Avoiding Scheduler Subversion using Scheduler–Cooperative Locks

Yuvraj Patel, Leon Yang*, Leo Arulraj+,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison

Abstract

We introduce the *scheduler subversion* problem, where lock usage patterns determine which thread runs, thereby subverting CPU scheduling goals. To mitigate this problem, we introduce *Scheduler-Cooperative Locks (SCLs)*, a new family of locking primitives that controls lock usage and thus aligns with system-wide scheduling goals; our initial work focuses on proportional share schedulers. Unlike existing locks, SCLs provide an equal (or proportional) time window called *lock opportunity* within which each thread can acquire the lock. We design and implement three different scheduler-cooperative locks that work well with proportional-share schedulers: a user-level mutex lock (u-SCL), a reader-writer lock (RW-SCL), and a simplified kernel implementation (k-SCL). We demonstrate the effectiveness of SCLs in two user-space applications (UpScaleDB and KyotoCabinet) and the Linux kernel. In all three cases, regardless of lock usage patterns, SCLs ensure that each thread receives proportional lock allocations that match those of the CPU scheduler. Using microbenchmarks, we show that SCLs are efficient and achieve high performance with minimal overhead under extreme workloads.

ACM Reference Format:
Yuvraj Patel, Leon Yang*, Leo Arulraj+, and Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. 2020. Avoiding Scheduler Subversion using Scheduler–Cooperative Locks. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3342195.3387521

* - Now at FaceBook, + - Now at Cohesity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387521

1 Introduction

In the modern shared datacenter, scheduling of resources is of central importance [30, 41, 54, 60]. Applications and services, whether within a single organization or from many competing ones, each place strong demands on compute, memory, network, and storage resources; as such, scalable and effective scheduling is required to ensure that no single user or application receives more than its desired share.

Of particular importance in datacenters is CPU scheduling, as recent studies have shown [39]. An ideal CPU scheduler should be able to decide what runs on a given CPU at any given moment in time. In doing so, the CPU scheduler can optimize for fair allocation [5, 57], low latency [51], performance isolation [4, 55], or other relevant goals.

However, the widespread presence of concurrent shared infrastructure, such as found in an operating system, hypervisor, or server, can greatly inhibit a scheduler and prevent it from reaching its goals. Such infrastructure is commonly built with classic locks [13, 59, 61], and thus is prone to a new problem we introduce called *scheduler subversion*. When subversion arises, instead of the CPU scheduler determining the proportion of the processor each competing entity obtains, the lock usage pattern dictates the share.

As a simple example, consider two processes, P_0 and P_1, where each thread spends a large fraction of time competing for lock L. If P_0 generally holds L for twice as long as P_1, P_0 will receive twice as much CPU time as compared to P_1, even if L is a "fair" lock such as a ticket lock [43]. Even though the ticket lock ensures a fair acquisition order, the goals of the scheduler, which may have been to give each thread an equal share of CPU, will have been subverted due to the lock usage imbalance. Similarly, for an interactive process where scheduling latency is crucial, waiting to acquire a lock being held repetitively by a batch process can subvert the scheduling goal of ensuring low latency.

To remedy this problem, and to build a lock that aligns with scheduling goals instead of subverting them, we introduce the concept of *usage fairness*. Usage fairness guarantees that each competing entity receives a time window in which it can use the lock (perhaps once, or perhaps many times); we call this *lock opportunity*. By preventing other threads from entering the lock during that time, lock opportunity ensures that no one thread can dominate CPU time.

To study usage fairness, we first study how existing locks can lead to the scheduler subversion problem. We then propose how lock usage fairness can be guaranteed by *Scheduler-Cooperative Locks (SCLs)*, a new approach to lock construction. SCLs utilize three important techniques to achieve their goals: tracking lock usage, penalizing threads that use locks excessively, and guaranteeing exclusive lock access with lock slices. By carefully choosing the size of the lock slice, either high throughput or low latency can be achieved depending on scheduling goals.

We implement three different types of SCLs; our work focuses on locks that cooperate with proportional-share schedulers. The user-space Scheduler-Cooperative Lock (u-SCL) is a replacement for a standard mutex and the Reader-Writer Scheduler-Cooperative Lock (RW-SCL) implements a reader-writer lock; these are both user-space locks and can be readily deployed in concurrent applications or servers where scheduler subversion exists. The kernel Scheduler-Cooperative Lock (k-SCL) is designed for usage within an OS kernel. Our implementations include novel mechanisms to lower CPU utilization under load, including a *next-thread prefetch mechanism* that ensures fast transition from the current-lock holder to the next waiting thread while keeping most waiting threads sleeping.

Using microbenchmarks, we show that in a variety of synthetic lock usage scenarios, SCLs achieve the desired behavior of allocating CPU resources proportionally. We also study the overheads of SCLs, showing that they are low. We investigate SCLs at small scale (a few CPUs) and at larger scale (32 CPUs), showing that behavior actually improves under higher load as compared to other traditional locks.

We also demonstrate the real-world utility of SCLs in three distinct use cases. Experiments with UpScaleDB [31] show that SCLs can significantly improve the performance of find and insert operations while guaranteeing usage fairness. Similarly, experiments with KyotoCabinet [35] show that unlike existing reader-writer locks, RW-SCL provides readers and writers a proportional lock allocation, thereby avoiding reader or writer starvation. Lastly, we show how k-SCL can be used in the Linux kernel by focusing upon the global file-system rename lock, *s_vfs_rename_mutex*. This lock, under certain workloads, can be held for hundreds of seconds by a single process, thus starving other file-system-intensive processes; we show how k-SCL can be used to mitigate this lock usage imbalance and thus avoid scheduler subversion.

The rest of this paper is organized as follows. We first discuss the scheduler subversion problem in Section 2 and show in Section 3 that existing locks do not ensure lock usage fairness. Then we discuss the design and implementation of SCLs in Section 4 and evaluate the different implementations of SCLs in Section 5. We present limitations and applicability of SCLs in Section 6, related work in Section 7, and our conclusions in Section 8.

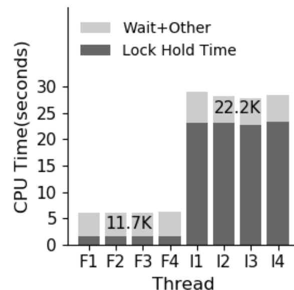


Figure 1. Scheduler subversion with UpScaleDB. *We use a modified built-in benchmarking tool ups_bench to run our experiments. Each thread executes either find or insert operations and runs for 120 seconds. All threads are pinned on four CPUs and have default thread priority. "F" denotes find threads while "I" denotes insert threads. "Hold" represents the critical section execution, i.e., the time the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The value presented on the top of the dark bar is the throughput (operations/second).*

2 Scheduler Subversion

In this section, we discuss scheduler goals and expectations in a shared environment. We describe how locks can lead to scheduler subversion in an existing application.

2.1 Motivation

Scheduling control is desired for shared services and complex multi-threaded applications. Classic schedulers, such as the multi-level feedback queue [3, 23], use numerous heuristics to achieve performance goals such as interactivity and good batch throughput [51]. Other schedulers, ranging from lottery scheduling [57] to Linux CFS [5], aim for proportional sharing, thereby allowing some processes to use more CPU than others.

Unfortunately, current systems are not able to provide the desired control over scheduling. For example, Figure 1 shows that UpScaleDB, an embedded key-value database [31], is not able to correctly give a proportional share of resources to find and insert operations. The workload is similar to an analytical workload where new data is continuously updated while being analyzed. For this experiment, all the threads performing insert and find operations are given equal importance. We set the thread priority of each thread to the default and thus the CFS scheduler will allocate CPU equally to each thread. Although the desired CPU allocation for each thread is the same, the graph shows that insert threads are allocated significantly more CPU than the find threads (nearly six times more) leading to subversion of the scheduling goals.

The breakdown of each bar in Figure 1 also shows that the global pthread mutex lock used by UpScaleDB to protect the environment state is held significantly longer by insert threads than find threads, and the majority of CPU time is spent executing critical sections.

Applications	Operation Type	LHT Distributions (microseconds)					Notes
		Min	25%	50%	90%	99%	
memcached (Hashtable)	Get	0.02	0.05	0.11	0.16	0.29	Get and Put operations are executed having 10M entries in cache using the memaslap tool.
	Put	0.02	0.04	0.10	0.14	0.28	
leveldb (LSM trees)	Get	0.01	0.01	0.01	0.01	0.02	2 threads issuing Get and 2 threads issuing Put operations are executed on an empty database using the db_bench tool.
	Write	0.01	0.11	0.44	4132.7	43899.9	
UpScaleDB (B+ tree)	Find	0.01	0.01	0.03	0.24	0.66	1 thread issuing Find and 1 thread issuing Insert operations are executed on an empty database.
	Insert	0.36	0.87	1.11	4.37	9.55	
MongoDB (Journal)	Write-1K	230.3	266.7	296	497.2	627.5	Write operations are executed on an empty collection. Write concern w = 0, j = 1 used. Size of operations is 1K, 10K and 100K respectively.
	Write-10K	381.6	508.2	561.6	632.8	670.3	
	Write-100K	674.2	849.3	867.8	902.4	938.9	
Linux Kernel (Rename)	Rename-empty	1	2	2	3	3	First rename is executed on an empty directory while the second rename is executed on directory having 1M empty files; each filename is 36 characters long.
	Rename-1M	10126	10154	10370	10403	10462	
Linux Kernel (Hashtable)	Insert	0.03	0.04	0.04	0.05	0.13	Design similar to the futex infrastructure in the Linux kernel that allows duplicate entries to be inserted and delete operation deletes all the duplicate entries in the hashtable.
	Delete	0.06	1.15	1.61	9.27	18.07	

Table 1. Lock hold time (LHT) distribution. *The table shows LHT distribution of various operations for various applications that use different data-structures.*

2.2 Non-Preemptive Locks

The scheduler subversion we saw in UpScaleDB is largely due to how UpScaleDB uses locks. Locks are a key component in the design of multi-threaded applications, ensuring correctness when accessing shared data structures. With a traditional non-preemptive lock, the thread that holds the lock is guaranteed access to read or modify the shared data structure; any other thread that wants to acquire the lock must wait until the lock is released. When multiple threads are waiting, the order they are granted the lock varies depending on the lock type; for example, test-and-set locks do not guarantee any particular acquisition order, whereas ticket locks [43] ensure threads take turns acquiring the given lock.

