Application-Informed Kernel Synchronization Primitives

Sujin Park[†] *Diyu Zhou* *Yuchen Qian* *Irina Calciu*[*] *Taesoo Kim*[†] *Sanidhya Kashyap*

EPFL [†]Georgia Tech [*]Graft

Abstract

Kernel synchronization primitives are the backbone of any OS design. Kernel locks, for instance, are crucial for both application performance and correctness. However, unlike application locks, kernel locks are far from the reach of application developers, who have minimal interpolation of the kernel's behavior and cannot control or influence the policies that govern kernel synchronization behavior. This disconnect between the kernel and applications can lead to pathological scenarios in which optimizing the kernel synchronization primitives under one context, such as high contention, leads to adversarial effects under a context with no lock contention. In addition, rapid-evolving heterogeneous hardware makes kernel lock development too slow for modern applications with stringent performance requirements and frequent deployment timelines.

This paper addresses the above issues with application-informed kernel synchronization primitives. We allow application developers to deploy workload-specific and hardware-aware kernel lock policies to boost application performance, resolve pathological usage of kernel locks, and even enable dynamic profiling of locks of interest. To showcase this idea, we design SynCord, a framework to modify kernel locks without recompiling or rebooting the kernel. SynCord abstracts key behaviors of kernel locks and exposes them as APIs for designing user-defined kernel locks. SynCord provides the mechanisms to customize kernel locks safely and correctly from the user space. We design five lock policies specialized for new heterogeneous hardware and specific software requirements. Our evaluation shows that SynCord incurs minimal runtime overhead and generates kernel locks with performance comparable to that of the state-of-the-art locks.

1 Introduction

With the ending of Moore's Law and Dennard scaling, the exponential growth of single-processor performance has come to a standstill. Hence, application developers now resort to customization, rather than generalization, to further squeeze out the performance from the hardware. For instance, different applications work best with changing underlying system mechanisms. Although a generic mechanism often provides acceptable performance, it rarely matches the performance of a specialized mechanism, whose performance difference often is an order of magnitude or more [10, 34, 59, 70].

Such a major improvement stems from the fact that specialization bridges the semantic gap between applications and the underlying system [23, 63]: It establishes the *context* under which an application requests functionality from the system. Thus, the underlying system can provide the most suitable implementation or even allow applications to provide their own implementation. The method of specialization is not new. For instance, prior works have targeted the widely used Linux OS that has become a major performance bottleneck for applications [29, 33, 47, 53, 61]. As a result, kernel customization has been extensively studied in the context of scheduling [34], networking [49], storage [70], and accelerators [10]. Although such works mostly focus on the scheduling aspect of the IO, they do not expose one of the basic building blocks of today's software design: concurrency control. Hence, this work takes a step in that direction by enabling the customization of the kernel synchronization primitives that have never been exposed to applications.

Kernel synchronization primitives, especially locks, are of paramount importance to ensuring correctness, achieving good performance, and scalability for applications [8, 9, 35, 37, 52]. Traditionally, kernel developers bake these primitives as a part of the OS implementation. Since it is difficult to change these primitives dynamically, the kernel developers favor supporting common scenarios and make all the decisions regarding their design and implementation. Thus, all these primitives are invisible and are out of reach of applications.

Given evolving hardware and changing software requirements, this static approach of lock design raises two issues: missing hardware and software contexts. From the hardware perspective, applications using kernel components, which rely on such generic primitives, suffer from regression issues in pathological cases [8, 37, 52]. In particular, these primitives suffer from a high level of contention with increasing core count [8], which requires further optimization for the underlying hardware [11, 21, 51]. In addition, increasing hardware heterogeneity in modern systems further exacerbates this issue [4, 5, 16, 41]. Second, from the software perspective, these baked generic primitives lack application context. As a result, it can lead to pathological cases, such as missing readers-writer context [11], priority inversion [35, 38, 52], scheduler subversion [58], and lock-holder preemption [36].

The current practice of addressing these issues involves developing synchronization primitives for specific scenarios [12, 21, 26, 39, 42, 45, 46, 51]. However, designing, implementing, and verifying new synchronization primitives is challenging. In addition, developers need a huge amount of effort to upstream and maintain them. To satisfy fast-evolving scenarios and requirements, synchronization primitives should be easily changeable and even on the fly instead of providing point-solutions as in previous works.

This paper proposes the idea of *application-informed kernel synchronization primitives* that enables users to develop custom *lock policies* to maximize performance or resolve pathological cases. For example, with an asymmetric multicore processor machine, in which processors operate at a different speed [4, 5, 16], application developers may want to prioritize lock waiters on fast cores to maximize performance. To demonstrate this idea, we design and implement SynCord, a framework built to safely modify kernel locks on the fly without recompiling or rebooting the kernel. We abstract and modularize the semantics of the locking primitives and expose them in the form of APIs. A developer uses these APIs for implementing policies, such as NUMA-awareness, priority boosting, readers-writer preference [11, 21, 37] etc. SynCord then verifies these policies and safely patches the running kernel in the end. It provides the capability to deploy custom code for a wide range of lock instances: from a single lock instance to a set of locks, or every lock in the kernel. Besides deploying lock policies, SynCord further allows users to profile locks at fine granularity. Our approach departs from the conventional tools profiling a fixed set of statistics for all kernel locks [73]. Instead, a user can now collect any lock statistic on arbitrary locks.

The ultimate goal of SynCord is to completely realize the idea of contextual concurrency control [57], which enables users to modify any synchronization primitives from the user space in a safe manner. As a first step in kernel lock customization, our SynCord prototype currently supports non-blocking locks. In particular, SynCord allows users to write their own logic for reordering lock waiters, setting priorities between competing threads to acquire a lock. We support three existing non-blocking primitives: ShflLock [37], CNA [19], and the stock readers-writer lock in Linux. We further demonstrate the generality of SynCord by adapting four different locking algorithms to the kernel and optimizing them based on our evaluation platform. In addition, we provide a case study of lock profiling using SynCord and show how it simplifies the performance analysis of a lock algorithm. Our evaluation shows that the custom algorithms developed with SynCord increase the application performance by up to three orders of magnitude compared to the generic locks.

This paper makes the following contributions:

- **Application-defined concurrency.** We propose the idea of on-the-fly modification of lock design. To realize that, we design and implement the SynCord framework.
- **APIs for non-blocking locks.** We provide a set of APIs that exposes the key decisions of non-blocking locks to implement various lock algorithms.
- **Lock algorithms.** We implement four lock algorithms and optimize them based on the platform. The optimized versions outperform generic locks up to three orders of magnitude.
- **Custom fine-granularity profiling.** SynCord provides custom, fine-granularity lock profiling that simplifies locks' performance analysis with smaller overhead.

