Locks as a Resource: Fairly Scheduling Lock Occupation with CFL

Jonggyu Park
University of Washington
USA
jonggyu@cs.washington.edu

Young Ik Eom
Dept. of Electrical and Computer Engineering /College of Computing and Informatics, Sungkyunkwan University
South Korea
yieom@skku.edu

Abstract

In multi-container environments, applications oftentimes experience unexpected performance fluctuations due to undesirable interference among applications. Synchronization such as locks has been targeted as one of the reasons but still remains an uncontrolled resource while a large set of locks are still shared across applications. In this paper, we demonstrate that this lack of lock scheduling incurs significant real-world problems including performance unfairness and interference among applications. To address this problem, we propose a new synchronization design with an embedded scheduling capability, called CFL (Completely Fair Locking). CFL fairly distributes a fair amount of lock occupation time to applications considering their priorities and cgroup information. For scalability, CFL also considers the NUMA topology in the case of NUMA machines. Experimental results demonstrate that CFL significantly improves performance fairness while achieving comparable or sometimes even superior performance to state-of-the-art locks.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

synchronization primitives, lock, scalability, resource sharing

ACM Reference Format:

Jonggyu Park and Young Ik Eom. 2024. Locks as a Resource: Fairly Scheduling Lock Occupation with CFL. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Problem statement Modern data centers co-locate multiple applications in a single physical machine to overcome the idleness of over-provisioned resources [7, 13, 17, 19][18]. In this circumstance, since each application requires a discrete level of performance, the underlying system should provide a fair amount of system resources according to their priorities. However, fairness in practice is not simply equalizing the number of resource occupations, but guaranteeing the proportional sharing of system resources according to the importance of applications (e.g., a thread with two times higher priority should occupy twice the amount of CPU than the others). To achieve this, operating systems embody an individual resource scheduler for each system resource, such as CPU and I/O, along with cgroup.

However, the presence of synchronization can dilute such efforts and incur various fairness problems. To prevent data inconsistency, locks serialize concurrent accesses to shared data by allowing only a single thread to enter its critical section (CS), which is the region between lock acquisition and release. Meanwhile, other threads across CPU cores should either sleep (mutex) or busy-wait (spinlock). Unfortunately, unlike hardware resources, locks do not embody a dedicated scheduler for fairly distributing lock occupation. This lack of lock scheduling can incur various real-world problems including performance interference/degradation as demonstrated in[8]. For example, the Linux kernel still utilizes various spinlocks and mutexes for protecting system-wide data structures including inode/directory cache, because the counterparts, such as lock-free data structures, increase the programming complexity and overheads in some cases. In this situation, a batch application can suspend the execution of a latency-sensitive application by holding the shared locks, resulting in significant performance interference, even if they are isolated by cgroup and containers. Similarly, the existence of locks can undermine some performance optimizations to boost the performance of user-interactive tasks using thread priorities because such priorities are not applied to lock acquisition.

Limitations of the state of the art The conventional locks [3, 5, 9, 10, 14] oftentimes tend to equalize the number of lock acquisitions by adhering to either random or pseudo-FIFO policy, while ignoring various characteristics of lock usage, such as the CS lengths and the frequencies of lock acquisition requests. To overcome the aforementioned problem, [12] proposed a scheduler-cooperative mutex, called SCL. SCL monitors lock usage and forcibly suspends certain threads if they hinder CPU time fairness. However, SCL has several drawbacks including non-workconserving nature, architecture-obliviousness, lack of cgroup support, etc. More recently, Syncord implements a spinlock version of SCL with NUMA-awareness as one of its use cases. However, it inherits all the drawbacks from SCL except for NUMA-awareness, thereby being effective only for limited situations.

Our solution Designing a fair lock scheduler is nontrivial since various components including thread priority, occupation frequencies, and hardware characteristics should be carefully considered for the sake of both fairness and performance. Fortunately, the notion of resource scheduling has been widely explored for several

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

decades, and there are various time-tested fair scheduling mechanisms/algorithms such as CFS (Completely Fair Scheduler) , BFQ (Budget Fair Queueing) , etc. For example, CFS enhances fairness and responsiveness by taking into account various factors, including priority and accumulated CPU runtime. With lessons learned from the previous principles, we design a new lock design with an embedded scheduling ability, called CFL (Completely Fair Locking). The goal of CFL is to view locks as an independent resource and fairly distribute lock occupation in a time-tested manner as with other Linux schedulers. To realize this, CFL keeps track of the virtual lock hold time (vLHT) of each thread in a memory-efficient way and modifies the order of lock acquisition so that low vLHT threads can acquire the lock sooner than others. Additionally, CFL considers the underlying hardware architecture such as NUMA to improve scalability. Finally, CFL integrates with cgroup so that it can schedule lock resources based on process groups such as containers.

To verify the efficacy of CFL, we implement spinlock and mutex versions of CFL on userspace and inside the Linux kernel. The evaluation with the set of CFL demonstrates that the design and implementation of CFL significantly improve performance fairness while achieving high performance, compared with conventional locks. In particular, when running a performance-critical application in a consolidated environment, CFL shows 2.4 – 7.1× higher throughput and 54.7 – 84.8% lower average latency, compared with state-of-the-art locks, through providing performance isolation.

Key results and contributions

- We design and implement a set of new locks, called CFL, which embody a fair resource scheduler by leveraging the lessons learned from the previous scheduling principles (§3).
- We present various optimization techniques, such as cgroup integration and starvation prevention, to improve both fairness and performance (§3.4).
- CFL significantly improves the application performances, compared with state-of-the-art locks, by showing 2.4 – 7.1× higher throughput when running a blogging service via mitigating performance interference (§4.2) and up to 1.86× higher throughput with RocksDB read- /write mixed workloads because of proportional lock occupation (§4.3).

