CST-DWake scale up to 600 threads, thereby showing the importance of blocking behavior. CST-DWake, which represents a distributed wakeup scheme for the readers, wakes up more readers in parallel, thereby improving the readers' performance by 1.2× over the global wakeup strategy and outperforming Linux by 9.1×.

Another micro-benchmark shown in Figure 7 (b) is updating a single cache line by multiple threads, which vary from 120 to 600. We use Linux's `mutex` and compare it with three variants: 1) simple `mutex` that does not modify the status invariant of queue locks (CST-S), 2) one that modifies the invariant but wakes up all the parked waiters (CST-WA), and 3) one that wakes up a selected number of waiters (CST-WS). For 120 threads, the Linux `mutex` suffers from cache-line bouncing and later suffers from the 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, CST-S suffers from spinning at higher core count since the waiters waste CPU cycles, thereby preempting the lock-holder after 120 threads. On the other hand, CST-WA and CST-WS maintain the performance even at increasing core count. CST-WA further solves the thundering-herd problem and lock-holder preemption problem, since it does not wake up all the waiters in one shot. As a result, CST-WS outperforms the Linux version by 1.7×, at 6× oversubscription.

Latency	RW-lock	
	Kernel (ns)	CST (ns)
Reader (1 reader)	28.67	37.57 (0.7×)
Reader (120 readers)	22,105.86	1,925.31 (11.5×)
Writer (0 reader)	29.12	74.98 (0.4×)
Writer (119 readers)	44,523.12	4,252.21 (10.5×)

Table 1: Empty critical section latency for reader-writer semaphores

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 NUMA-aware locks are a better fit in the case of multiple readers/writers, whereas it suffers in the case of low contention since it has to pay the cost of finding the `snode` (for readers) and has to perform multiple atomic operations to get the lock. We can improve the latency at low contention by employing a Hysteresis-based technique [7].

8 Discussion and Limitations

NUMA-aware locks show better performance at high contention whereas they show lower performance at low contention compared to non-NUMA-aware alternatives (e.g, test-and-test-and-set lock), as they incur the cost of extra atomic operations. We are investigating to use hardware transactional memory (TSX) to acquire and release the locks in a transaction like prior work [7]. The current designs of queue-based locks do not address the problems in nested-level locking, where a thread acquires various lock objects. This is common in large code bases with have multiple subsystems which interact among themselves such as Linux [14]. Hence, the introduction of queue-based locks can degrade the performance and this will exacerbate in hierarchical locks.

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 the convoy effect in both scenarios, and are oblivious to the existing NUMA machines. In this work, we present scalable NUMA-aware, memory-efficient blocking primitives, which exploit the NUMA hardware topology along with scheduling-aware parking and wakeup 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.

References

[1] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, Apr. 1979.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] H. Dickins. [PATCH] mm lock ordering summary, 2004. http://lkml.iu.edu/hypermail/linux/kernel/0406.3/0564.html.

[15] H. P. Enterprise. HPE Integrity Superdome X, 2016. https://www.hpe.com/h20195/v2/GetPDF.aspx/c04383189.pdf.

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

[17] 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.

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

[19] *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.

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *Ottawa Linux Symposium*, OLS, 2002.

[27] 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.

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

[29] 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.

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

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

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

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

[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] SGI. SGI UV, The World's Most Powerful In-Memory Supercomputers, 2017. https://www.sgi.com/products/servers/uv/.

[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.