2 Background And Motivation

Modifying kernel locking primitives without recompiling and rebooting the OS spans various domains of concurrent OS design. We first discuss the evolution of locks, followed by various mechanisms for kernel customization and the specific need for dynamic patching for kernel locks.

2.1 Lock evolution

Locks are widely used and heavily influenced by hardware. For example, queue-based locks minimize cache-line contention [51] among CPUs by forming a queue of waiters who spin on private cache lines. Hierarchical locks [11, 17, 21] improve application throughput for non-uniform memory access (NUMA) architecture, in which local memory is faster than remote NUMA memory. Such locks exploit the NUMA characteristic by batching requests from the same NUMA node at the cost of higher memory use and lower throughput in non-contended scenarios. CNA [19] and ShflLock [37] address these limitations by dynamically reordering the queue. ShflLock enforces *policies* given by hardware characteristics and software behaviors through shuffling. Although both locks allow designing new lock algorithms by abstracting both hardware and software requirements in the form of policy, their approach is insufficient for changing kernel locks on the fly. A developer still needs to recompile and reboot the kernel to test a new policy. SynCord allows users to develop custom policies and safely deploy them to a live kernel.

2.2 Kernel customization

With the introduction of fast IO devices, hardware accelerators and hundreds of cores, customizing the kernel on the fly is the new norm for improving application performance. However, this idea is not new, as Exokernel [22] is

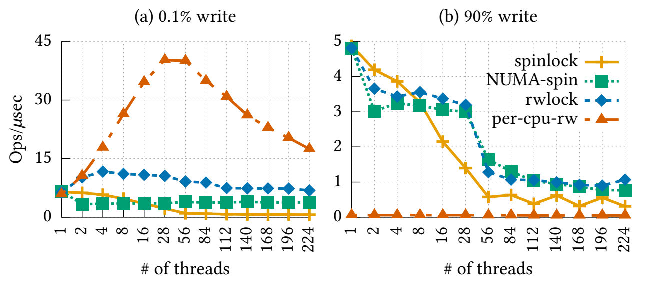


Figure 1: Impact of locks on throughput with different write ratios. The workload is a hashtable benchmark in the kernel [69] where a global lock guards the hashtable.

the first kernel design that enables customization by safely exporting hardware resources to untrusted library OSes and downloading application code to the kernel. Another prominent one that enables customization is the split-level I/O scheduling [70], which enables users to deploy custom I/O scheduling mechanisms across various layers of the storage stack.

Recently, Linux has been allowing user-level applications to customize the kernel by handling page faults in user space [14] or using the eBPF framework [23]. eBPF has seen wide deployment at various places in the kernel. For example, EXTFUSE speeds up user-level file systems with eBPF [6]. eXpress Data Path (XDP) [63] introduces a programmable network data path that allows a user-supplied eBPF program to control network packets. Moreover, recent work proposed delegating kernel operations to user space. For example, Syrup [34], ghOSt [28] and Scheduler BPF [15] allow users to specify scheduling policy and deploy it in the kernel networking stack and thread schedulers. Snap [49] enables the development of networking features in user space, while DPDK [61] and SPDK [29] provide libraries to accelerate packet processing and develop storage features. Compared to these works, SynCord takes a step further in customizing the kernel: it allows users to control the concurrency mechanisms in the underlying kernel.

2.3 Application-defined locking matters

Figure 1 illustrates the fact that one lock design cannot perform the best in all scenarios. For example, when the workload is read-dominant, rwlock outperforms spinlock because spinlock requires mutual exclusion even between read operations. In particular, per-CPU rwlock works better than a centralized one by avoiding cache traffic caused by a reader indicator across cores. However, with a write-dominant workload, the per-CPU rwlock performs the worst. In addition to the application semantics, underlying hardware also requires a different lock [17]. The NUMA-aware spinlock performs better than the MCS spinlock when threads execute across multiple sockets, but MCS can be a better choice on a single socket machine for the first few threads.

One might solve this problem by designing and implementing a special kernel lock. However, developing and maintaining kernel locks customized for each application and hardware is difficult, time-consuming, and costly. Meanwhile, SynCord eases the development of new lock algorithms. First, unlike other subsystems, synchronization primitives are not well isolated in the kernel. Hence, changes to synchronization primitives require understanding the surrounding details that impact a lot of other kernel codes. SynCord provides modularity for synchronization primitives. Second, SynCord APIs serve as an abstraction layer. These APIs hide the underlying tricky implementation details of lock, such as concurrency, memory model, and atomic instructions use. Instead, developers implement policies for scheduling waiters, such as what to do before and after acquiring a lock, and which type of waiters should be prioritized (§5). Moreover, locks designed with SynCord require no changes to the kernel's components and achieve similar speed up with only a few lines of code (Table 5).

2.4 The need for dynamic lock patching

Apart from the difficulty of designing and implementing new lock algorithms in the kernel, installing a modified kernel requires a system reboot. However, there are common scenarios where applications or underlying hardware change during execution, requiring live kernel lock changes. In terms of application changes, applications whose performance matters might change over runtime. This case is possible in a cloud environment, as multiple applications execute in a particular order. In addition to performance, applications try to maintain some form of service level agreements in the form of latency or fairness. Besides this, a scenario of runtime hardware modification is virtual machine (VM) live migration [2]. For example, if a cloud provider migrates a VM from a single socket machine to a multi-socket NUMA machine, users should modify their kernel locks' policies to handle the NUMA behavior. Moreover, with increasing hardware heterogeneity, applications require policies incorporating both hardware and software policies for better performance. One might argue that the traditional approach of kernel patching is sufficient. In this case, the conventional static kernel patching approach cannot efficiently handle such scenarios, motivating the need for the SynCord dynamic approach, which allows developers to implement specific APIs and patch a set of locks, while generally ensuring safety properties.

3 The SynCord Framework

SynCord is a framework for customizing kernel locks on the fly without recompiling or rebooting the kernel. To modify kernel locks, a user writes custom lock algorithms in user space, and SynCord safely deploys them in the kernel. SynCord can patch individual lock instances or every lock in the kernel. In addition, SynCord enables fine-grained profiling of kernel locks, helping users better understand the impact the kernel has on their application. We design

Type	Group	API	Description
Safe	General	① `void lock_to_acquire(lock)`	Invoked before acquiring the lock.
		② `void lock_acquired(lock)`	Invoked after acquiring the lock.
		③ `void lock_to_release(lock)`	Invoked before releasing the lock.
		④ `void lock_released(lock)`	Invoked after releasing the lock.
	Fast path	⑤ `void lock_to_enter_slowpath(lock, node)`	Invoked before entering the slow path.
		⑥ `bool lock_enable_fastpath(lock)`	If true, allow acquiring the lock by the fast path.
	Waiter reordering	⑦ `bool should_reorder(lock, anchor, curr)`	If true, move thread curr forward in the queue.
		⑧ `bool skip_reorder(lock, anchor)`	If true, skip the current reordering operation.
Unsafe	Lock bypass	⑨ `bool lock_bypass_acquire(lock)`	If true, bypass lock aquisition.
		⑩ `bool lock_bypass_release(lock)`	If true, bypass lock release.