2 Background and Motivation

2.1 Importance of Lock Scheduling

The lack of a scheduling ability for lock occupation can generate various kinds of real-world problems, depending on the types of locks. There are roughly two types of locks: inter-process locks and intra-process locks. Inter-process locks protect system-wide shared data such as the page/inode/ directory cache, filesystem metadata, memory LRU lists, and so on. Since inter-process locks are shared across processes or process groups, unfair usage of such locks incur performance interference among applications/containers, and also, batch-job/low-priority ones can monopolize the lock resources in the worst case. On the other hand, intraprocess locks, such as locks for page faults, protect data on the same address space. The unfairness of such locks incurs priority inversion, resulting in a failure of performance optimizations [15] that utilize thread priority, such as increasing the priority of user-interactive or foreground tasks. Finally, regardless of the lock types, locks without scheduling abilities, such as FIFO-based locks, can exacerbate **the convoy effect**[1], where short CS threads are suspended by long CS threads. In some cases, this results in overall performance degradation by increasing the average lock waiting time.

To experimentally demonstrate this phenomenon, we run a performance-critical (blogging service) workload while co-running a batch workload from "Start" to "End" in Figure 1. For a performance-critical application, we run a blogging service using Blogbench, which mimics a real-world blogging service. The Blogbench program consists of four writer threads that create new blogs, modify their articlesand pictures, and add comments. The writer threads occasionally perform file renaming within the same directory to provide atomicity, which needs to hold a system-wide spinlock (*rename_lock*). While running Blogbench, we also execute a batch program with a various number of threads from "Start" to "End" as a low-priority application. The batch program performs move operations within its directory to imitate applications that rely on atomic file-rename operations for data updates, such as MapReduce and editors. The two programs are isolated and have individual cgroups. Here, since Blogbench directly interacts with users, we set the priority of the Blogbench eight times higher than that of the batch program. To observe performance interference by locks, we vary the number of threads in the batch workload from 16 to 64 and present the throughput (ops) of the blogging service while denoting its average latencies in a unit of μsin the legend.

As the number of threads in the batch workload increases, the blogging service experiences significant performance degradation due to the shared lock and its unfair usage. Here, the average lock waiting time is 1.54 μs without batch threads, while it increases up to 159.99 μs with 64 batch threads. As a result, the operations per second (OPS) of the blogging service drops by up to 80%. In the meantime, its average latency increases by up to 4.81 times, compared with the case where the blogging service runs alone. On the other hand, CFL, which we will explain in this paper, shows 2.5 times higher OPS than the vanilla Linux kernel when 64 batch threads are running, due to its scheduling ability. Like this, locks can incur performance interference between applications despite conventional scheduling efforts including cgroup. In this paper, we try to solve this problem via fair lock scheduling.

2.2 Lessons Learned from Conventional Wisdom

Since modern computer systems often co-locate multiple applications that have different performance requirements, priority-aware fairness is a crucial building block to provide performance isolation as well as SLA guarantees. Traditionally, operating systems control the occupation of various resources using individual schedulers, such as CFS for CPU and CFQ/BFQ for Storage I/Os.

Fortunately, locks have a lot in common with other stateless resources such as CPU and I/Os, and we can utilize various time-tested mechanisms and algorithms of theirs for lock scheduling. Here, we detail the requirements of lock scheduling and describe how other resource schedulers have solved similar problems from both performance and fairness perspectives.

(R1) Fairness: Diversity of resource usage The lock usage characteristics are diverse in two aspects depending on the tasks of the threads, and change over time. First, the CS lengths of locks vary depending on the types of operations or the size of the shared data (e.g., hash table lookup). Second, the frequency of lock acquisition requests can be diverse depending on the operations. These aspects equally exist in other resources, and they solve the problem by adopting virtual resource occupation time (e.g., vruntime/vdisktime), which denotes the accumulated amount of time for resource occupation, and keeping the value up-to-date.

(R2) Fairness: Proportional resource sharing The modern Linux distributions support proportional resource management, which distributes resources according to the ratio of priorities (e.g., 2:1 or 8:1), instead of binary priority where an urgent thread always takes precedence over others. To proportionally reflect priorities, existing schedulers adopt priority variables, such as nice/weight value for each resource (CPU/IO), and use them in calculating virtual resource occupation time. For example, CFS slowly increases the vruntime of high-priority applications so that they can occupy more CPU resources.

(R3) Fairness: Cgroup integration Recent Linux systems employ cgroup technology to control system resources in units of process groups, such as VMs or containers, and each fair scheduler supports cgroup integration. Therefore, the lock scheduler should also support cgroup integration to provide group-based scheduling. This is essential to align with current trends in container-based computing and to prevent performance interference by certain process groups with a large number of threads.

(R4) Performance:Work conservation When it comes to resource scheduling, a well-known desirable requirement for performance is work conservation[11], where resources should be occupied unless no waiter exists. This is significant also in locking because the overhead of lock idle time is augmented by the number of waiting threads ($wastedtime = lockidletime * \#of waitingthreads$). This work conservation is necessary particularly in the case of spinlock since the waiters waste their CPU resources while busy-waiting. To achieve work conservation within CPU cores, CFS sorts all the scheduling entities by their vruntime and picks the lowest vruntime entity as soon as the core becomes idle.

(R5) Performance: Hardware-awareness For the sake of high performance, various schedulers consider underlying hardware characteristics. For example, with respect to CPU architecture, the Linux AutoNUMA considers the NUMA topology and preferentially places a process closer to the memory that it is currently accessing, because of nonuniform memory latencies. This problem still exists in locks because the lock holder thread should hand over the lock to the threads on the same NUMA socket, if possible, to minimize the expensive data movement between remote memories [12]. In this paper, we primarily address NUMA architecture for our lock design, and it can be disabled and replaced depending on the underlying architecture.