For our work, we assume the presence of such non-preemptive locks to guard critical data structures. While great progress has been made in wait-free data structures [29] and other high-performance alternatives [16], classic locks are still commonly used in concurrent systems, including the operating system itself.

2.3 Causes of Scheduler Subversion

There are two reasons why scheduling subversion occurs for applications that access locks. First, scheduler subversion may occur when critical sections are of significantly different lengths: when different threads acquire a lock, they may hold the lock for varied amounts of time. We call this property *different critical-section lengths*.

The property of different critical-section lengths holds in many common use cases. For example, imagine two threads concurrently searching from a sorted linked list. If the workload of one thread is biased towards the front of the list, while the other thread reads from the entire list, the second thread will hold the lock longer, even though the operation is identical. Similarly, the cost of an insert into a concurrent data structure is often higher than the cost of a read; thus, one thread performing inserts will spend more time inside the critical section than another thread performing reads.

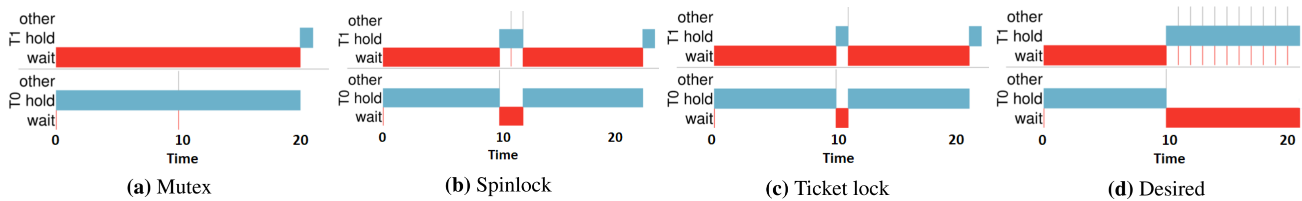
The lengths of critical sections do vary in real-world applications, as shown in Table 1. We note two important points. First, the same operation within an application can have significant performance variation depending on the size of the operation (e.g., the size of writes in MongoDB) or the amount of internal state (e.g., the size of a directory for renames inside the Linux kernel). Second, the critical section times vary for different types of operations within an application. For example, in leveldb, write operations have a significantly longer critical section than find or get operations.

Second, scheduler subversion can occur when time spent in critical sections is high and there is significant contention for locks. We call this *majority locked run time*. There are many instances in research literature detailing the high amount of time spent in critical sections; for example, there are highly contended locks in Key-Value stores like UpScaleDB (90%) [25], KyotoCabinet [19, 25], LevelDB [19], Memcached (45%) [25, 37]; in databases like MySQL [25], Berkeley-DB (55%) [37]; in Object stores like Ceph (53%) [47]; and in file systems [46] and the Linux kernel [6, 58].

Unfortunately, different critical section lengths combined with spending a significant amount of time in critical sections directly subverts scheduling goals. When most time is spent holding a lock, CPU usage is determined by lock ownership rather than by scheduling policy: the algorithm within the lock to pick the next owner determines CPU usage instead of the scheduler. Similarly, when locks are held for different amounts of time, the thread that dwells longest in a critical section becomes the dominant user of the CPU.

For the UpScaleDB experiment discussed earlier, we see that these two properties hold. First, insert operations hold the global lock for a longer period of time than the find operations, as shown in Table 1. Second, the insert and find threads spend around 80% of their time in critical sections. Thus, under these conditions, the goals of the Linux scheduler are subverted and it is not able to allocate each thread a fair share of the CPU.

Summary: Locks can subvert scheduling goals depending on the workload and how locks are accessed. Previous lock implementations have been able to ignore the problem of scheduler subversion because they have focused on optimizing locks within a single application across cooperating



(a) Mutex **(b)** Spinlock **(c)** Ticket lock **(d)** Desired

Figure 2. Impact of Critical Section Size. *The behavior of existing locks when the critical section sizes of two threads differ are shown. The CFS scheduler is used and each thread is pinned on a separate CPU. "wait" represents the time spent waiting to acquire the lock; "hold" represents the critical section execution, i.e., the time the lock is held; "other" represents the non-critical section execution.*

threads; in this cooperative environment, the burden of using locks correctly can be placed on the application developer, since any scheduling subversion hurts only the threads within that application. However, when locks are accessed by different clients in a competitive environment (e.g., within a shared service, the kernel, or a virtual machine), important locks may be unintentionally or even maliciously held for significantly different amounts of time by different clients; thus, the entire system may exhibit fairness and starvation problems. In these cases, lock design must include mechanisms to handle the competitive nature of their usage.

3 Lock Opportunity

In this section, we describe how existing locks do not guarantee lock usage fairness and introduce *lock opportunity* – a new metric to measure lock usage fairness.

3.1 Inability to control CPU allocation

Existing locks, such as mutex, spin-locks, and even ticket locks, do not enable schedulers to control the CPU allocation given to different threads or processes. We illustrate this with an extremely simple application that has two threads and a single critical section. For one of the threads (T0), the critical section is very long (10 seconds), while for the other (T1), it is very short (1 second); for both, the non-critical section time is negligible (0). We consider three common lock implementations: a pthread mutex, a simple spinlock that uses the test-and-set instruction and busy-waits, and a ticket lock that uses the *fetch-and-add* instruction and busy-waits. The application is run for 20 seconds.

As shown in Figure 2, even though the scheduler is configured to give equal shares of the CPU to each thread, all three existing lock implementations enable the thread with the long critical section (T0) to obtain much more CPU.

In the case of the mutex (Figure 2a), T0 dominates the lock and starves T1. This behavior arises because the waiter (T1) sleeps until the lock is released. After T1 is awoken, but before getting a chance to acquire the lock, the current lock holder (T0) reacquires the mutex due to its short non-critical section time. Thus, with a mutex, one thread can dominate lock usage and hence CPU allocation.

The behavior of the spinlock (Figure 2b) is similar, where the next lock holder is decided by the cache coherence protocol. If the current lock holder releases and reacquires the lock quickly (with a negligible non-critical section time), it can readily dominate the lock. With spinlocks, since the waiting thread busy-waits, CPU time is spent waiting without making forward progress; thus, CPU utilization will be much higher than with a mutex.

Finally, the ticket lock suffers from similar problems, even though it ensures acquisition fairness. The ticket lock (Figure 2c) ensures acquisition fairness by alternating which thread acquires the lock, but T0 still dominates lock usage due to its much longer critical section. Lock acquisition fairness guarantees that no thread can access a lock more than once while other threads wait; however, with varying critical section sizes, lock acquisition fairness alone cannot guarantee lock usage fairness.

This simple example shows the inability of existing locks to control the CPU allocation. One thread can dominate lock usage such that it can control CPU allocation. Additionally, if an interactive thread needs to acquire the lock, lock usage domination can defeat the purpose of achieving low latency goals as the thread will have to wait to acquire the lock. Thus, a new type of locking primitive is required, where lock usage (not just acquisition) determines when a thread can acquire a lock. The key concept of *lock opportunity* is our next focus.

3.2 Lock Opportunity

The non-preemptive nature of locks makes it difficult for schedulers to allocate resources when each thread may hold a lock for a different amount of time. If instead each thread were given a proportional "opportunity" to acquire each lock, then resources could be proportionally allocated across threads.

Lock opportunity is defined as the amount of time a thread holds a lock or *could* acquire the lock, because the lock is available. Intuitively, when a lock is held, no other thread can acquire the lock, and thus no thread has the opportunity to acquire the lock; however, when a lock is idle, any thread has the opportunity to acquire the lock. The *lock opportunity time* (LOT) for Thread i is formally defined as:

$$LOT(i) = \sum Critical_Section(i) + \sum Lock_Idle_Time \quad (1)$$

For the toy example above, Table 2 shows the lock opportunity time of threads T0 and T1. We see that thread T1 has a much lower lock opportunity time than thread T0; specifically, T1 does not have the opportunity to acquire the lock while it is held by thread T0. Therefore, thread T1's LOT is small, reflecting this unfairness. Using lock opportunity time for

	Mutex	Spinlock	Ticketlock	Desired
LOT Thread 0	20	20	20	10
LOT Thread 1	1	3	2	10
Fairness Index	0.54	0.64	0.59	1

Table 2. Lock Opportunity and Fairness. *The table shows lock opportunity and the Jain fairness index for the toy example across the range of different existing locks, as well as the desired behavior of a lock.*

each thread, we quantify the fairness using Jain's fairness index [32]; the fairness index is bounded between 0 and 1 where a 0 or 1 indicates a completely unfair or fair scenario respectively. As seen in the table, all three existing locks achieve fairness scores between 0.54 and 0.64, indicating that one thread dominates lock usage and thereby CPU allocation as well. Note that even though the ticket lock ensures acquisition fairness, it still has a low fairness index.

Thus, fair share allocation represents an equal *opportunity* to access each lock. For the given toy example, the desired behavior for equal lock opportunity is shown in Figure 2d. Once T0 acquires the lock and holds it for 10 seconds, the lock should prevent thread T0 from acquiring the lock until T1 has accumulated the same lock opportunity time. As T0 is prevented from accessing the lock, T1 has ample opportunity to acquire the lock multiple times and receive a fair allocation. Notice that at the end of 20 seconds both threads have the same lock opportunity time and achieve a perfect fairness index of 1.