Table 1: A summary of SynCord APIs. General APIs intercept the entry and exit points of the lock acquire and release phase. Today most of the lock algorithms have at least two paths to enter the critical section: fast path and slow path. Here, the fast path APIs intercept the fast path access to acquire the lock. Meanwhile, the slow path provides waiter reordering APIs that control the reordering of waiters for lock acquisition. Lock bypass APIs allow threads to bypass locks. The lock bypass APIs allow expert developers to design their algorithm, which comes at their own risk.

SynCord to modify kernel locks as a sandbox that adds new policies on top of existing locks.

Design goals. SynCord has three main design goals:

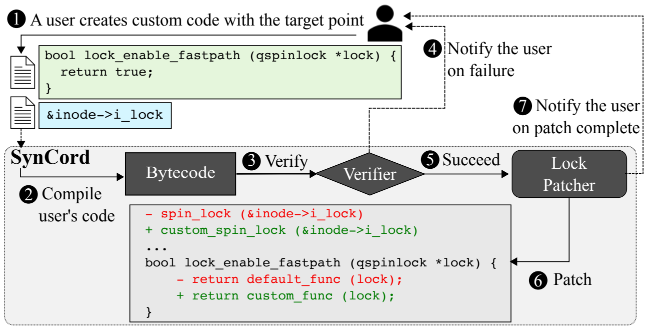
- *Correct lock patching.* SynCord must maintain the mutual exclusion of the lock instances being patched and should not introduce any correctness bugs through the process of patching.
- *Sandboxed user's code.* Users may provide unsafe code that leads to mutual exclusion violation. SynCord aims to prevent such code from corrupting the kernel as long as they use SynCord safe APIs, so that relieves users' concerns about the correctness of their lock design.
- *Usability and expressiveness.* SynCord aims to provide APIs expressive enough to tune kernel locks for various platforms or requirements.

To strike a balance between expressiveness and sandboxed impact, we design two sets of APIs. A set of *safe* APIs (①–⑧ in Table 1) guarantees mutual exclusion for general use, and the other set of *unsafe* APIs (⑨, ⑩) grant expert kernel developers full control of locks at their own risk.

Moreover, we envision that a single organization uses SynCord to modify kernel locking primitives. In particular, the sysadmins of that organization, with root privileges, handle the conflicting policies for various applications contending on the same lock or a set of locking instances. We follow this model because unlike other subsystems, locking primitives guard shared resources, which an unprivileged user should not change. In addition, we assume that such kernel changes do not occur frequently. Thus, a single policy optimized for underlying hardware or applications' usage patterns may last several minutes.

3.1 SynCord overview

Figure 2 illustrates SynCord's key components and workflow. SynCord exposes a set of APIs (Table 1) to abstract underlying lock implementation and allows users to write



❶ A user creates custom code with the target point
❹ Notify the user on failure
❼ Notify the user on patch complete
SynCord
Bytecode
❸ Verify
Verifier
❺ Succeed
Lock Patcher
❷ Compile user's code
❻ Patch

```
bool lock_enable_fastpath (qspinlock *lock) {
  return true;
}
```

```
&inode->i_lock
```

```
- spin_lock (&inode->i_lock)
+ custom_spin_lock (&inode->i_lock)
...
bool lock_enable_fastpath (qspinlock *lock) {
    - return default_func (lock);
    + return custom_func (lock);
}
```

Figure 2: Overview of SynCord's key components and workflow. (1) A user writes custom lock code and specifies lock instances to patch; (2) SynCord compiles the user's lock code (*e.g.*, eBPF) and (3) verifies basic properties of the compiled bytecode. (4) SynCord notifies the user if the verification fails, (5) otherwise loads the program into the kernel and generates a patch. (6) The lock patching module patches the kernel to call compiled bytecode on predefined hook points.

custom kernel locks in the abstracted layer. These APIs are the pre-defined hooking points that developers use for inserting their custom code to control the logic of underlying locks. To use SynCord, a privileged user first specifies the lock instance they want to patch and writes the custom locking code in C with SynCord APIs in a separate file (❶). SynCord processes this file in a semi-interactive fashion. It first reads the file and compiles the custom code (❷). It then passes the compiled program to the verifier that performs static analysis to validate the safety requirements (❸). The verification process usually takes a few milliseconds. If the verification fails, SynCord notifies the user (❹). Otherwise, it loads the policy into the kernel and gets a unique ID of the policy (❺). With the ID, SynCord patches the locking func-

```
def spin_lock(lock):
  lock_to_acquire(lock) # ① Hook the start of lock acquire

  if lock_bypass_acquire(lock): # ⑨ bypass lock acquire
    # lock acquisition is bypassed: used by lock experts
    return

  # If fastpath is enabled, first try to acquire the lock
  # instead of going into the wait queue
  if lock_enable_fastpath(lock) and # ⑥ can steal lock?
                CAS(&lock.state, UNLOCK, LOCKED):
    lock_acquired(lock) # ② lock is acquired
    return

  node = Node() # A node to join the queue
  lock_to_enter_slowpath(lock, node) # ⑤ Hook before enqueuing
  queued_spin_lock_slowpath(lock, node) # Time to join the queue
  lock_acquired(lock) # ② Hook the start of critical section

def spin_unlock(lock):
  lock_to_release(lock) # ③ Hook the end of critical section

  if lock_bypass_release(lock): # ⑩ bypass lock release
    # lock release is bypassed; used along with ⑨
    return

  lock.state = UNLOCK # Lock released; critical section ends
  lock_released(lock) # ④ Hook right after critical section
```

Figure 3: Pseudocode of `spin_lock` and `spin_unlock` with SynCord APIs. We place the waiter reordering APIs in the slow path of the existing `qspinlock` (`queued_spin_lock_slowpath`) function [13].

tions to execute the policy at pre-defined hooking points (❻). The patching process is time-consuming, as the patching module (`Livepatch`) has to find a quiescence period, *i.e.*, no task is executing the locking functions to patch. This period can last a few seconds. Finally, SynCord notifies the user after the patch completes (❼).