2.3 Contemporary Lock Mechanisms

Previous studies [2–5, 10] about locks mainly focus on performance-wise optimizations. For instance, the Linux stock spinlock [12] is an MCS lock-based queued spinlock, which attempts to minimize cache-line traffic via a combination of a global lock and per-CPU local locks. NUMA-aware locks, such as CNA and Shfllock [9], try to hand over the lock to the threads on the same socket.

For lock occupation fairness,proposed a new mutex, called scheduler-cooperative lock (SCL), which monitors lock usage and forcibly suspends certain threads to achieve fairness. However, SCL violates some of the aforementioned requirements which we believe should be guaranteed. First, **(R1)** SCL does not use accumulated variables for lock occupation, and so it cannot properly handle real-world cases where lock hold time changes over time. Especially, the kernel version of SCL only considers the very previous lock hold time without keeping track of previous lock usage characteristics. Second, **(R3)** SCL does not support cgroup integration, and thus it oftentimes fails to prevent performance interference among processes or containers. Third, **(R4)** since SCL forcibly suspends certain threads for fairness, SCL is not work-conserving, and thus locks can be idle even if there are waiting threads. Finally, **(R5)** the design of SCL is built without consideration of hardware characteristics such as NUMA. These aspects of SCL significantly degrade performance and scalability, especially in many-core systems.

More recently, Syncord proposed a spinlock version of SCL with NUMA-awareness as one of its use cases. It implements SCL with a backoff scheme, which penalizes particular threads by forcing them to spin outside of the lock waiting queue without acquiring the lock. Unfortunately, it inherits all the drawbacks of SCL except for NUMA-awareness. For example, Syncord determines the backoff duration by the equation, ($current\_lock\_hold\_time * num\_threads - total\_lock\_hold\_time$). However, the num_threads variable does not decrease even if some threads stop requesting the locks because the design of Syncord does not have any method to automatically maintain up-to-date information, thereby penalizing an unnecessary amount of time in some cases. Additionally, Syncord does not support cgroup integration, which will be problematic under a consolidation setting. Finally, since spinlocks are mostly used for very short critical sections in the Linux kernel, its non-work-conserving nature induces a significant amount of overheads.

3 Design and Implementation of CFL

With the lessons learned from existing schedulers, we propose a newlock design, called CFL (Completely Fair Locking), which embodies a fair lock scheduler.We first briefly explain the scheduling mechanism and policy, and the detailed behavior and implementation will be explained later.

Scheduling mechanism CFL adopts a single queuebased lock mechanism, where a new lock waiter goes tothe tail and the head of the queue acquires the lock. For scheduling, CFL continually designates one of the lock waiter threads as the lock scheduler (LS), whose responsibility is to gather up-to-date lock usage information and place threads in an adequate position in the queue. The LS thread traverses the lock waiting queue and searches for the threads that meet scheduling criteria based on the scheduling algorithm. Whenever the LS thread finds adequate threads, it moves them to the front side of the queue by re-ordering the lock waiting queue so that they can acquire the lock sooner. At the same time, the LS thread also collects various information to apply the scheduling

policy, which we will explain in the next paragraph. Each lock instance has an individual LS thread for its lock waiting queue.

Scheduling policy Similar to vruntime of CFS, which indicates the accumulated virtual time of CPU occupation, we adopt virtual lock hold time (vLHT) to keep track of the lock occupation information for each thread. To achieve lock occupation fairness as well as high performance, CFL defines two scheduling criteria. First, the vLHT of the thread should be lower than the average vLHT of the current waiting threads. Second, the thread should be located on the NUMA socket whose per-socket average vLHT is the lowest among the sockets. When a thread meets both two conditions, CFL preferentially schedules the thread in advance to other threads. The required information, such as the average vLHT value of the waiting threads and the socket ID whose per-socket average vLHT is the lowest, is collected when the LS thread traverses the queue. Here, since the socket ID is for architecture-awareness, it can be disabled for non-NUMA machines or replaced with big.Little info. for such machines.

3.1 Design Overview

For better understanding, we first describe the fundamental behavior of CFL for lock occupation fairness without cgroup integration and optimizations. CFL is a queue-based lock that consists of a single global lock structure, a single lock-free doubly-linked link for lock waiting queue, and perthread queue nodes (qnodes). The lock-free doubly linked list has only a single entry point for initial insertion which is protected by atomic exchange. In the per-thread qnode, CFL add two new variables, virtual lock hold time (vLHT) and an indicator variable (sid) for architecture-awareness. The vLHT variable denotes the amount of time for holding the lock considering the thread priority. The sid is the ID of the socket on which the thread is currently running. In the lock structure, CFL saves avg_vLHT, which is the average vLHT for the entire queue, and min_sid which denotes the socket ID whose per-socket average vLHT is the smallest. We explain the core behavior of CFL using an example in Figure 2. [**one**] When a thread requires a lock, its qnode is inserted into the tail of the lock waiting queue. At that time, it stores vLHT information and the socket ID inside the qnode. The vLHT of each thread for the lock is maintained in the thread structure and updated on every lock acquisition/release. Meanwhile, [**two**] the LS thread (t_3) traverses the lock waiting queue from itself to the tail. The LS thread searches for any qnodes that meet two scheduling conditions. First, the vLHT of the qnode should be lower than the average vLHT of the entire queue (avg_vLHT). Second, the socket ID of the qnode should be min_sid. [**three**] If a qnode meets these two conditions, the LS thread moves the qnode (t_6 in the figure) to the next of the LS thread. In the meantime, the LS thread investigates the number of threads and their vLHT information. However, since the LS thread begins with itself for traversing, the vLHT information of the qnodes ahead of the LS thread (t_1 and t_2 in Figure 3) is not reflected. [**four**] Therefore, the LS thread performs reverse traversing to correctly update the vLHT information of the current lock waiting queue. [**five**] Afterward, the LS thread updates the avg_vLHT variable in the global lock structure ($(10 + 20 + 30 + 10 + 50 + 90)/6 = 35$) and hands over the duty of scheduling to the lastly moved qnode (t_6), thereby causing the t_6 thread become the next LS thread. Here, since the average vLHT of socket-0 ($(10 + 20 + 30 + 10)/4$) is still lower than that of socket-1 ($(50 + 90)/2$), the min_sid variable remains 0. [**six**] Finally, when t_0 thread releases the lock, its vLHT is updated, and the head (t_1) of the lock waiting queue acquires the lock.