Building upon this idea, we introduce Scheduler-Cooperative Locks (SCLs). As we will see, SCLs track lock usage and accordingly adjust lock opportunity time to ensure lock usage fairness.

4 Scheduler-Cooperative Locks

We describe the goals for Scheduler-Cooperative Locks, discuss the design of SCL, and present the implementation of three types of Scheduler-Cooperative Locks: a user-space Scheduler-Cooperative Lock (u-SCL), a kernel version of u-SCL (k-SCL), and a Reader-Writer Scheduler-Cooperative Lock (RW-SCL).

4.1 Goals

Our SCL design is guided by four high-level goals:

Controlled lock usage allocation. SCLs should guarantee a specified amount of lock opportunity to competing entities irrespective of their lock usage patterns. To support various scheduling goals, it should be possible to allocate different amounts of lock opportunity to different entities. Lock opportunity should be allocatable across different types of schedulable entities (e.g., threads, processes, and containers) or within an entity according to the type of work being performed.

High lock-acquisition fairness. Along with lock usage fairness, SCLs should provide lock-acquisition fairness when arbitrating across threads with equal lock opportunity. This secondary criterion will help reduce wait-times and avoid starvation among active threads.

Minimum overhead and scalable performance. SCLs must track the lock usage pattern of all threads that interact with a given lock, which could be costly in time and space. SCLs should minimize this overhead to provide high performance, especially with an increasing number of threads.

Easy to port to existing applications. For SCLs to be widely used, incorporating SCLs into existing applications (including the OS) should be straightforward.

In this work, our primary focus is on proportional allocation. In the future, lock cooperation with other types of schedulers will be an interesting avenue of work.

4.2 Design

To ensure lock usage fairness without compromising performance, the design of SCL is comprised of three components.

Lock usage accounting. Each lock must track its usage across each entity that the scheduler would like to control. Accounting and classification can be performed across a wide range of entities [4]. For example, classification can be performed at a per-thread, per-process, or per-container level, or according to the type of work being performed (e.g., reading or writing). For simplicity in our discussion, we often assume that accounting is performed at the thread granularity. We believe that different types of locks can be built depending on the classification. Similarly, each thread (or schedulable entity) has a goal, or allotted, amount of lock opportunity time; this goal amount can be set to match a proportional share of the total time as desired by the CPU scheduler; for example, a default allocation would give each thread an equal fair share of lock opportunity, but any ratio can be configured.

Penalizing threads depending on lock usage. SCLs force threads that have used their lock usage quota to sleep when they prematurely try to reacquire a lock. Penalizing these threads allows other threads to acquire the lock and thus ensures appropriate lock opportunity. The potential penalty is calculated whenever a thread releases a lock and is imposed whenever a thread attempts to acquire the lock. The penalty is only imposed when a thread has reached its allotted lock usage ratio. Threads with a lower lock usage ratio than the allotted ratio are not penalized.

Dedicated lock opportunity using lock slice. Accounting for lock usage adds to lock overhead, especially for small critical sections (lasting nanoseconds or microseconds). To avoid excessive locking overhead, we introduce the idea of a *lock slice*, building on the idea of Lock Cohorts [22]. A lock slice is similar to a time slice (or quantum) used for CPU scheduling. A lock slice is the window of time where a single thread is allowed to acquire or release the lock as often as it would like. Once the lock slice expires, ownership is transferred to the next waiting thread. Thus, a lock slice guarantees lock opportunity to the thread owner; once lock ownership changes, lock opportunity changes as well. Lock slices mitigate the cost of frequent lock owner transfers and

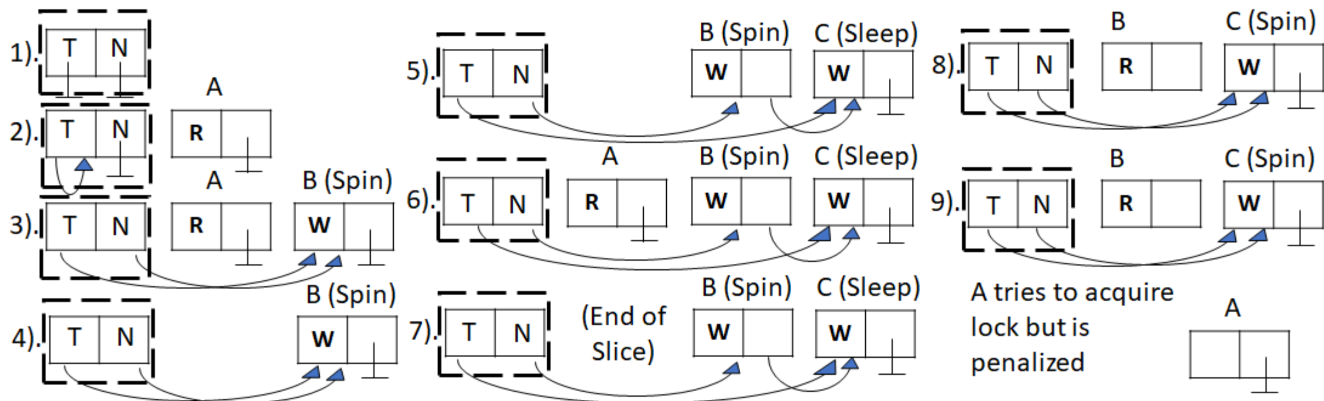


Figure 3. User-space Scheduler-Cooperative Locks. *"R" indicates running and the lock is owned. "W" indicates waiting for the slice. The lock is shown as a dashed box and each lock acquisition request is shown as a node (box). In (1) the lock is initialized and free. (2) A single thread A has acquired the lock. The tail "T" pointer points to itself and the next "N" pointer points NULL. (3) Thread B arrives to acquire the lock and is queued. As B is the next-in-line to acquire the lock, it spins instead of parking itself. The tail and the next pointers now point to B. (4) Thread A releases the lock but as the lock slice has not expired, B will wait for its turn. (5) Thread C also arrives to acquire the lock and is queued after B. C parks itself as it is not the next-in-line to acquire the lock. The tail now points to the C as it is the last one to request lock access. (6) Thread A again acquires the lock as it is the owner of the lock slice. (7) Thread A releases the lock. (8) A's lock slice is over and B is now the owner of the slice. C is woken up and made to spin as it will acquire the lock next. The tail and the next pointers now point to C. (9) A again tries to acquire the lock but is penalized and therefore will wait for the penalty period to be over before it can be queued.*

related accounting. Thus, lock slices enable usage fairness even for fine-grained acquisition patterns.

One can design many types of SCL. We now discuss the implementation of three types of SCL: u-SCL, k-SCL, and RW-SCL. While u-SCL and k-SCL guarantee lock usage fairness at a per-thread level, RW-SCL classifies threads based on the work they do (i.e., readers vs. writers).

4.3 u-SCL Implementation

The implementation of u-SCL is in C and is an extension of the K42 variant of the MCS lock [2]. Threads holding or waiting for the lock are chained together in a queue guaranteeing lock acquisition. Like the K42 variant, the u-SCL lock structure also uses two pointers: tail and next. The tail pointer points to the last waiting thread while the next pointer refers to the first waiting thread, if and when there is one.

For lock accounting in u-SCL, we classify each thread as a separate class and track lock usage at a per-thread level. Per-thread tracking does incur additional memory for every active u-SCL lock. u-SCL maintains information such as lock usage, weight (which is used to determine lock usage proportion), and penalty duration, using a per-thread structure allocated through the pthread library. The first time a thread acquires a lock, the data structure for that thread is allocated using a key associated with the lock. u-SCL does not assume a static set of threads; any number of threads can participate and can have varied lock usage patterns.

To achieve proportional allocations that match those of the CPU scheduler, u-SCL tracks the total weight of all threads and updates this information whenever a new thread requests access to a lock or the thread exits. u-SCL identifies the nice value of the new thread and converts it to weights using the same logic that the CFS scheduler uses. The mapped weights are then added to the total weight to reflect the new proportions.

Consider two threads, T0 and T1, with nice values of 0 and -3. Based on these nice values, the CFS scheduler sets the CPU allocation ratio to approximately 1:2. u-SCL identifies the nice values and converts them to weights using the same logic that CFS scheduler uses (0 maps to 1024 and -3 maps to 1991). The sum of the weights (1024 + 1991) is assigned to the total weight. To calculate each thread's proportion, u-SCL uses each thread's weight and the total weight to calculate the proportion. For T0 and T1, the proportion is calculated as 0.34 and 0.66 respectively, making the lock opportunity ratio approximately 1:2. This way, u-SCL guarantees lock opportunity allocations that match those of the CPU scheduler.

The interfaces *init()* and *destroy()* are used to initialize and destroy a lock respectively. The *acquire()* and *release()* routines are called by the threads to acquire and release a lock respectively. Figure 3 shows the operation of u-SCL. For simplicity, we just show the tail and next pointers to explain the flow in the figure and not other fields of the lock.

The figure begins with lock initialization (Step 1). If a lock is free and no thread owns the slice, a thread that tries to acquire the lock is granted access and is marked as the slice owner (Step 2). Alternatively, if the lock is actively held by a thread, any new thread that tries to acquire the lock will wait for its turn by joining the wait queue (Steps 3 and 5). When a lock is released (Step 4), the lock owner marks the lock as free, calculates its lock usage, and checks if the slice has expired. If the lock slice has not expired, the slice owner

can acquire the lock as many times as needed (Step 6). Since a lock slice guarantees dedicated lock opportunity to a thread, within a lock slice lock acquisition is fast-pathed, significantly reducing lock overhead.