3.2 Programming with SynCord

To design a custom kernel lock with SynCord, a developer first specifies a set of lock instances to patch and implements a new policy by writing code blocks for each API. At a high level, there are two main purposes for developers to write code in the APIs: to enforce user-defined policies on scheduling waiters, and fine-grained profiling. From the scheduling perspective, a lock algorithm ensures the mutual exclusion property while scheduling a set of waiters based on user requirements *e.g.*, FIFO. Thus, SynCord exposes various means to schedule lock waiters, such as queue ordering and backoff schemes. In addition, both custom lock design and profiling often record extra information. Thus, we also provide auxiliary data structures (refer to §3.2.2) that serve as the extra storage space for the custom code.

3.2.1 SynCord APIs

Table 1 summarizes the APIs in the current SynCord prototype. The APIs expose several key behaviors of queue-based non-blocking locks, especially the ordering between lock waiters. We design SynCord APIs to be general across many existing lock designs and safe enough to use in user space. Most locks have well-known interfaces: acquire() and release(). Hence, the first category of our APIs (①–④) hooks these interfaces. Figure 3 shows the pseudo-code of `spin_lock` and `spin_unlock` with the hooking points for SynCord APIs. The *General* APIs allow users to intercept the entry and exit points of the acquire and release phase. These APIs are particularly useful for lock profiling as their hooking points are inspired by Linux's lockstat to profile every kernel lock. For example, users can use these APIs to record the time spent in acquiring the lock or the time spent in the critical section.

The second set of APIs—the *Fast path*—hooks the entry of a slow path (⑤) or controls the fast path of the lock (⑥). The fast path uses test-and-set-based lock for low contention scenarios [13, 19, 37]. For example, in `qspinlock`, CNA and ShflLock, a thread first tries to issue a test-and-set instruction to grab the lock, and only enqueues itself on failure. The fast path optimizes performance but may impact fairness, which now can be easily controlled with APIs. The slow path involves queue maintenance when the lock is in use, which applies to almost all queue-based lock algorithms [13, 44, 54, 55].

To design policies controlling the order to acquire a lock, we rely on a queue-based lock design that provides a powerful abstraction to reorder the waiting queue on the fly without using extra memory. Both CNA and ShflLock allow arbitrary dynamic queue ordering to achieve user-defined policies. Thus, we provide two *waiter reordering* APIs (⑦–⑧). `should_reorder()` moves a waiter in front of the queue by comparing the current node with an anchor node. `skip_reorder()` skips the reordering procedure. When the reordering is skipped, the waiting queue goes back to the first-in-first-out policy to maintain waiting threads. This is useful to enforce fairness for specific scenarios or purposes.

Although these APIs can implement several queue-based lock algorithms, they cannot change the basic mechanism of the underlying kernel locks. However, an experienced lock developer may want to redesign the kernel lock completely [17, 20, 21]. Thus, SynCord introduces a set of APIs that allow the custom code to bypass the lock acquire and release phase (⑨–⑩). These APIs grant users complete control of the kernel lock and thus allow users to design arbitrary lock algorithms. Since these APIs bypass underlying locks, it is the user's responsibility to correctly maintain the mutual exclusion property.

A point to note is that most of the APIs only introduce performance bugs with incorrect usage. Although the reordering API might affect the fairness, SynCord prevents threads from starvation with runtime checks. Meanwhile, our APIs are designed not to introduce correctness bugs (*e.g.*, infinite loops, mutual exclusion violations) except for the lock bypass APIs.

3.2.2 Auxiliary data structures

Sometimes, designing new lock algorithms or profiling existing locks might require extra information. For example, to implement a NUMA-aware lock scheduling algorithm (§5.1), each waiter needs to record its socket ID. Hence, we support auxiliary data structures to save some semantic information. In particular, we support three such types: per-node data, per-lock data, and global data. Per-node data is associated with a thread (node) that waits to acquire the lock. Its lifetime starts when a thread joins the waiting queue and ends when the thread acquires the lock. Per-lock data is associated with a lock instance, and global data plays the identical role as global variables in the kernel. Both per-lock and global data structures are created and destroyed explicitly by SynCord and follow the lifetime of the associated policy in most cases. The user, while implementing custom code, also defines the required types of auxiliary data structures, which get compiled along with the lock.

3.3 SynCord properties for lock design

Unlike current kernel customization approaches that localize resources within a process model, exposing dynamic lock modification requires reasoning about the mutual exclusion properties, minimizing the impact of starvation and ensuring that we patch the lock code correctly.

Sandboxed impact. If the user provides buggy code, SynCord should prevent such code from corrupting the kernel. SynCord guarantees code safety: memory safety (no access to illegal memory address), termination (no infinite loop), liveness (no deadlock) and mutual exclusion. The current verifier, which relies on the `eBPF` verifier, uses static analysis to enforce code safety. In particular, SynCord APIs pass a lock instance as a read-only argument to prevent arbitrary changes to the lock state during the lock acquisition and release phases. For instance, SynCord APIs (except the bypass ones) do not modify the functioning of the existing lock algorithm, such as atomic instruction, barriers, and concurrent executions. Because of this, SynCord does not introduce any new deadlock situation, meanwhile maintaining the liveness of the underlying lock even after adding the user logic. Our APIs only provide suggestions/hints to existing locks, as we do not change their underlying working. Hence, it is impossible to incur mutual exclusion violation. Meanwhile, we advise only lock experts to use the bypass APIs for complete access to the lock state.

Avoiding starvation. The reordering of waiters by SynCord introduces starvation that can severely affect the kernel response. We address this issue with bounded runtime checks. For example, if a custom lock uses backoff, we ensure that a waiter only waits for a maximum amount of time. To ensure this behavior, SynCord disables the custom logic for the respective APIs if the thread is suffering from starvation. We currently set the bounded time to 10 ms.

Correct lock patching. If a user provides the correct code, SynCord must address three issues: 1) only patch the required code, 2) apply the patch when the system is in a quiescence state to avoid any inconsistency, and 3) resolve the lock patch if multiple conflicting policies exist. We address these issues by using the mature patching service in Linux: `Livepatch` [32], which works as follows. `Livepatch` first generates a difference between the changed code. It then compiles the diff as a kernel module. Now, `Livepatch` inserts the module once the system reaches a quiescent state, *i.e.*, when a thread leaves the kernel space or CPUs are in idle mode.

In the case of SynCord, `Livepatch` can fail to insert the code if a user patches the same lock instances with multiple policies. We address this issue as follows: Before applying a patch, SynCord checks for an existing patch on the lock. If so, it aborts and reports the conflict to the system administrator. We also support re-patching the same locking call site by bypassing the previous check. Thus, SynCord provides the flexibility to resolve patch conflicts. For example, the user can completely override the old patch with the new one. Alternatively, they can develop and apply a new patch by manually merging those two patches.

4 SynCord Implementation

We now discuss the implementation of SynCord. First we present a summary of SynCord's implementation and then the two existing mature Linux kernel tools SynCord builds upon: `eBPF` (§4.1) and `Livepatch` (§4.2).