In this way, CFL continuously prioritizes certain threads that have held the lock for a smaller amount of time than their shares, to achieve lock occupation fairness. This behavior is similar to CFS in that CFS continuously prioritizes threads that have low vruntime. CFS manages the waiting thread in a self-sorted RB (Red-Black) tree based on vruntime. Therefore, CFS can just pick the leftmost node without trying to find out which one has the lowest vruntime. However, the tree structure allows multiple entry points and thus requires another lock for itself in order to prevent an inconsistent state. Accordingly, CFL utilizes a lock-free doubly-linked list with a single entry point for initial insertions, protected by atomic exchange. For reordering the queue, CFL ensures that any qnodes prior to the LS thread are not reordered, so that lock release does not impact lock reordering. To prevent stale reference, CFL makes sure that it does not access a qnode that already became a lock holder during reverse traversing. In addition to fair lock occupation, CFL can also minimize the instances of transferring the lock to a different socket, since it moves qnodes that are located only on min_sid.

3.2 Implementation of CFL Spinlock

CFL borrows the basic structures of the Linux stock qspinlock [4], which is an MCS-based qspinlock. For each lock, there is a single global lock, which protects shared data, and a single lock waiting queue, which accommodates other threads that perform local spin. Its main purpose is to reduce cache-line bouncing by only allowing the head node to poll for the global lock. Similarly, CFL follows the basic principle of MSC-based qspinlock and does not use a global lock to update fields in the queue or the qnodes. Rather, CFL uses atomic operations to update variables or carefully avoid such race conditions (e.g., only a single Lock Scheduler (LS) thread traverses the queue.) In this section, we describe the detailed behavior of CFL using the simplified pseudo-code for the spinlock version of CFL.

3.2.1 ***Lock acquire/release.*** The pseudo-code of lock acquire/release is shown in Figure 3. First, when the lock is available and the waiting queue is empty, the thread promptly acquires the lock while storing the current time inside the thread structure for further vLHT calculation (lines 2 – 4). Otherwise, the thread initializes its qnode by storing its socket ID and vLHT, and inserts the qnode to the tail of the lock waiting queue (lines 9 – 12). Afterward, if it becomes the lock scheduler, it performs queue traversing (lines 16 – 17). When the LS thread can acquire the lock, it halts lock traversing and holds the lock while storing the start time inside its thread structure (lines 18 – 24).

When the thread finishes its critical section and releases the lock, the thread calculates its vLHT via the equation at line 30, which is similar as how to calculate vruntime in CFS. This means that the vLHT of a thread increases more slowly as its priority rises. Note that the current version of CFL shares a per-thread vLHT variable across locks that are enabled for lock scheduling.

3.2.2 ***Queue traversing.*** Queue traversing performs both the calculation of vLHT statistics and lock occupation scheduling. The pseudo-code of queue traversing is shown in Figure 4. First, the lock scheduler (LS) thread traverses the waiting queue from itself to the tail and calculates per-socket vLHT information (lines 4 – 7). When the LS thread encounters a qnode that is located on the socket of min_sid and has lower vLHT than avg_vLHT, the qnode is prioritized. In other words, the LS thread moves the qnode to the next of the lastly_moved_qnode (or to the next of the LS thread if there is no qnode moved previously) so that it acquires the lock quickly (lines 10 – 12). Afterward, the moved qnode becomes the new lastly_moved_qnode (line 13). During queue traversing, the LS thread keeps checking if it can acquire the lock or not (line 16). If the current LS thread is the head of the queue and the lock is available, the LS thread transfers the scheduling duty to the thread of the lastly_moved_qnode and immediately acquires the lock (lines 17 – 20). If none of qnodes have been moved, the LS thread designates its next thread as the new lock scheduler (line 22).

If the qnode is not the head or the lock is presently unavailable, the LS thread updates the vLHT statistics, because it is currently off the critical path. Here, since queue traversing was not performed for the qnodes that are ahead of the LS thread, the current vLHT statistics do not include their vLHT. Therefore, the LS thread performs reverse traversing to the head of the queue (lines 26 – 28). Afterward, using the up-to-date vLHT information, the LS thread investigates which socket has the lowest average vLHT, calculates the average vLHT of all qnodes in the queue, and then updates the corresponding variables (avg_vLHT and min_sid) of the lock structure (line 30). Finally, if there is any qnode that was moved, the current LS thread delivers the scheduling duty to the corresponding thread (lines 33 – 35). Otherwise, the current LS thread performs queue traversing again.