On the other hand, if the lock slice has expired, the current slice owner sets the next waiting thread to be the slice owner (Steps 7 and 8) and checks to see if it has exceeded the lock usage ratio to determine an appropriate penalty. When acquiring the lock, another check is performed to see if the requesting thread should be penalized for overusing the lock. A thread whose lock usage ratio is above the desired ratio is banned until other active threads are given sufficient lock opportunity. The thread is banned by forcing it to sleep (Step 9) and can try to acquire the lock after the penalty is imposed.

We have chosen two milliseconds as the slice duration for all experiments unless otherwise stated. The two millisecond duration optimizes for throughput at the cost of longer tail latency. We will show the impact of slice duration on throughput and latency in Section 5.4.

Optimizations. To make u-SCL efficient, we use the spin-and-park strategy whenever the thread does not immediately acquire the lock; since waiting threads sleep the majority of the time, the CPU time spent while waiting is minimal.

Another optimization we implement is *next-thread prefetch* where the waiting thread that will next acquire the lock is allowed to start spinning (Steps 3 and 8); this mechanism improves performance by enabling a fast switch when the current lock holder is finished with its slice. When this optimization marks the next waiting thread as runnable, the scheduler will allocate the CPU. However, as the thread has not acquired the lock, it will spin wasting the CPU. This behavior will be more visible when the number of threads is greater than the number of cores available.

Limitations. We now discuss a few of the implementation limitations. First, threads that sporadically acquire a lock continue to be counted in the total weight of threads for that lock; hence, other active threads may receive a smaller CPU allocation. This limitation is addressed in our kernel implementation of k-SCL.

Next, our current implementation of u-SCL does not update lock weights when the scheduler changes its goals (e.g., when an administrator changes the nice value of a process); this is not a fundamental limitation. Finally, we have not yet explored u-SCL in multi-lock situations (e.g., with data structures that require hierarchical locking or more complex locking relationships). We anticipate that multiple locks can interfere with the fairness goals of each individual locks leading to performance degradation. The interaction of multiple SCLs remains as future work.

4.4 k-SCL Implementation

k-SCL is the kernel implementation of u-SCL; it is a much simpler version without any of the optimizations discussed above. As larger lock slice sizes will increase the wait-time to own the lock slice, we set the lock slice size to zero while accessing kernel services. The lock functioning is similar to u-SCL. The primary difference is that k-SCL *does* track whether or not threads are actively interacting with a kernel lock and accordingly removes them from the accounting; this is performed by periodically walking through the list of all thread data and freeing inactive entries. Currently, we use a threshold of one second to determine if a thread is inactive or not. A longer threshold can lead to stale accounting for a longer duration, leading to performance issues.

4.5 RW-SCL Implementation

RW-SCL provides the flexibility to assign a lock usage ratio to readers and writers, unlike the existing reader-writer locks that support reader or writer preference. The implementation of RW-SCL is in C and is an extension of the centralized reader-writer lock described by Scott [52]. Being centralized, RW-SCL tracks the number of readers and writers with a single counter; the lowest bit of the counter indicates if a writer is active, while the upper bits indicate the number of readers that are either active or are waiting to acquire the lock. To avoid a performance collapse due to heavy contention on the counter, we borrow the idea of splitting the counter into multiple counters, one per NUMA node [8].

With RW-SCL, threads are classified based on the type of work each executes; the threads that execute read-only operations belong to the reader class while the threads executing write operations belong to the writer class. Since there are only two classes, RW-SCL does not use per-thread storage and track each thread. Fairness guarantees are provided across the set of reader threads and the set of writer threads. As with other SCL locks, different proportional shares of the lock can be given to readers versus writers. For the current implementation, we assume that all the readers will have the same priority. Similarly, all writers will have the same priority. Thus, a single thread cannot assume the role of a reader and writer as the same nice value will be used leading to a 50:50 usage proportion and that might not be the desired proportion.

Figure 4 shows the operation of RW-SCL. The relevant RW-SCL routines include *init()*, *destroy()*, *writer_lock()*, *reader_lock()*, *reader_unlock()*, and *reader_unlock()*. The lock begins in a read slice at initialization (Step 1). During a read slice, the readers that acquire the lock atomically increments the counter by two (Step 2 and 3). On releasing the lock, the counter is atomically decremented by two (Step 4). During the read slice, all the writers that try to acquire the lock must wait for the write slice to be active (Step 5). When readers release the lock, they will check if the read slice has expired and may activate the write slice (Step 6).

While the write slice is active, writers try to acquire the lock by setting the lowest bit of the counter to 1 using the compare-and-swap instruction (Step 7). With multiple writers, only one writer can succeed and other writers must wait for the first writer to release the lock. If a reader tries to acquire

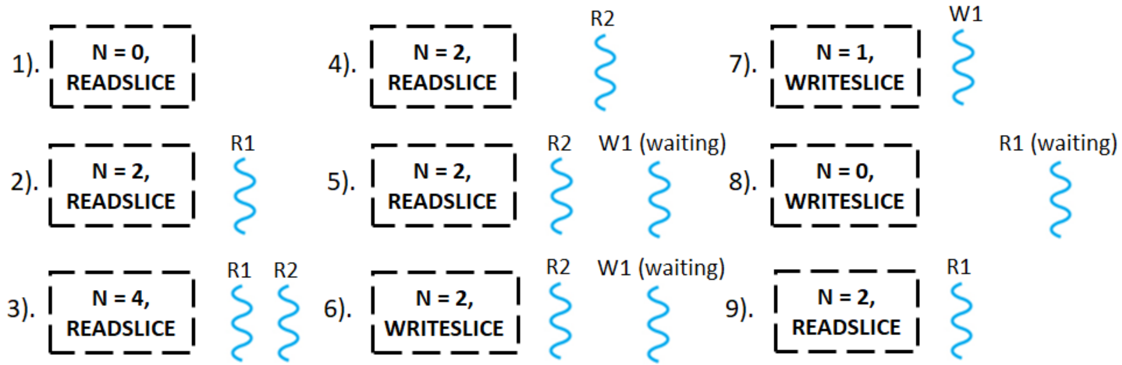


Figure 4. Reader-Writer Scheduler-Cooperative Locks. *In (1) The lock is initialized and free. (2) A reader thread R1 acquires the lock and continues its execution in the critical section. (3) Another reader thread R2 also joins in. (4) Reader R1 leaves the critical section. (5) A writer thread W1 arrives and waits for the write slice to start. (6) Write slice is started and W1 waits for reader R2 to release the lock. (7) Reader R2 releases the lock and writer W1 acquires the lock. (8) Reader R1 now arrives again and waits for the read slice to start. (9) W1 releases the lock and the read slice starts allowing the reader R1 to acquire the lock.*

the lock while the write slice is active, it will wait for the read slice to be active (Step 8). When a writer releases the lock, it will check if the write slice has expired and activate the read slice (Step 9). Whenever a read slice changes to a write slice or vice versa, the readers and writers will wait for the other class of threads to drain before acquiring the lock.

As RW-SCL does not track per-thread lock usage, RW-SCL cannot guarantee writer-writer fairness; however, RW-SCL can improve performance for multiple writers. Within the write class, multiple writers can contend for the lock and thus while one writer is executing the non-critical section, another writer can acquire the lock and execute. This behavior is contrary to the u-SCL behavior where the lock remains unused when the owner is executing the non-critical section code.

5 Evaluation

In this section, we evaluate the effectiveness of SCLs. Using microbenchmarks, we show that u-SCL provides both scalable performance and scheduling control. We also show how SCLs can be used to solve real-world problems by replacing the existing locks in UpScaleDB with u-SCL, the Linux rename lock with k-SCL, and the existing reader-writer lock in KyotoCabinet with RW-SCL.

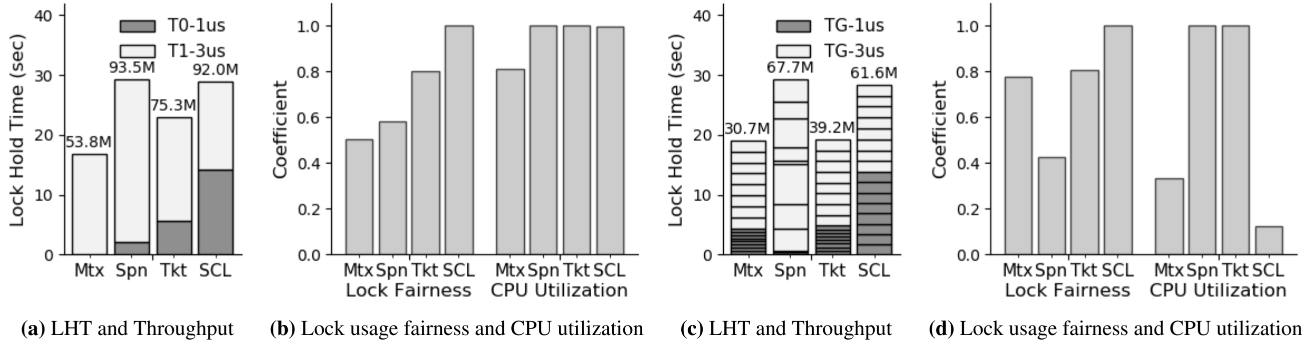
We perform our experiments on a 2.4 GHz Intel Xeon E5-2630 v3. It has two sockets and each socket has eight physical cores with hyper-threading enabled. The machine has 128 GB RAM and one 480 GB SAS SSD. The machine runs Ubuntu 16.04 with kernel version 4.19.80, using the CFS scheduler.

We begin with a synthetic workload to stress different aspects of traditional locks as well as u-SCL. The workload consists of a multi-threaded program; each thread executes a loop and runs for a specified amount of time. Each loop iteration consists of two elements: time spent outside a shared lock, i.e., non-critical section, and time spent with the lock acquired, i.e., critical section. Both are specified as parameters at the start of the program. Unless explicitly specified, the priority of all the threads is the default thereby ensuring each thread gets an equal share of the CPU time according to the CFS default policy.