The current SynCord prototype targets non-blocking locks and supports both exclusive locks (*e.g.*, spinlocks) and readers-writer locks. For spinlocks, we implemented SynCord with the stock Linux spinlock, CNA [19] and ShflLock [37]. For readers-writer locks, we implemented SynCord with the readers-writer locks in ShflLock and the Linux kernel. Our current prototype uses Linux v5.4.

SynCord requires a one-time kernel modification to expose APIs (§3.2.1), extra `eBPF` helper functions, and runtime checks. Exposing the current SynCord APIs is relatively straightforward. Except for the waiter reordering APIs, we exposed all the other APIs by inserting a dummy function. We expose the waiter reordering for ShflLock and CNA in the form of a comparison function and also support the case for skipping the comparison with the `skip_reorder()` function. In total, we modify 143 lines of code in the Linux kernel. Our code is publicly available at https://github.com/rs3lab/SynCord.

4.1 eBPF for SynCord

`eBPF` allows applications to run custom code at specific points in the kernel (called hook points or target points). To use `eBPF`, a user first writes a program in C and compiles it into the `eBPF` bytecode. The kernel then loads the bytecode, verifies the memory safety, and then deploys it at the specified hook points. The `eBPF` verifier uses static analysis to check any illegal memory access in the program and also verifies if the program terminates. To guarantee the termination of a program, the verifier only allows bounded loops [65] and rejects unreachable instructions or out-of-bound jumps. eBPF further tries to ensure safety by only whitelisting a set of safe functions (called helper functions), so that applications can obtain system state, such as the current time, CPU ID, etc. SynCord exploits the eBPF safety guarantees to enforce several safety requirements (§3.3).

Several lock algorithms require threads to spin until they satisfy a set of specific conditions. However, the eBPF verifier does not allow loops since it cannot guarantee termination. To address this issue, SynCord introduces a new eBPF helper function: `backoff()`. With this helper function, a thread terminates its spinning either by meeting the defined condition or the specified time is over. SynCord further sets an upper limit on the timeout value (10 ms) to avoid starvation. We designed `backoff()` as an eBPF helper function so users can use these functions in any SynCord APIs.

4.2 Kernel livepatching for SynCord

Kernel livepatch [32, 56, 60, 66] modifies the kernel on the fly without rebooting the system. While eBPF alone can deploy user-defined code into the kernel (§4.1), the effect is global and thus affects all the lock instances in the kernel. To support a finer deployment granularity (§3) and auxiliary data structures (§3.2.2), SynCord uses `Livepatch`, specifically `Kpatch` [60] to deploy the custom code.

Implementing auxiliary data structures. We expose three types of auxiliary data structures: per-node data, per-lock data, and global data. Per-node data stores additional information when a thread joins a waiting queue in a queue-based lock design. Currently, the per-node data is 16 bytes but it is aligned at a cache-line boundary (64 bytes) to avoid false sharing. Thus, SynCord uses the remaining 48 bytes to store extra information. Presently, the current spinlock size is fixed at 4 bytes, and modifying the lock itself increases the memory footprint of any lock instance. Hence, for per-lock data, we use shadow variables [40] to allocate extra memory only for the target lock instances. In particular, we store the auxiliary data inside an in-kernel key-value store created by Livepatch. The address of a target lock instance serves as the key, while the value is per-lock auxiliary data. In addition to the per-lock data, global data such as per-CPU data can be also stored in that key-value store. SynCord frees the extra memory allocated as shadow variables when it removes the corresponding policy. In other words, SynCord does not modify the structure of the lock itself, instead it stores the additional per-lock data separately from the parent lock object. Hence, our design choice does not increase the memory footprint of all locks in the kernel. SynCord only allocates memory for the target locks, thereby having no additional memory footprint without an installed policy.

Workload	Lock: Usage
MWRL [52]	**rename_lock:** Rename files within a directory
lock1 [7]	**files_struct.file_lock:** fcntl and fd allocation
page_fault1 [7]	**mmap_sem:** Anonymous memory page-fault
LevelDB [25]	**futex** contention on futex hash bucket
Metis (wrmem) [48]	reader side of **mmap_sem** on page-fault
SCL-Victim [58]	**rename_lock:** Rename files from/to an empty directory
SCL-Bully [58]	**rename_lock:** Rename files from an empty directory to a directory with 1M files

Table 2: Lock usage in each benchmark.

```
# per-node auxiliary data structures
class node:
  ...
+ int socket_id # Store socket ID for the thread

def lock_enable_fastpath(lock): # Allow lock stealing
  return True

def lock_to_enter_slowpath(lock, node):
  node.socket_id = get_numa_id() # Record socket ID for waiter

# Return true if anchor and curr are in the same socket
# Applicable to both ShflLock and CNA
def should_reorder(lock, anchor, curr):
  return anchor.socket_id == curr.socket_id

# randomly skip reordering to pass the lock to another socket
def skip_reorder(lock, anchor):
  return random() & 0xffff
```

Figure 4: Pseudocode of a NUMA-aware lock with SynCord.

5 Use Cases

We discuss the design and evaluation of various use cases enabled by SynCord. We first cover lock scheduling algorithms that manipulate the ordering of the lock acquisition. SynCord provides the means to define a custom lock acquisition order either by reordering the waiting queue or by blocking specific threads from joining the queue. We classify the lock scheduling algorithms into two types: (1) acquisition-aware scheduling, which considers the characteristics of lock waiters to enter the critical section, such as NUMA-awareness (§5.1) or biased readers-writer (§5.4); (2) occupancy-aware scheduling, which involves scheduling based on the time a thread spends in the critical section (§5.2, §5.3). The second use case focuses on customized fine-grained profiling (§5.5). Finally, we discuss our experience of using SynCord to implement these use cases (§5.6). Note that every lock implemented with SynCord is marked with "*" in figures. "-static" represents Linux kernel compiled with a static implementation of the equivalent locking strategy.

Experimental setup. We evaluate SynCord on an 8-socket, 224-core machine equipped with Intel Xeon Platinum 8276L CPUs. The machine runs Ubuntu 20.04 with Linux kernel 5.4.0 with disabled hyperthreading. Table 2 lists the benchmarks we use for the evaluation.