The per-socket vLHT information and average vLHT could be saved in the lock structure and updated upon every enqueue to keep the values up-to-date without repetitive calculation. However, such a design significantly increases memory consumption, and the amount of required memory linearly increases as the number of NUMA sockets increases. Locks should consume as little memory as possible for wide adoption and high performance because applications and operating systems utilize a substantial number of fine-grained locks. Therefore, for memory efficiency, CFL does not hold the per-socket vLHT values inside the lock structure and instead re-calculates them when the calculation can be off the critical path. Additionally, CFL takes into consideration both the vLHT and underlying architecture (socket ID in here) when scheduling lock occupation. Therefore, CFL can achieve not only fairness in lock occupation but also scalability, because the lock acquisition will be likely given to the threads on the same socket until min_sid changes. Finally, since CFL does not induce forcible sleep time or throttle certain threads, CFL fairly schedules the amount of lock occupation in a work-conserving manner.

3.3 Implementation of CFL Mutex

The mutex version of CFL is similar to the spinlock version except for the local spin and additional optimizations introduced in the previous research. Specifically, CFL adopts the spin_then_park mechanism, which performs spinning for a while and then sleeps to minimize wake-up overheads. Additionally, lock stealing , so-called barging, is enabled but in a fairness-oriented way. Lock stealing allows a new thread to acquire the lock instead of the sleeping head thread in the lock waiting queue, to improve performance by hiding the wake-up latency. This can significantly degrade fairness since such lock acquisition is not affected by the scheduling effect of queue traversing. For this reason, CFL allows only certain threads that meet the aforementioned scheduling conditions (socket ID and vLHT) to steal the lock. Finally, when CFL relocates qnodes, the mutex version proactively wakes up the corresponding threads to minimize wake-up overheads.

3.4 Optimizations

3.4.1 ***Efficient Queue*** . Traversing An increase in the number of threads inside the waiting queue linearly exacerbates the overheads of queue traversing. To minimize the traversing overheads, we adopt a traversing optimization that avoids unnecessary queue traversing, which is borrowed from [9]. Whenever the LS thread finishes queue traversing, it delivers the information of the lastly checked qnode to the new LS thread so that the new LS thread can traverse the waiting queue not from itself but from the next qnode of the lastly checked qnode to remove duplicate tasks. In this way, the new LS thread does not meaninglessly traverse already checked qnodes, thereby reducing the traversing overheads and increasing scalability. However, if min_sid has changed, the next LS thread should traverse the queue from the beginning since the scheduling criteria are changed. In this case, we do not deliver the lastly checked qnode information to the new LS thread.

We also have tried various conventional techniques to reduce the queue traversing overheads including sampling and traversing threshold. Although they contribute to performance improvement, the gain is mostly smaller than 10because most of the overheads do not originate from queue traversing after applying the first optimization in our evaluation. One of the reasons why queue traversing does not induce much overhead is that CFL assigns the most recently reordered qnode as a new LS thread. This guarantees that the new LS thread runs on the same socket as the previous LS threads, unless min_sid has changed. Therefore, the qnodes that have been accessed by the previous LS thread will already reside on the shared CPU cache, which minimizes cross-socket memory accesses.

3.4.2 ***Group Scheduling Support*** . In order to extend CFL to support cgroup integration, we create a lock subsystem inside cgroup and several knobs including weights. If cgroup is enabled, CFL additionally considers the cgroup ID of each thread when re-ordering the lock waiting queue. To achieve this, we add gid in the qnode structure to keep the corresponding cgroup ID of the qnode and min_gid in the global lock structure to find out a cgroup that has the lowest vLHT. Also, instead of thread priorities, CFL uses cgroup lock.weight. Supporting cgroup integration in CFL is similar to NUMA-awareness. When a qnode is initialized, the corresponding cgroup ID is stored in the gid of each qnode. Afterward, when the LS thread collects the vLHT statistics, it also calculates the aggregated vLHT of each cgroup, finds the cgroup that has the lowest aggregated vLHT, and then stores the ID in min_gid. Finally, CFL

adds another condition for scheduling criteria that the cgroup ID of a qnode should be min_gid, along with the aforementioned two conditions (line 10 at Figure 4).

3.4.3 ***Starvation Prevention.*** Since vLHT is accumulated over time, newly created threads or intermittent-intensive threads might monopolize the lock resource since their vLHT will be low, which can incur starvation of old threads. To prevent such starvation, when a thread initializes its qnode, CFL checks if its vLHT is zero or much lower than avg_vLHT, and if so, vLHT is initialized to avg_vLHT. In this way, CFL not only prevents starvation but also achieves better fairness of lock occupation, regardless of thread creation time. This concept is borrowed from the CFS scheduler and its vruntime management. When a thread is enqueued to a runqueue after being first created or waking up from sleeping, the vruntime of the thread is oftentimes initialized to the minimum vruntime of the runqueue. Since CFS picks the next threads with minimum vruntime, such initialization can ensure that the new thread will not monopolize the CPU but be scheduled soon. Meanwhile, CFL preferentially schedules threads with lower vLHT than avg_vLHT. Therefore, initialization with avg_vLHT ensures that new threads will not monopolize the lock but acquire the lock soon.

3.4.4 ***Strict Guarantee.*** Since CFL pursues work conservation, it sometimes sacrifices fairness for the sake of high performance and scalability. However, some systems might prefer strict service guarantees over high performance, even if it sacrifices work conservation. To fulfill this requirement, CFL also provides a strict_guarantee option. If this option is enabled, CFL detects any threads that have occupied an unfairly large amount of lock resource and makes them perform idling for a pre-defined threshold or until their vLHT becomes avg_vLHT. This optimization is inspired by the strict_guarantee option of BFQ, which performs idling instead of servicing other I/Os. This option is disabled by default, and we set the threshold to 6ms * (1+ilog(ncpus)), which is the default sched_latency_ns in CFS.