We use the following metrics to show our results:
(i) Throughput: For synthetic workloads, throughput is the number of times through each loop, whereas for real workloads, it reflects the number of operations completed (e.g., inserts or deletes). This metric shows the bulk efficiency of the approach.
(ii) Lock Hold Time: This metric shows the time spent holding the lock, broken down per thread. This shows whether the lock is being shared fairly.
(iii) Lock Usage Fairness: This metric captures fair lock usage among all threads. We use the method described in Section 3 to calculate lock opportunity time.
(iv) CPU Utilization: This metric captures how much total CPU is utilized by all the threads to execute the workload. CPU utilization ranges from 0 to 1. A higher CPU utilization means that the threads spin to acquire the lock while a lower CPU utilization means the lock may be more efficient due to blocking. Lower CPU utilization is usually the desired goal.

5.1 Fairness and Performance

To gauge the fairness and performance of u-SCL compared to traditional locks, we first run a simple synthetic workload with two threads. For this 30 second workload, the critical section sizes are 1 µs and 3 µs and the two threads are pinned on two different CPUs. In these experiments, the desired result is that for fair scheduling, each thread will hold the lock for the same amount of time, and for performance, they will complete as many iterations as possible.



(a) LHT and Throughput **(b)** Lock usage fairness and CPU utilization **(c)** LHT and Throughput **(d)** Lock usage fairness and CPU utilization

Figure 5. Comparison on 2 and 16 CPUs. *The graphs present a comparison of four locks : mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for 2 (a and b) and 16 (c and d) threads, each has the same thread priority. For 2 threads, each thread has a different critical section size (1 µs vs. 3 µs). For 16 threads, half have shorter critical section sizes (1 µs) while others have a larger critical section size (3 µs). "TG" stands for thread group.*

Figure 5a shows the amount of time each of the two threads holds the lock (dark for the thread with a 1 µs critical section, light for 3 µs); the top of each bar reports throughput. The mutex, spinlock, and ticket lock each do not achieve equal lock hold times. For the mutex, the thread with the longer (3 µs) critical section (light color) is almost always able to grab the lock and then dominate usage. With the spinlock, behavior varies from run to run, but often, as shown, the thread with the longer critical section dominates. Finally, with the ticket lock, threads hold the lock in direct proportion to lock usage times, and thus the (light color) thread with the 3 µs critical section receives three-quarters of the lock hold time. In contrast, u-SCL apportions lock hold time equally to each thread. Thus, u-SCL achieves one of its most important goals. Figure 5b summarizes these results with Jain's fairness metric for lock hold time. As expected, u-SCL's lock usage fairness is 1 while that of other locks is less than 1.

Figure 5a also shows overall throughput (the numbers along the top). As one can see, u-SCL and the spinlock are the highest performing. The mutex is slowest, with only 53.8M iterations; the mutex often blocks the waiting thread using futex calls and the thread switches between user and kernel mode quite often, lowering performance. For CPU utilization, the mutex performs better than others since the mutex lets the waiters sleep by calling *futex_wait()*. The spinlock and ticket lock spin to acquire the lock and thus their utilization is high. u-SCL's high CPU utilization is attributed to the implementation decision of letting the next thread (that is about to get the lock) spin. However, with more threads, u-SCL is extremely efficient, as seen next.

We next show how u-SCL scales, by running the same workload as above with 16 threads on 16 cores. For this experiment, the critical section size for half of the threads (8 total) is 1 µs and other half (also 8) is 3 µs respectively. From the Figure 5c, we see again that the three traditional locks – mutex, spinlock, and ticket lock – are not fair in terms of lock usage; not all threads have the same lock hold times. The threads with larger critical section sizes (lighter color) clearly dominate the threads with shorter critical section sizes (darker). In contrast, u-SCL ensures that all threads receive the same lock opportunity irrespective of critical section size. The throughput with u-SCL is comparable to that of spinlock and we believe that further tuning could reduce this small gap.

However, the more important result is found in Figure 5d, which shows CPU utilization. u-SCL's CPU utilization is reduced significantly compared to other spinning (spinlock and ticket lock) approaches. With u-SCL, out of 16 threads, only two are actively running at any instance, while all other threads are blocked. u-SCL carefully orchestrates which threads are awake and which are sleeping to minimize CPU utilization while also achieving fairness. While a mutex also conserves CPU cycles, it has much higher lock overhead, delivers much lower throughput, and does not achieve fairness.

In summary, we show that u-SCL provides lock opportunity to all threads and minimizes the effect of lock domination by a single thread or a group of threads, thus helping to avoid the scheduler subversion problem. While ensuring the fair-share scheduling goal, u-SCL also delivers high throughput and very low CPU utilization. u-SCL thus nearly combines all the good qualities of the three traditional locks – the performance of spinlock, the acquisition fairness of the ticket lock, and the low CPU utilization of the mutex.

5.2 Proportional Allocation

We next demonstrate that u-SCL enables schedulers to proportionately schedule threads according to a desired ratio other than 50:50. Figure 6 shows the performance of all four locks when the desired CPU time allocation is varied. We now consider four threads pinned to two CPUs, while the other workload parameters remain the same (the critical sections for two threads are 1 µs, for the other two threads they are 3 µs; the workload runs for 30 seconds).

To achieve different CPU proportions for the thread groups, we vary the CFS *nice* values. The leftmost group (3:1) indicates that shorter critical section threads (darker color) should receive three times the CPU of longer critical section threads

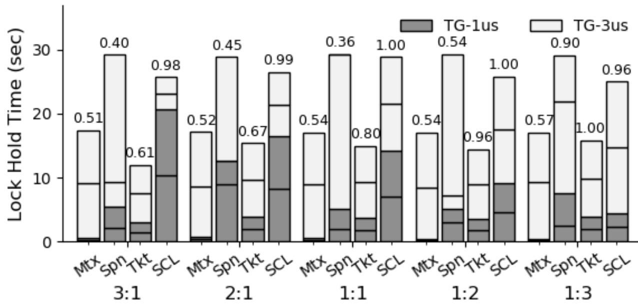


Figure 6. Changing Thread Proportionality. *Comparison of the four locks : mutex (Mtx), spinlock (Spn), ticket lock (Tkt), and u-SCL (SCL) for four threads running on two CPUs having different thread priorities (shown as ratios along the bottom) and different critical section sizes. The number on the top of each bar shows the lock usage fairness.*

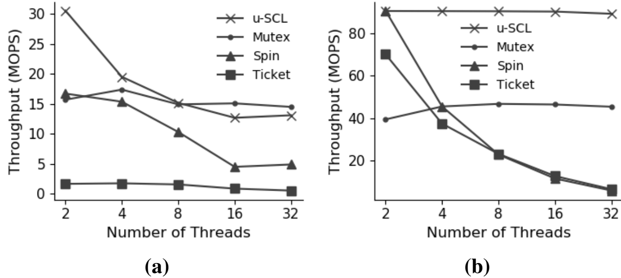


Figure 7. Lock Overhead Study. *The figure presents two lock overhead studies. On the left(a), the number of threads and CPU cores are increased, from 2 to 32, to study scaling properties of u-SCL and related locks. On the right(b), the number of CPUs is fixed at two, but the number of threads is increased from 2 to 32.*

(lighter). The rightmost group shows the inverse ratio, with the shorter critical section threads meant to receive one third the CPU of the longer critical section threads.

Figure 6 shows that traditional locks subvert the target CPU allocations of the CFS scheduler. Having a longer critical section leads to threads holding onto the CPU for a longer duration. Note that the fairness of ticket locks improves whenever the critical section ratio and the thread priority matches.

In contrast, u-SCL performs exactly in alignment with CPU scheduling goals and allocates the lock in the desired proportion to each thread. To do so, u-SCL uses the same weight for the lock usage ratio as the thread's scheduling weight, thereby guaranteeing CPU and lock usage fairness. By configuring u-SCL to align with scheduling goals, the desired proportional-sharing goals are achieved.

5.3 Lock Overhead

Minimal overhead was one of our goals while designing u-SCL. To understand the overhead of u-SCL, we conduct two different experiments as we increase the number of threads with very small critical sections.

For the first experiment, we set each critical section and non-critical section size to 0. We then run the synthetic application varying the number of threads from 2 to 32, with each thread pinned to a CPU core.

Figure 7a (left) compares the throughput of the four lock types. For spinlocks and ticket locks, throughput is generally low and decreases as the number of threads increases; these locks generate a great deal of cache coherence traffic since all threads spin while waiting. In contrast, the mutex and u-SCL block while waiting to acquire a lock and hence perform better. While u-SCL significantly outperforms the other locks for 8 or fewer threads, u-SCL performance does drop for 16 and 32 threads when the threads are running on different NUMA nodes; in this configuration, there is an additional cost to maintain accounting information due to cross-node cache coherency traffic. This indicates that u-SCL could benefit from approximate accounting across cores in future work.

In the second experiment, we show performance when the number of CPUs remains constant at two, but the number of threads increases. For the experiment in Figure 7a (right), we vary the number of threads from 2 to 32 but pin them to only two CPUs. The critical section size is 1 μs.

As expected, the performance of u-SCL and mutex is indeed better than the alternatives and remains almost constant as the number of threads increases. With spinlock and ticket locks, the CPU must schedule all the spin-waiting threads on two CPUs regardless of the fact that threads cannot make forward progress when they are not holding the lock. Moreover, as the threads never yield the CPU until the CPU time slice expires, a great deal of CPU time is wasted.

In contrast, with u-SCL and mutex, only two threads or one thread, respectively, are running, which significantly reduces CPU scheduling. u-SCL performs better than mutex since the next thread to acquire the lock is effectively prefetched; the dedicated lock slice also helps u-SCL achieve higher performance since lock overhead is minimal within a lock slice. With a mutex lock, a waiting thread must often switch between user and kernel mode, lowering performance.