5.1 NUMA-aware spinlock

Motivation. Modern servers have non-uniform memory access (NUMA) architectures, with multiple sockets, multi-

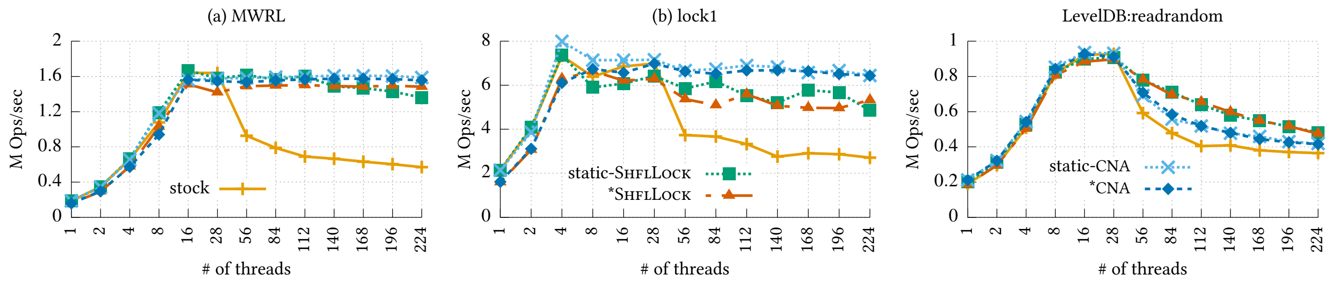


Figure 5: Comparison between kernel locks (stock, ShflLock, CNA) and their SynCord equivalent for NUMA-aware scenario.

ple cores, locally attached memories, and shared last-level caches. In such a server, accessing local memory is faster than accessing remote NUMA memory [21, 62]. NUMA-aware locks improve application throughput by batching lock acquisitions from the same socket together [19, 37].

Design. SynCord adopts the dynamic reordering mechanism for NUMA-aware locking from ShflLock and CNA, and batches requests from the same socket. NUMA locks ensure long-term fairness by periodically passing the lock to another socket. Figure 4 shows the implementation of this algorithm with SynCord. We add a per-node auxiliary integer: `socket_id` to record the socket ID of each waiting thread (line 4). The pseudocode denotes it inside `Class`, while the actual implementation is the `struct` of C code. We also enabled the fast path (lines 6–7) for lock stealing. If the fast path fails, the waiter enters the slow path after recording the socket ID (lines 9–10). The reordering process occurs in the slow path, and the underlying lock decides a reordering strategy. For example, in the case of ShflLock, the shuffler (`S`) first checks whether it should skip reordering by invoking the `skip_reorder` (lines 18–19) API. If not, `S` iterates the queue starting from itself; it invokes `should_reorder` (lines 14–15) with itself being the `anchor` and a waiting thread as `curr`. For all the waiting threads that make `should_reorder` return true, `S` moves that waiter forward, which groups waiters from the same socket. If `skip_reorder` returns true, `S` assigns the next waiter as the new shuffler, which ensures long-term fairness. The CNA lock also applies the same approach of grouping, but with a queue-splitting mechanism.

Evaluation. Figure 5 compares the stock version and two versions of ShflLock and CNA. The first is a static implementation requiring kernel re-install and reboot, while the other is the SynCord-based (marked $*$). Since ShflLock and CNA have a NUMA-aware policy on default, the evaluation shows how much overhead is introduced by SynCord's dynamic approach compared to the static implementation of the identical policy. We evaluate two microbenchmarks that contend on a single lock, and LevelDB's readrandom, which contends on the lock guarding the `futex` bucket. We find that the SynCord-based locks enforce a NUMA-aware policy on the fly without any significant overhead. Their performance is similar to their static counterparts and outperforms the stock version present in the Linux kernel, without having to compile or reboot the kernel. LevelDB's performance drops can be attributed to its use of a global database lock, which is not NUMA-aware.

5.2 Asymmetric multicore lock

Motivation. Asymmetric multicore processors (AMP) [4, 5, 16] consist of heterogeneous cores with different computing powers: energy-efficient slow cores and power-hungry fast cores. By combining both fast and slow cores in one processor, an AMP machine can adjust for dynamic usage patterns, for example, utilizing all cores to maximize the performance or using only slow cores for better energy-efficiency. Moreover, an AMP-aware scheduler can place low computation tasks on slow cores and place compute-intensive tasks on the fast ones. Unfortunately, current lock designs are unsuitable for the AMP architecture, as most lock designs assume homogeneous cores [41]. In particular, their performance significantly degrades when running on an AMP machine. This happens because slow cores execute critical sections up to 4× slower than faster cores [4], leading to lower throughput and higher latency [41]. Moreover, no lock design works efficiently for a multi-socket NUMA server, each has AMP.

Design. Our design is inspired by the LibASL [41], an AMP-aware user space lock without NUMA-awareness. Taking a step further, we extend LibASL's design for future AMP NUMA machines. LibASL works as follows: During a low contention scenario, it allows both slow cores and fast cores to acquire the lock to maximize performance. Meanwhile, during high contention, it penalizes slow cores so that fast cores can acquire the lock more aggressively for better performance. We slightly depart from LibASL with regards to penalty. In particular, we penalize slow core threads by forcing them to wait for a maximum of fixed time (10ms) before acquiring the lock. Providing appropriate wait time prevents starvation and ensures acceptable latency for workloads running on slow cores.

Figure 6 shows our version of the AMP-aware lock implemented using SynCord APIs. We use `MAX_WAIT_TIME` (line 2) as the maximum wait time for threads running on slow cores, and per-node socket ID (line 6) is used for NUMA-awareness.

```
1  class global_aux: # Global auxilary data structure
2  + int MAX_WAIT_TIME = 10ms # Max backoff for the slow core
3
4  class node: # Per-node auxiliary data structures
5    ...
6  + int socket_id # Store socket ID for the thread
7
8  def lock_enable_fastpath(lock): # Allow lock stealing
9    return True
10
11 def is_lock_unlocked(lock):
12   return lock.val == UNLOCK # Check if the lock is unlocked
13
14 def lock_to_enter_slowpath(lock, node):
15   node.socket_id = get_numa_id() # Record socket ID for waiter
16   cpu = get_processor_id() # Get the CPU ID of the thread
17
18   # Fast core thread directly enters; the slow one joins if:
19   #   1. the lock is not held or
20   #   2. it has been waiting for a predefined time
21   if is_slow_core(cpu):
22     backoff(lock, MAX_WAIT_TIME, is_lock_unlocked)
23
24 # Group same socket thread together
25 def should_reorder(lock, anchor, curr):
26   return anchor.socket_id == curr.socket_id
27
28 def skip_reorder(lock, anchor):
29   return rand() & 0xffff # Skip reordering to avoid starvation
```

Figure 6: AMP algorithm pseudocode with SynCord.