3.4.5 ***Grace Period.*** To improve performance via stronger NUMA-affinity while sacrificing short-term fairness, we also introduce another optimization option, which is grace_period. If this option is enabled, CFL guarantees min_sid not to be changed for a certain period of time (e.g., 50ms). This option can degrade short-term fairness by not immediately reflecting the lock occupation information but can improve the performance by keeping the lock within a socket for a while. In our evaluation, this option could improve the performance of the microbenchmark by up to 14%.

3.5 Summary

Previously, we have mentioned five requirements for fair lock scheduling. Here, we confirm if CFL can fulfill such requirements. First, **(R1)** CFL adopts virtual lock hold time, which indicates accumulated lock hold time, and keeps the value up-to-date by repetitive queue traversing. This aspect facilitates handling various real-world cases where lock usage changes over time since vLHT can keep track of the current and previous lock usage characteristics, instead of considering only the very previous or outdated usage patterns. Second, **(R2)** since vLHT increases at a different speed according to priorities, similarly with vruntime of CFS, CFL can proportionally distribute lock resources according to the priorities. Third, **(R3)** CFL supports cgroup integration by adopting several additional variables and scheduling criteria such as group ID, so that it can provide better fairness under multi-tenant scenarios. Fourth, **(R4)** by default, CFL always keeps the lock held without penalizing particular threads unless the lock waiting queue is empty, thereby achieving workconservation. Finally, **(R5)** CFL tries to maintain the lock holder within a socket using min_sid unless this breaks fairness, thereby improving scalability. We believe that, under a non-NUMA setting, this variable can be used to understand other architectural characteristics, such as the big.LITTLE architecture.

4 Evaluation

4.1 Evaluation Setup

We implement both the spinlock and mutex versions of kernel-level and user-level CFL by leveraging the implementation of Shfllock [9]. The data structure size of kernel-level locks is presented in Table 1. To support group scheduling, CFL requires four additional bytes per lock and per waiter to save the cgroup ID. For kernel experiments, we replace the kernel qspinlock and mutex with CFL in the Linux kernel v5.4.0. CFL decouples its locking mechanism and scheduling functionality, such as queue traversing and vLHT calculation. Therefore, we can enable/disable the functionality depending on the contention level of locks. In the case of user-level locks, we integrate CFL into LiTL , which allows transparent lock interposition without modifying application codes. In our evaluation, we compare CFL with various conventional locks as described in Table 2. In the table, (S) denotes spinlocks, whereas (M) denotes mutexes. Unless otherwise specified,we disable the optimization of CFL except for §3.4.1 and §3.4.3. We run our experiments on a system with a 4- socket Intel Xeon Gold 6130 CPU @ 2.10GHz featuring 32 hyperthreaded cores each (128 logical cores in total). In our experiments, we do not pin threads to cores unless otherwise specified, relying on the OS to make its choices, and thus load balancing and other optimizations are applied as usual.

4.2 Fileserver Benchmark

To evaluate the cgroup integration of CFL and its performance isolation, we re-perform the motivational experiment with 64 batch threads, comparing CFL with various kernel spinlock including CFL-NG, which disables cgroup integration. The numbers next to each lock in the legend of Figure 5 show the average latencies of the blogging service. As shown in Figure 5, the performance of CFL is comparable to most, but notably superior to both Syncord and Shfllock when the blogging service operates independently. When the two programs run together, CFL accomplishes the highest performance among the locks by showing 2.4× – 7.1× better OPS than others. In the meantime, CFL decreases the average latency of Blogbench by 54.7% – 84.8%, compared with other locks due to its better performance isolation. Since CFL-NG does not support group-based scheduling, it distributes a fair amount of lock resources to each thread without considering the cgroup information. Therefore, the batch program occupies most of the lock resources due to its larger number of threads, resulting in performance interference. In the case of Syncord, since it does not provide cgroup integration as with CFL-NG, it suffers from significant performance interference

by the batch program, which has a larger number of threads. Furthermore, the performance of the blogging service does not recover even after the batch program ends, because Syncord cannot keep track of up-to-date lock/thread information and thus the blogging service keeps being penalized as if the batch program exists.

4.3 RocksDB Benchmark

RocksDB provides a performance optimization option that decreases the CPU and I/O priority of background tasks, such as compaction, to minimize the delay of foreground tasks handling user requests. Unfortunately, this optimization can be diluted by synchronization because both foreground and background tasks compete for a certain set of locks to access data residing in memory, even when two tasks seem unrelated from the application perspective. For example, the page cache in the main memory is protected by xArray spinlock and shared across threads. RocksDB intensively tries to acquire the lock for various activities, such as adding/removing data to/from the page cache. Similarly, the lru_lock is also intensively required for managing the page LRU lists, such as shrinking active/inactive lists. Like this, multiple kernel-level spinlocks collaboratively determine the performance of RocksDB.

To verify if CFL can invigorate the optimization, we run several workloads using db_bench while enabling such locks to be scheduled and present both throughput and 99.9th tail latency. We run four workloads that issue write operations. The workloads OW/UP/RW/RRWR in the figure denotes overwrites/updaterandom/readwhilewriting/readrandomwriterandom, respectively. In our evaluation, RocksDB simultaneously launches around 194 threads on average for all the workloads. Here, 'Stock_wo_prio' denotes the stock version of the Linux kernel without the RocksDB priority option enabled, while the option is enabled with other cases. As shown in Figure 6, the stock version of Linux does not show meaningful performance gain even with the option enabled. Similarly, NUMA-aware locks, such as CNA and Shfllock, rather show slightly lower performance than Stock due to their additional overhead for NUMA-awareness. In these workloads, remote memory access caused by locking comprises a small portion of operational overheads, thereby NUMA-awareness showing a marginal improvement. On the other hand, CFL shows up to 1.86× better throughput and up to 25% better tail latency than the others. This improvement comes from the fact that CFL prioritizes foreground tasks over background ones during memory-related operations. We also tried to run Syncord in this evaluation but could not include the results because of a kernel panic. Syncord does not maintain up-to-date lock usage information, such as the number of active lock waiters, but the workload keeps changing the lock usage characteristics such as the number of threads and their CS lengths, thereby falsely penalizing lock waiting threads. Note that we could not observe meaningful performance differences across various locks with read-only workloads.