5.4 Lock Slice Sizes vs. Performance

We next show the impact of lock slice size on throughput and latency, as a function of critical section size. In general, increasing lock slice size increases throughput, but harms latency. As we will show, the default two millisecond slice size optimizes for high throughput at the cost of long-tail latency.

Throughput: For our first workload, we run four identical threads pinned to two cores for 30 seconds, varying the size of each critical section. Figure 8a shows throughput in a heatmap. The x-axis varies the lock slice size while the y-axis varies the critical section size. Throughput is calculated by summing the individual throughput of each thread. For larger slice sizes, the throughput increases while for very small slice sizes, the throughput decreases significantly; the overhead

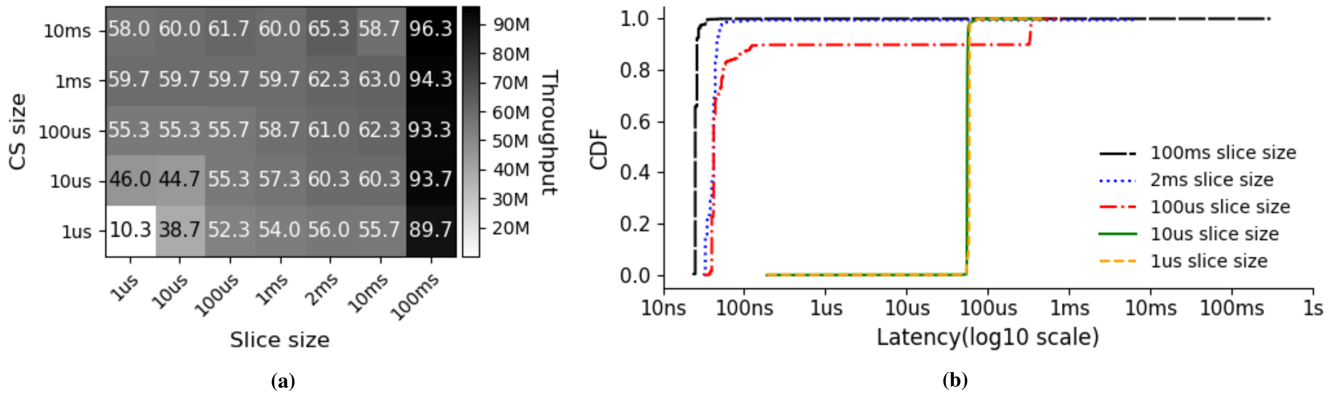


Figure 8. Impact of lock slice size on performance. *The left figure (a) shows the throughput across two dimensions : critical section and the slice size. The right figure (b) shows the wait-time distribution when the lock slice varies and the critical section size is 1 µs.*

of repeatedly acquiring and releasing the lock causes the substantial decrease.

Latency: Figure 8b shows the wait-time distribution to acquire the lock as a function of the slice size for a 10 µs critical section; we chose 10 µs to show the impact of having a slice size smaller, greater, or equal to the critical section size. We omit 1 ms and 10 ms lock slices because those results are similar to that of 2 ms. The figure shows that for lock slices larger than the critical section, the wait-time of the majority of operations is less than 100 ns; each thread enters the critical section numerous times without any contention. However, when the thread does not own the lock slice, it must wait for its turn to enter the critical section; in these cases, the wait time increases to a value that depends on the lock slice size and the number of participating threads.

When the lock size is smaller than or equal to the size of the critical section (i.e., 1 or 10 µs), lock ownership switches between threads after each critical section and each thread observes the same latency. We observe this same relationship when we vary the size of the critical section (not shown).

To summarize, with larger lock slices, throughput increases but a small portion of operations observe high latency, increasing tail latency. On the other hand, with smaller lock slices, latency is relatively low, but throughput decreases tremendously. Hence, applications that are latency sensitive should opt for smaller lock slices while applications that need throughput should opt for larger lock slices.

Interactive jobs: We now show that u-SCL can deliver low latency to interactive threads in the presence of batch threads. Batch threads usually run without user interaction and thus do not require low scheduling latency [5]. On the other hand, interactive threads require low scheduling latency; thus, the scheduler's task is to reduce the wait-time for interactive threads so they can complete tasks quickly. Both Linux's CFS and FreeBSD's ULE schedulers identify interactive and batch threads and schedule them accordingly [51]; for example, as the interactive threads sleep more often without using their entire CPU slice, CFS schedules such threads before others to minimize their latency.

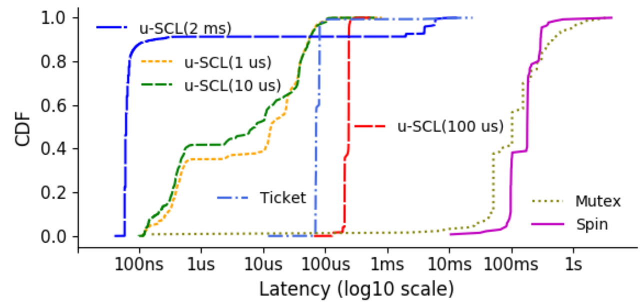
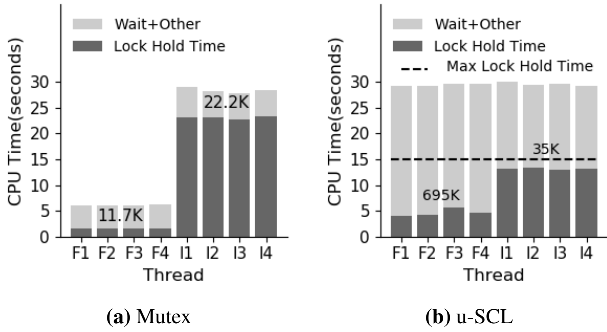


Figure 9. Interactivity vs. Batching. *The figure shows the comparison of the wait-time to acquire the lock for mutex, spinlock, ticket lock and u-SCL.*

To show that u-SCL can effectively handle both batch and interactive threads, we examine a workload with one batch thread and three interactive threads. The batch thread repeatedly acquires the lock, executes a 100 µs critical section, and releases the lock; the three interactive threads execute a 10 µs critical section, release the lock, and then sleep for 100 µs. The four threads are pinned on two CPUs. The desired result is that the interactive threads should not need to wait to acquire the lock.

Figure 9 shows the CDF of the wait-time for one of the interactive threads to acquire each of the four lock types: mutex, spinlock, ticket lock, and u-SCL. For u-SCL, we show four lock slice sizes. The results of the other interactive threads are similar. The graph shows that for the mutex and spinlock, wait-time is very high, usually between 10 ms and 1 second. Even though the goal of the scheduler is to reduce latency by scheduling the interactive thread as soon as it is ready, lock ownership is dominated by the batch thread, which leads to longer latency for the interactive threads. The ticket lock reduces latency since lock ownership alternates across threads; however, wait-time is still high because the interactive thread must always wait for the critical section of the batch thread to complete (100 µs).

The graph shows that for u-SCL, the length of the slice size has a large impact on wait-time. When the slice size is smaller



(a) Mutex **(b)** u-SCL

Figure 10. Mutex and u-SCL performance with UpScaleDB. *The same workload is used as Section 2.1. The same CFS scheduler is used for the experiments. "F" denotes find threads while "I" denotes insert threads. The expected maximum lock hold time is shown using the dashed line. "Hold" represents the critical section execution, i.e., the time until the lock is held; "Wait + Other" represents the wait-times and non-critical section execution. The number on top of the dark bar represents the throughput (operations/second). The left figure (a) shows the same graph as shown in Section 2.1. The right figure (b) shows the performance of u-SCL.*

than or equal to the interactive thread's critical section (e.g., 1 or 10 µs), the interactive thread often has a relatively short wait time and never waits longer than the critical section of the batch thread (100 µs). When the slice size is relatively large (e.g., 2 ms), the interactive thread often acquires the lock with no waiting, but the wait-time distribution has a long tail. Finally, the 100 µs slice performs the worst of the u-SCL variants because the interactive threads sleep after releasing the lock, wasting the majority of the lock slice and not allowing other waiting threads to acquire the lock.

In summary, to deliver low latency, the slice size in u-SCL should always be less than or equal to the smallest critical section size. Our initial results with the ULE scheduler are similar, but a complete analysis remains as future work.

5.5 Real-world Workloads

We conclude our investigation by demonstrating how SCLs can be used to solve real-world scheduling subversion problems. We concentrate on two applications, UpScaleDB and KyotoCabinet, and a shared lock within the Linux kernel. The user-space applications show how SCLs can be used to avoid the scheduler subversion problem within a single process. With the kernel example, we illustrate a competitive environment scenario where multiple applications running as different processes (or containers) can contend for a lock within a kernel, thus leading to cross-process scheduler subversion.

5.5.1 UpScaleDB

As part of the original motivation for u-SCL, we saw in Figure 1 that UpScaleDB was unable to deliver a fair share of the CPU to threads performing different operations. For easy comparison, Figure 10a shows the same graph. We now show that u-SCL easily solves this problem; the existing locks in

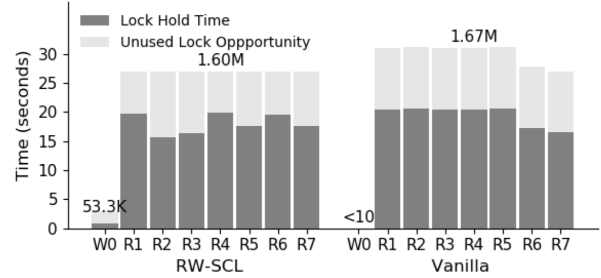


Figure 11. Comparison of RW-SCL and KyotoCabinet *The dark bar shows the lock hold time for each individual thread and the light bar shows the lock opportunity not being unused. The values on top of the bar shows the aggregated throughput (operations/sec) for the writer and reader threads.*

UpScaleDB can simply be converted to u-SCL locks and then the unmodified Linux CFS scheduler can effectively schedule UpScaleDB's threads independent of their locking behavior.