Similar to ShflLock, we allow lock stealing in the fast path (lines 8–9). On failing the fast path, the waiter assigns itself a socket ID (line 15) and checks its core type (line 21). If the waiter runs on a fast core, it immediately enters the slow path and joins the waiting queue. Otherwise, the waiter needs to wait before joining the waiting queue using the `backoff` function (lines 21-22). The thread on the slow core spins either until the `MAX_WAIT_TIME` has elapsed or if the lock has been released (lines 11–12). After returning from the `backoff` function, the waiter finally enters the slow path and joins the waiting queue. Once a thread enters the slow path, it follows a similar strategy as that of the NUMA lock for queue reordering and skipping. Thus, our NUMA-aware AMP lock prioritizes threads on fast cores before joining the queue, and further prefers the thread from the same socket after entering the queue. On the other hand, slow cores do acquire the lock after spinning for a predefined time (10ms). With the time-bound spinning, our approach prevents starvation while boosting application throughput.

Evaluation. Since there is no NUMA-AMP machine, we emulate the AMP environment by changing the CPU frequency. The fast cores are 4× faster than the slow cores and each socket has 14 fast and 14 slow cores, respectively. We use the same workloads as before (§5.1) and compare stock, ShflLock, and AMP. AMP is implemented statically (static-AMP) and using SynCord (∗AMP) to compare the overhead coming from SynCord. Figure 7 shows that AMP outperforms both ShflLock and stock by 1.5x and 13.4x each and maintains the performance with increasing core count. To dig deeper, we measure the throughput of fast and slow cores separately. We find that all three locks have similar throughput under low contention (eight threads) for all workloads. This happens because all three locks are able to steal locks during the fast path. However, when the contention becomes high, both stock and ShflLock allow slow cores to acquire the lock, leading to lower throughput. On the other hand, AMP lets fast cores acquire the lock most of the time and thus achieves higher throughput.

5.3 Scheduler-cooperative lock

Motivation. Patel *et al.* [58] described the *scheduler subversion* problem, in which competitive threads hold the same lock for varying times, leading to a subversion of CPU scheduling goals. In particular, current CPU schedulers let each thread have an equal share of the CPU time. Suppose two threads are spending most of their CPU time executing in a critical section protected by the same lock; one thread (the bully thread) holds the lock for a much longer time (*e.g.*, orders of magnitude longer) than the other (the victim thread). In this case, the bully thread essentially receives a much longer CPU time than the victim thread, subverting the scheduling goal. This can lead to pathological cases of denial of service attacks, and lower application performance [58].

Design. To address this problem, we implement a new lock algorithm, that strives to achieve fair hold time across threads: SCL. SCL utilizes ShflLock as the underlying kernel lock and uses SynCord APIs. The algorithm assumes that all threads have the same priority and receive the same CPU time. The algorithm tracks the time spent in the critical section for each thread. If one thread holds the lock longer than its share, it cannot acquire the lock until other threads have received an equal chance to acquire the lock. On top of SCL, we also implement a NUMA-aware version: NUMA-SCL.

Figure 8 presents the implementation of SCL. We have not included the NUMA part for brevity, which is similar to §5.1. We introduce several auxiliary data structures to implement this algorithm: a per-thread lock hold time variable (line 3), a per-thread variable for recording the beginning of the critical section (line 5), a per-lock integer for counting contending thread (line 9), and total hold time for each lock (line 10). The algorithm works as follows: Before a thread (`t`) joins the waiting queue, it first computes the lock quota based on the number of threads and overall lock holding time (line 14). Based on the time `t` spent in the critical section (line 15), it waits until other threads get the equal opportunity (lines 15–20) by backing off for that approximated time (line 20). We track the per-lock total lock hold time and per-thread lock hold time by tracking when the thread enters the critical section (line 23) and update the overall lock usage in the release phase (lines 26–31).

Evaluation. We evaluate five locks with a workload proposed by Patel *et al.* [58]. The workload creates two types of threads: victim threads and bully threads that contend on the `rename_lock`. Table 2 shows the configuration of the work-

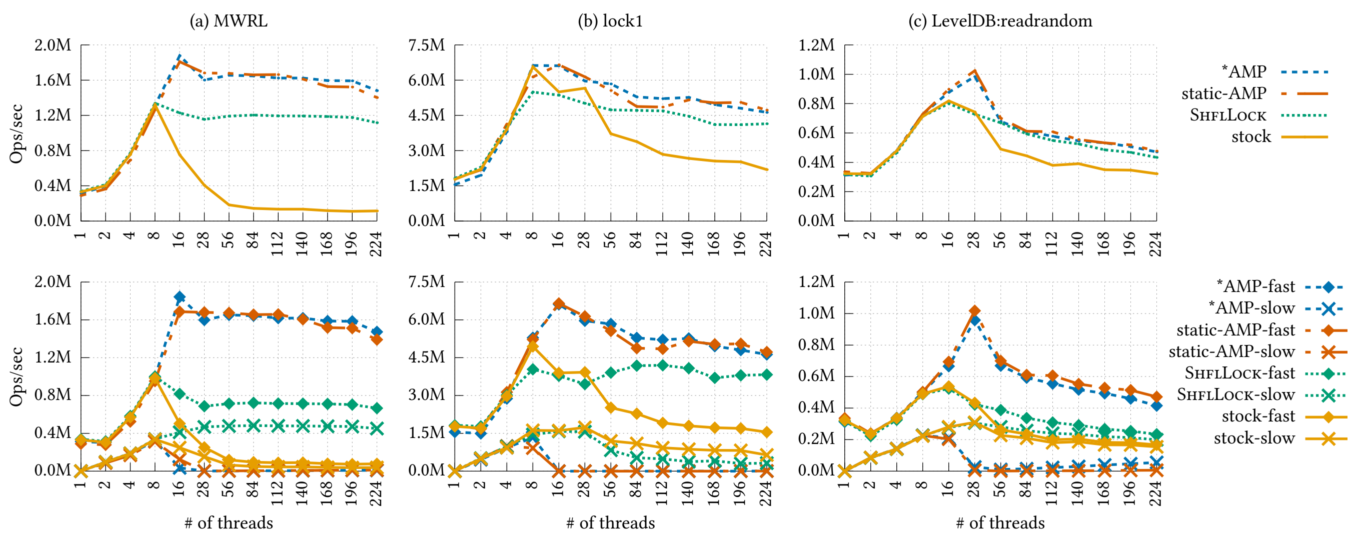


Figure 7: Overall throughput (up) and the throughput of fast and slow cores (down) for stock, ShflLock, and AMP implemented statically (static-AMP) and with SynCord (*AMP). Throughput of AMP-slow after 28 threads is very low but not zero. Refer to Table 4 for details.

```
class global_aux: # Global auxiliary data structures
    # Per-thread variable to record the lock hold time
+   int lock_hold_time<thread>
    # Per-thread variable to timestamp the beginning of CS
+   int cs_beg_ts<thread>

class lock: # Per-lock auxiliary data structures
    ...
+   int num_threads # Threads contending for the lock
+   int tot_lock_hold_time # Total lock hold time of all threads

def lock_to_enter_slowpath(lock, node):
    # Calculate the lock hold quota
    quota = lock.tot_lock_hold_time / lock.num_threads
    if lock_hold_time[curr_thread] > quota:
        # Exceeded local quota. Wait until threads
        # use same amount of quota
        wait = (lock_hold_time[curr_thread] * lock.num_threads)
        wait -= lock.tot_lock_hold_time
        backoff(lock, wait, None)

def lock_acquired(lock):
    cs_beg_ts[curr_thread] = get_time() # get timestamp after acq

def lock_to_release(lock):
    # Calculate the length of the critical section
    cs_len = get_time() - cs_beg_ts[curr_thread]

    # Update the lock usage for this thread and the lock
    lock_hold_time[curr_thread] += cs_len
    lock.tot_lock_hold_time += cs_len
```