4.4 Microbenchmark

To evaluate the kernel mutex version of CFL, we run a crossdirectory rename benchmark, which is introduced in [8], and [20]. In this experiment, Group #1 (G#1) repetitively renames empty files to an empty directory with 16 threads. Meanwhile, Group #2 (G#2) renames empty files to an actively used directory that contains many files with 32 threads. In the Linux kernel, cross-directory rename operations require a system-level rename mutex for data consistency, called s_vfs_rename_mutex. Since the CS length of this lock depends on the size of the directory in this setting, the average CS length of G#2 is 214.26μs whereas that in G#1 is only 37.73μs.We execute the two groups in individual containers with identical priorities. Here, we add CFL-NS which disables the starvation prevention option.

Figure 7a shows the time-series performance of G#1. Stock and Shfllock fail to provide performance isolation due to their lack of fair lock scheduling capability, thereby degrading the performance of G#1 by 92% and 88% after G#2 begins. Since SCL lacks cgroup support, it gives more lock resources to G#2 because G#2 has a higher number of threads than G#1. Accordingly, SCL exhibits 64% of performance drop when G#2 begins. On the other hand, CFL considers the cgrouplevel vLHT information and thus provides the same amount of lock resources to each group, regardless of the thread counts in the groups. As a result, CFL shows around 49% of performance drop when G#2 starts, which means that CFL equally distributes lock resources to the groups. In terms of the combined throughput of G#1 and G#2, CFL outperforms other locks by up to 2.44× due to the reduced average lock waiting time by minimizing the convoy effect, which results from the similar effect of the shortest job first (SJF) scheduling policy. Specifically, CFL preferentially hands over the lock to G#1 over G#2 to fairly distribute the lock resource because the CS of G#1 is shorter than the other and thus G#1 needs more lock acquisitions to acquire a fair amount of lock resource. This indirectly results in minimizing the case where many threads become suspended while waiting for long CS threads to complete their tasks.

Meanwhile, CFL-NS always initializes vLHT of newly created threads to zero due to lack of the starvation prevention technique. Thus, G#2 occupies most of the lock resource at first because G#1 began first and has already high vLHT. Over time, they begin to fairly share the lock resources since the vLHT of G#2 quickly increases due to its higher thread counts and longer CS lengths. Finally, CFL-2 denotes CFL with increasing the priority of G#1 by two times in the middle of execution. As soon as the priority increases, the performance of G#1 promptly increases and reaches up to around 65% of the performance when G#1 runs alone. Since the performance of the workload highly depends on the amount of lock occupation, this result indicates that CFL-2 provides lock resources to G#1 around two times more than G#2.

Figure 7b shows the performance of the two groups when they run alone and together. Since they have the same priority, the performance of "together" should be at least 50% of "alone" in an ideal case. As shown in the figure, other locks fail to achieve group-based proportional lock scheduling. In the case of CFL, the differences between "alone" and "together" of G#1 and G#2 are 51% and 64%, respectively. This result demonstrates that CFL provides group-based proportional lock scheduling. Note that Shfllock and CFL show higher overall performance than Stock and SCL in the "alone" case due to their NUMA-awareness.

4.5 Userspace Application

We also test the userspace spinlock version of CFL to explore how CFL behaves in real database applications using UpscaleDB1. UpscaleDB is an open-source key-value store, where both "insert" and "find" operations require a database lock. Here, the CS length of "insert" operations is much longer than that of "find" operations, thereby exacerbating the convoy effect with conventional locks. As a result, CFL achieves 4.19× and 11.46× higher total throughput than CNA and Shfllock. The scheduling feature of CFL allows the "find" threads more # of lock acquisition in order to achieve fairness of lock occupation since their CS is short. This can indirectly increase the overall throughput because the average lock waiting time is significantly reduced, which results from a similar effect to the shortest job first (SJF) scheduling policy. Specifically, with 128 threads, CFL decreases the average waiting time of the database lock by 61.6% and 70.9%, compared with CNA and Shfllock, respectively.

4.6 Comparison with RelatedWork

4.6.1 SCL. To directly compare CFL with SCL, we run the author-provided microbenchmark , where two sets of threads execute 'while' loops for different periods within a critical section, protected by user-level mutexes. In this evaluation, we use various userspace mutexes that have different attributes. 'Pmutex' in the figure is the default pthread mutex, which randomly picks the next lock holder. 'MCS' denotes a FIFO-based queued mutex.We also introduce CFL-SG, which enables the strict_guarantee option. The bar graph of Figure 8 denotes the aggregate throughput of all threads to compare the performance. The red cross mark on each bar denotes a fairness factor, which shows the lock hold time ratio of the threads. Therefore, this value should be close to 1 to achieve lock occupation fairness.

Figure 8a shows the case with a lower number of threads and a non-NUMA setting. For the non-NUMA setting, we pin all the threads within a single socket to minimize the NUMA effect. As shown in the figure, Pmutex and MCS fails to achieve lock fairness due to their lack of lock scheduling. In the meantime, SCL guarantees lock occupation fairness in this setting by showing a fairness factor of near 1.0. Similarly, CFL also provides lock occupation fairness except for the uncontended cases (two and four threads). Since CFL pursues work conservation in general, CFL does not forcibly suspend particular threads when the lock is idle, even if they occupy more lock resources. However, when the strict_guarantee option is enabled, the lock occupation fairness can be guaranteed as seen with CFL-SG. Regarding performance, CFL outperforms SCL by up to 3.35 times even with a non-NUMA setting. This performance gain comes from the work-conservation nature of CFL and its various optimizations.