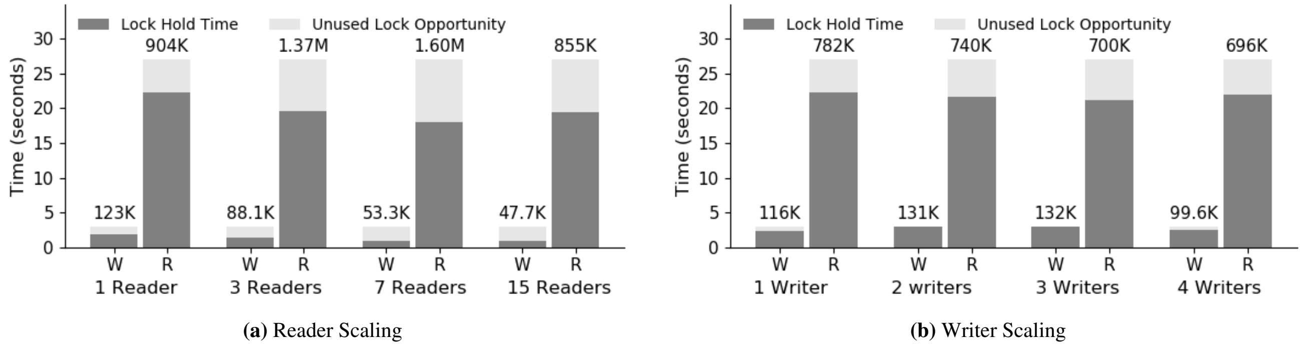
We repeat the experiment shown in Figure 1, but with u-SCL locks. UpScaleDB is configured to run a standard benchmark.[1] We again consider four threads performing find operations and four threads performing inserts, pinned to four CPUs. Figure 10b shows how much CPU time each thread is allocated by the CFS scheduler.

As desired, with u-SCL, the CPU time allocated to each type of thread (i.e., threads performing find or insert operations) corresponds to a fair share of the CPU resources. With u-SCL, all threads, regardless of the operation they perform (or the duration of their critical section), are scheduled for approximately the same amount of time: 30 seconds. In contrast, with mutexes, UpScaleDB allocated approximately only 2 CPU seconds to the find threads and 24 CPU seconds to the insert threads (with four CPUs). With a fair amount of CPU scheduling time, the lock hold times becomes more fair as well; note that it is not expected that each thread will hold the lock for the same amount of time since each thread spends a different amount of time in critical section versus non-critical section code for its 30 seconds.

The graph also shows that the overall throughput of find and insert threads is greatly improved with u-SCL compared to mutexes. On four CPUs, the throughput of find threads increases from only about 12K ops/sec to nearly 700K ops/sec; the throughput of insert threads also increases from 22K ops/sec to 35K ops/sec. The throughput of find operations increases most dramatically because find threads are now provided more CPU time and lock opportunity. Even the throughput of inserts improves since u-SCL provides a dedicated lock slice where a thread can acquire the lock as many times possible; thus, lock overhead is greatly reduced.

Finally, when we sum up the total lock utilization across the u-SCL and mutex versions, we find that the lock is utilized for roughly 59% of the total experiment duration for u-SCL but

[1] ups_bench –use-fsync –distribution=random –keysize-fixed –journal-compression=none –stop-seconds=120 –num-threads=N



(a) Reader Scaling

(b) Writer Scaling

Figure 12. Performance of RW-SCL with reader and writer scaling. *For reader scaling, only one writer is used while for writer scaling, only one reader is used. The number of readers and writers vary for reader scaling and writer scaling experiments. The dark bar shows the lock hold time while the light bar shows the unused lock opportunity. The values on top of the bar shows the throughput of the writers and readers.*

nearly 83% for mutexes. Given that the overall lock utilization decreases with an increase in throughput, we believe that u-SCL can help scale applications. Instead of redesigning the applications to scale by minimizing critical section length, a more scalable lock can be used.

5.5.2 KyotoCabinet

KyotoCabinet [35] is an embedded key-value storage engine that relies on reader-writer locks. Given that locks are held for significant periods of time, and critical sections are of different lengths, it also suffers from scheduler subversion; specifically, writers can be easily starved. We show that our implementation of RW-SCL allows KyotoCabinet and the default CFS Linux scheduler to control the amount of CPU resources given to readers and writers, while still delivering high throughput; specifically, RW-SCL removes the problem of writers being starved.

To setup this workload, we use KyotoCabinet's built-in benchmarking tool kccachetest in wicked mode on an in-memory hash-based database. We modified the tool to let the thread either issue read-only (reader) or write (writer) operations and run the workload for 30 seconds. The database contains ten million entries which are accessed at random. We pin threads to cores for all the experiments. We assign a ratio of 9:1 to the reader and writer threads. The original version of KyotoCabinet uses pthread reader-writer locks.

To begin, we construct a workload with one writer thread and seven reader threads. In Figure 11, we present the write throughput and the aggregated read throughput, the average lock hold time, and the lock opportunity for readers and writer. In the default KyotoCabinet using pthread reader-writer locks which give strict priority to readers, the writer is starved and less than ten write operations are performed over the entire experiment. In other experiments (not shown), we find that the writer starves irrespective of the number of readers.

On the other hand, RW-SCL ensures that the writer thread obtains 10% of the lock opportunity compared to 90% for the readers (since readers can share the reader lock, their lock opportunity time is precisely shared as well). Since the writer thread is presented with more lock opportunity with RW-SCL, the write throughput increases significantly compared to the vanilla version. As expected, RW-SCL read throughput decreases slightly because the reader threads now execute for only 90% of the time and writes lead to many cache invalidations. The overall aggregated throughput of readers and writers for eight threads with RW-SCL is comparable to other reader-writer locks with KyotoCabinet [19].

We run similar experiments by varying the number of readers and show the result in Figure 12a. KyotoCabinet scales well until 8 threads (7 readers + 1 writer). The throughput drops once the number of threads crosses a single NUMA node. We believe that this performance drop is due to the excessive data sharing of KyotoCabinet data structure across the sockets. Another point to note here is that irrespective of the number of readers, RW-SCL continues to stick to the 9:1 ratio that we specified.

When there is only one writer thread with RW-SCL, the writer cannot utilize its entire write slice since the lock is unused when the writer is executing non-critical section code. To show how multiple writers can utilize the write slice effectively, we conduct another experiment with only one reader while varying the number of writers. As seen in Figure 12b, when the number of writers increases from one to two, the lock opportunity time becomes completely used as lock hold time (as desired); when one writer thread is executing the non-critical section code, the other writer thread can acquire the lock, thereby fully utilizing the write slice. Increasing the number of writers past two cannot further increase write lock hold time or therefore improve throughput; continuing to increase the number of writers past three simply increases the amount of cache coherence traffic.

5.5.3 Linux Rename Lock

A cross-directory rename in Linux is a complicated operation that requires holding a global mutex to avoid a deadlock. When accessed by competing entities, such a global lock can lead to performance problems since all threads needing to perform a rename must wait to acquire it. We show that k-SCL can prevent a bully process that holds the global lock for long

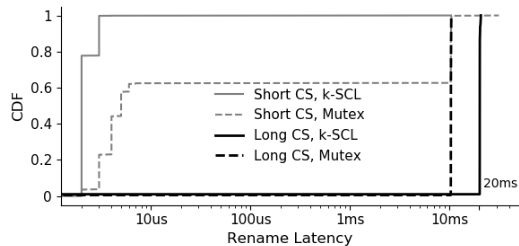
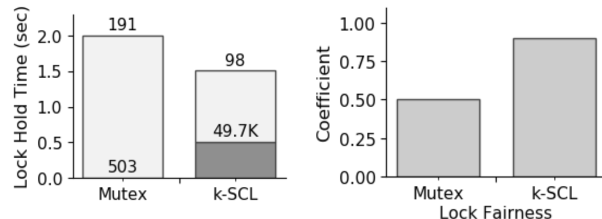


Figure 13. Rename Latency. *The graph shows the latency CDFs for SCL and the mutex lock under the rename operation. The dark lines show the distributions for the long rename operation (the bully), whereas lighter lines represent the short rename operation costs (the victim). Dashed lines show standard mutex performance, whereas solid lines show k-SCL performance.*



(a) LHT and Throughput **(b)** Lock usage fairness

Figure 14. Rename Lock Comparison. *The figure presents a comparison of two locks – mutex and k-SCL for 2 threads on two CPUs; each has the same thread priority and one thread is performing rename operations on a directory that is empty while another thread is performing rename on a directory having a million empty files.*

periods of time from starving out other processes that must also acquire the global lock.

Our specific experiment to recreate a bully and a victim process is as follows. We use Linux version 4.9.128 and the ext4 file system; we have disabled dir_index using tune2fs [9] since that optimization leads to problems [56] which force administrators to disable it. We run a simple program[2] that repeatedly performs cross-directory renames. We create three directories, where one directory contains one million empty files and the other directories are empty; each file name is 36 characters. A bully process executes the rename program with *dst* set to the large directory (potentially holding the rename lock for a long time), while a victim process executes the same program with two empty directories as arguments (thus only needing the lock for a short while).

We use the *ftrace* kernel utility to record cross-directory rename latency, which is plotted in Figure 13 as a CDF. The dimmed and the bold lines represent the victim and bully, respectively. The dotted lines show the behavior with the default Linux locks. The bully has an expected high rename latency of 10 ms. About 60% of victim's rename calls are performed when the bully is not holding the global lock, and thus have a latency less than 10 µs. However, about 40% of the victim's calls have a latency similar to that of the bully due to lock contention. If more bullies are added to the system, the victim has even less chance to acquire the lock and can be starved (not shown).