Figure 8b shows the case with a high number of threads and NUMA setting. In this case, all of SCL, CFL, and CFL-SG guarantee the lock occupation fairness. However, SCL shows the lowest performance out of the three fair locks because of its lack of NUMA-awareness. When comparing CFL with CFL-SG, CFL-SG shows at most 27% lower performance because CFL-SG sacrifices the work-conserving feature to further improve fairness by forcibly equalizing vLHT of each thread. Although we did not include the performance of CFL without the traversing optimization, introduced in §3.4.1, the optimization contributes around 27% of performance improvement at 128 threads, thereby increasing scalability.

4.6.2 ***SyncordNUMA-SCL.*** To directly compare CFL with Syncord NUMA-SCL, we run the author-provided crossrename benchmark [8], which is similar to §4.4 but has identical thread numbers for bully/ victim threads. Although the cross-rename operation is originally protected by a mutex, [8] switched it with its spinlock version of SCL. Thus, we also use the spinlock version of CFL for a fair evaluation. For the directory of the bully threads, we pre-created 1,000 empty files in advance. Since Syncord cannot keep track of up-to-date lock usage characteristics, such as # of lock waiters, the benchmark initializes them for each workload. Figure 9 denotes the total throughput and Jain's fairness index [20], which should be close to 1 to be fair. As shown in the figure, CFL outperforms Syncord in all cases by from 1.32× to 1.45× in terms of throughput, while both of them achieve fairness in lock occupation.

5 Discussion

The cases of scheduling failure due to locks [6], such as priority inversion, have been continuously reported. For example, [16] prioritizes a foreground application on mobile environments by handling lock contentions in the page cache layer. Unfortunately, most of the prior work relies on localized solutions that cannot fundamentally solve the other problems. However, we believe the scheduling ability of CFL can naturally mitigate a variety of priority inversion problems by preferentially giving the lock to higher-priority threads. In this way, higher-priority threads are less likely to be suspended by lower-priority ones.

The current Linux kernel accounts for each resource usage in a unit of cgroup/container and delivers resource scarcity to the users through PSI (Pressure Stall Information) so that the users can increase the priority of a container for a particular resource or reorganize hardware configuration. In order to extend our work, we can reveal to users the amount of time a container has spent waiting for locks, and employ this information to dynamically determine the priority of each container for lock resources.

Most of the locks evaluated in this paper have their own strengths. For instance, Shfllock [9] offers strong NUMA affinity according to our evaluation, while CNA has the lowest overhead in supporting NUMA affinity. In contrast, CFL delivers proportional lock sharing along with NUMA affinity. Therefore, CFL may introduce unnecessary overheads in systems that run a single application or do not leverage thread priority.

6 Conclusion

This paper proposes a new lock design, called CFL, which embodies a scheduling capability for lock occupation fairness along with architecture-awareness. CFL also provides a series of optimizations including cgroup integration for both fairness and performance. Our evaluation with various benchmarks confirms the practical necessity of CFL and demonstrates its superiority over state-of-the-art locks.

References

[1] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. 1979. The convoy phenomenon. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 20–25.

[2] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. *SIGPLAN Not.* 48, 8 (Feb. 2013), 157–166.

[3] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 851–865.

[4] Dave Dice. 2017. Malthusian Locks. https://arxiv.org/abs/1511.06035

[5] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In *Proceedings of Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (San Jose, California, USA). New York, NY, USA, 65–74.

[6] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News* 21, 2 (May 1993), 289–300.

[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. 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* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[8] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Yiming Du, Xiaoning Ding, and Tianyin Xu. 2023. HugeGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 62–73.

[9] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. [n. d.]. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. New York, NY, USA, 586–599.

[10] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware blocking synchronization primitives. In *In Proceedings of USENIX Annual Technical Conference* (ATC). 603–615.

[11] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. 2022. OS scheduling with nest: keeping tasks close together on warm cores. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. 368–383.

[12] Zichong Li, Yanbo Xu, Simiao Zuo, Haoming Jiang, Chao Zhang, Tuo Zhao, and Hongyuan Zha. 2023. SMURF-THP: score matching-based uncertainty quantification for transformer Hawkes process. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA). Article 833, 11 pages.

[13] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2016. 2DFQ: Two-Dimensional Fair Queuing for Multi-Tenant Cloud Services. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 144–159.

[14] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65.

[15] Jonggyu Park and Young Ik Eom. 2019. URS: User-Based Resource Scheduling for Multi-User Surface Computing Systems. *IEEE Transactions on Consumer Electronics* 65, 3 (2019), 426–433.

[16] Zhen Peng, Rizwan A. Ashraf, Luanzheng Guo, Ruiqin Tian, and Gokcen Kestor. 2023. Automatic Code Generation for High-Performance Graph Algorithms. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 14–26.

[17] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 349–362.

[18] Shanjiang Tang, Bu-Sung Lee, and Bingsheng He. 2018. Fair Resource Allocation for Data-Intensive Computing in the Cloud. *IEEE Transactions on Services Computing* 11, 1 (2018), 20–33.

[19] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 397–410.

[20] Xilie Xu, Jingfeng Zhang, Feng Liu, Masashi Sugiyama, and Mohan Kankanhalli. 2023. Efficient adversarial contrastive learning via robustness-aware coreset selection. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Article 3312, 28 pages.