The root cause of this problem is the lack of lock opportunity fairness. To fix this problem, we modified the global rename lock to use the k-SCL implementation. The solid lines in Figure 13 show the new results. With k-SCL, almost all of the victim's rename calls have less than 10 µs latency and even its worst-case latency is lower, at roughly 10 ms. Both results arise because the bully is banned for about 10 ms after it releases the lock, giving the victim enough lock opportunity to make progress. If more bullies are added, the effect on the victim remains minimal. We believe that this behavior is desired because all tenants on a shared system should have an equal opportunity to utilize the lock.

Figure 14 shows the breakdown of the bully and the victim process's lock behavior. We can see that for the mutex, the bully process dominates lock usage, and the fairness coefficient is very low. k-SCL penalizes the bully process by providing enough opportunity to the victim process to acquire the lock. Therefore, the victim thread is easily able to perform around 49.7K rename operations compared to 503 with the mutex version. As the critical section size of the victim program is very small, the non-critical section involving touch and unlink operation dominates the lock opportunity time presented to the victim. Thus, the victim's lock hold time is not quite equal to that of the bully process.

6 Limitations and Applicability

In this section, we discuss the limitations and applicability of SCLs. We start by discussing limitations and their impact on performance. The current design of SCL is non-work-conserving relative to locks. That is, within a lock slice, whenever the lock slice owner is executing non-critical section code, the lock will be unused, even if other threads are waiting to acquire the lock. Since the lock remains unused, performance could be impacted. Threads that have larger non-critical section sizes compared to critical section sizes are more likely to see this effect. One way to alleviate the performance problem is to reduce the size of the lock slice, but at the cost of throughput as shown earlier.

To design a work-conserving lock, one can assign multiple threads to the same class such that multiple threads can acquire the lock withing a single lock slice. While one thread is executing the non-critical section, another thread can enter the critical section thereby ensuring high performance. However, it is hard to classify multiple threads statically to one class and hence the lock should have the capability to dynamically classify the threads and create classes. Exploring dynamic classification remains an interesting avenue for future work.

To support a variety of scheduling goals, the implementation of SCL needs to be accordingly adjusted. For example, our current implementation of SCL does not support priority

[2]while True: touch(src/file); rename(src/file, dst/file); unlink(dst/file);

scheduling. To support priority scheduling, the lock should also contain queues and grant lock access depending on the priority.

Scheduler subversion happens when the time spent in critical sections is high and locks are held for varying amounts of time by different threads. As these two conditions can lead to lock usage imbalance, the likelihood of the scheduler subversion problem increases. If there is not much lock contention or all threads hold locks for similar amounts of time, then it might be better to use other simpler locks that have less overhead. We believe that shared infrastructure represents a competitive environment where multiple applications can be hosted having varied locking requirements. SCLs will play a vital role in such an environment where one application or user can unintentionally or maliciously control the CPU allocation via lock usage imbalance.

7 Related Work

Locks have been split into five categories based on their approaches [26]: (i) flat, such as pthread mutexes [34], spinlocks [52], ticket locks [43] and many other simple locks [1, 21] (ii) queue [14, 42, 43], (iii) hierarchical [11, 12, 20, 40, 49], (iv) load-control [17, 27], and (v) delegation-based [24, 28, 38, 48]. Our work borrows from much of this existing literature. For example, queue-based approaches are needed to control fairness and hierarchical locks contain certain performance optimizations which are useful on multiprocessors. Although less related, even delegation-based locks could benefit from considering usage fairness, perhaps through a more sophisticated scheduling mechanism of delegated requests.

Many studies have been done to understand and improve the performance and fairness characteristics of locks [10, 15, 25]. However, the authors do not consider lock usage fairness as we propose herein. Guerraoui et al. [25] do acknowledge that interaction between locks and schedulers is important. In this work, we show how locks can subvert the scheduling goals; and SCLs and schedulers can align with each other.

Reader-writer locks have been studied extensively for the past several decades [7, 18, 33, 36, 44, 45] to support fairness, scaling and performance requirements. Brandenburg et al. [7] present a phase fair reader-writer lock that always alternates the read and write phase thereby ensuring that no starvation occurs. Our approach for RW-SCL and the phase fair reader-writer lock do have certain properties in common. Like the phase fair lock, RW-SCL will ensure that read and write slices alternate. RW-SCL is flexible enough to assign lock usage ratio to readers and writers depending on the workload.

The closest problem related to the scheduler subversion problem is the priority inversion problem [53] where a higher priority process is blocked by a lower priority process. The scheduler subversion problem can occur with any priority thread and as we have shown, it can also happen when all the threads have the same priority. We believe that to prevent priority inversion, priority inheritance [53] should be combined with SCL locks.

8 Conclusion

In this paper, we have demonstrated that locks can subvert scheduling goals as lock usage determines which thread is going to acquire the lock next. To remedy this, we introduce Scheduler-Cooperative Locks (SCLs) that track lock usage and can align with the scheduler to achieve system-wide goals. We present three different types of SCLs that showcase their versatility, working in both user-level and kernel environments.

Our evaluation shows that SCLs ensure lock usage fairness even with extreme lock usage patterns and scale well. We also show that SCLs can solve the real-world problem of scheduler subversion imposed by locks within applications. We believe that any type of schedulable entity (e.g., threads, processes, and containers) can be supported by SCLs and look forward to testing this hypothesis in the future. The source code for SCL can be accessed at https://research.cs.wisc.edu/adsl/Software/.

9 Acknowledgments

We thank Steven Hand (our shepherd) and the anonymous reviewers of Eurosys ’20 for their insightful comments and suggestions. We thank David Dice and Alex Kogan for reviewing the initial draft and providing feedback. We thank the members of ADSL for their excellent feedback. We also thank CloudLab [50] for providing a great environment to run our experiments. This material was supported by funding from NSF grants CNS-1421033, CNS-1763810 and CNS-1838733, and DOE grant DE-SC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or any other institutions.

References

[1] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[2] Jonathan Appavoo, Marc Auslander, Maria Butrico, Dilma M Da Silva, Orran Krieger, Mark F Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI ’99, page 45–58, USA, 1999. USENIX Association.

[5] Justinien Bouron, Sébastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In

2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 85–96, 2018.

[6] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[7] Björn B. Brandenburg and James H. Anderson. Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems. *Real-Time Systems*, 46:25–87, 2010.

[8] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 157–166, 2013.

[9] Remy Card. tune2fs(8) - linux man page. https://linux.die.net/man/8/tune2fs, 2020.

[10] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1309–1320. IEEE, 2013.

[11] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 215–226, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Milind Chabbi and John Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F Martinez. Workload Characterization of Interactive Cloud Services on Big and Small Server Platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134. IEEE, 2017.

[14] Travis Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical report, Technical Report TR 93-02-02, Department of Computer Science, University of Washington, 1993.

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, February 2012.

[17] Dave Dice. Malthusian Locks. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 314–327, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Dave Dice and Alex Kogan. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 315–328, USA, 2019. USENIX Association.

[19] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 65–74, New York, NY, USA, 2011. Association for Computing Machinery.

[21] David Dice. Brief Announcement: A Partitioned Ticket Lock. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 309–310, New York, NY, USA, 2011. Association for Computing Machinery.

[22] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[23] D. H. J. Epema. An Analysis of Decay-Usage Scheduling in Multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 23(1):74–85, May 1995.

[24] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[25] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Trans. Comput. Syst.*, 36(1), March 2019.

[26] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 649–662, USA, 2016. USENIX Association.

[27] Bijun He, William N. Scherer, and Michael L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, page 7–18, Berlin, Heidelberg, 2005. Springer-Verlag.

[28] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[29] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[31] UpScaleDB Inc. UpScaleDB. https://upscaledb.com/.

[32] Rajendra K Jain, Dah-Ming W Chiu, and William R Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[33] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, page 201–204, USA, 1993. IEEE Computer Society.

[34] Lawrence Livermore National Laboratory. Mutex variables. https://computing.llnl.gov/tutorials/pthreads, 2017.

[35] FAL Labs. KyotoCabinet. https://fallabs.com/kyotocabinet/.

[36] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable Reader-Writer Locks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[37] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 6, USA, 2012. USENIX Association.

[38] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4), January 2016.

[39] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In *Euro-Par 2006 Parallel Processing*, pages 801–810, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[41] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 144–159, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, page 165–171, USA, 1994. IEEE Computer Society.

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

[44] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. *SIGPLAN Not.*, 26(7):106–113, April 1991.

[45] John M. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, April 1991.

[46] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 71–85, USA, 2016. USENIX Association.

[47] M. Oh, J. Eom, J. Yoon, J. Y. Yun, S. Kim, and H. Y. Yeom. Performance Optimization for All Flash Scale-Out Storage. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 316–325, Sep. 2016.

[48] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, volume 16, page 95. Citeseer, 1999.

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

[50] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[51] Jeff Roberson. ULE: A Modern Scheduler for FreeBSD. In *BSDCon*, 2003.

[52] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[53] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: an Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[54] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 349–362, USA, 2012. USENIX Association.

[55] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. *SIGPLAN Not.*, 33(11):181–192, October 1998.

[56] Axel Wagner. ext4: Mysterious "No space left on device"-errors. https://blog.merovius.de/2013/10/20/ext4-mysterious-no-space-left-on.html, 2018.

[57] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, page 1–es, USA, 1994. USENIX Association.

[58] David Wentzlaff and Anant Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[59] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 3–14, New York, NY, USA, 2010. Association for Computing Machinery.

[60] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The Datacenter Needs an Operating System. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, page 17, USA, 2011. USENIX Association.

[61] A. Zhong, H. Jin, S. Wu, X. Shi, and W. Gen. Optimizing Xen Hypervisor by Using Lock-Aware Scheduling. In *2012 Second International Conference on Cloud and Green Computing*, pages 31–38, Nov 2012.