Figure 8: Pseudocode of NUMA-SCL with SynCord. We omit the NUMA-awareness code (refer to Figure 4).

load, in which the bully holds the lock up to three orders of higher magnitude than the victim. These five locks include Linux's stock, ShflLock, static-SCL [58], *SCL (SCL without NUMA) and *NUMA-SCL. We implement the last two SCL locks with SynCord. Figure 9 shows the overall throughput and Jain's fairness index [31] of these locks. Jain's fairness

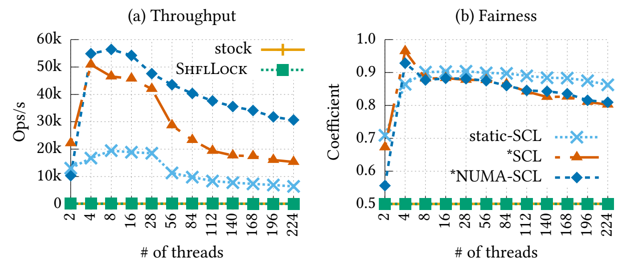


Figure 9: Impact of different lock designs on throughput and fairness. The workload is a rename program contending on rename lock where bully threads hold lock much longer than victim threads. Refer to Table 2 for more details.

index ranges from zero to one, where zero and one indicate completely unfair and fair, respectively. Since the SCL policy ensures that each thread holds the lock for the same length, all SCL versions allow the victim threads to hold the lock much more often than the bully threads. As a result, SCL implementations achieve orders of magnitudes higher throughput than both stock and ShflLock. Moreover, *NUMA-SCL outperforms the non-NUMA version (*SCL) by minimizing cache-line bouncing, thereby having the best overall throughput. The static implementation of SCL performs worse than SynCord versions, as it requires periodic scanning of the thread lists to remove inactive waiters. Such an approach is not required for SCL locks with SynCord because the user can dynamically decide the timeframe to enforce the lock hold time fairness.

We further confirm the aggregated lock hold time of bullies and victims. Figure 10 shows that a bully thread holds the lock

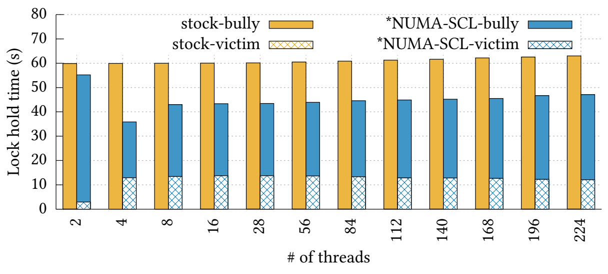


Figure 10: Total lock hold time of bully (B) and victim (V) threads.

for more than 99.9% in the stock version while ⁎NUMA-SCL ensures a better share between bully and victim.

5.4 Biased per-cpu readers-writer lock

This section shows the use of SynCord lock bypass APIs.

Motivation. Readers-writer lock (`rwlock`) is one of the most widely used primitives in Linux [37]. This primitive allows either multiple readers or one writer to acquire a lock. Most `rwlock` designs track active readers with a centralized readers indicator. However, the centralized readers indicator has poor scalability (Figure 1) because frequent atomic instructions for readers result in cache-line invalidation and coherence traffic. Prior work addressed this issue using distributed counters, but with the cost of high memory usage and longer writer latency. Thus, it is a good candidate only for read-intensive workloads [18, 42].

Design. We design a distributed readers-writer lock using the SynCord bypass APIs. Our design is inspired by the `BRAVO` design [18]. With SynCord's dynamic approach, users can enable the distributed `rwlock` only when needed and disable it to avoid unnecessary overhead, such as memory footprint and writer latency. Note that SynCord cannot ensure the mutual exclusion property for the unsafe APIs, and it is up to the lock developers to ensure the correctness. Moreover, we also use another unsafe function (`backoff_unsafe`) that waits indefinitely until the condition is met. We forbid the user from using this unsafe function with our safe APIs, as we throw an error on detecting it.

Figure 11 shows the per-CPU `rwlock` implementation in SynCord. We add two more fields per lock: `rbias` to track the read bias mode, and `visible_readers` table with cache-line aligned entries. Before a reader (`R`) acquires a lock, it first checks the read-biased mode. If the read-biased mode is set, `R` marks itself as an active reader (lines 7–9) and checks the `rbias` again due to a possible race from the writer's side (line 33). If `rbias` is still set, `R` bypasses the underlying lock, else it falls back to the underlying implementation (lines 11–13). At the time of release, `R` checks whether it acquired the lock in the read-biased mode (lines 16–20), and bypasses the underlying lock release if so. `R` is also responsible for setting `rbias` once it acquires the underlying lock without

```
1  class lock: # Per-lock auxiliary data structures
2    ...
3  + int rbias # When set, the lock is on readers-biased mode
4  + int visible_readers[max_cpu] # Distributed read indicator
5  # === Reader ===
6  def lock_bypass_acquire(lock):
7    if lock.aux.rbias: # If in read biased
8      lock.aux.visible_readers[cpu] = 1 # Mark the reader
9      if lock.aux.rbias: # No writer is waiting
10       return True
11     else: # Writer is present
12       lock.aux.visible_readers[cpu] = 0
13   return False
14
15 def lock_bypass_release(lock):
16   # Bypass the lock if reader acquired in rbias mode
17   if lock.aux.visible_readers[cpu] == 1:
18         lock.aux.visible_readers[cpu] = 0
19       return True
20   return False
21
22 def lock_acquired(lock):
23   # Enter the read-biased mode
24   if not lock.aux.rbias:
25     lock.aux.rbias = True
26
27 # === Writer ===
28 def wait_for_reader(lock, cpu):
29   return lock.aux.visible_readers[cpu] == 0
30
31 def lock_acquired(lock):
32   if lock.aux.rbias:
33     lock.aux.rbias = False # Revoke bias
34     for i in range(0, NUM_CPU): # Wait for readers to leave
35       backoff_unsafe(lock, wait_for_reader(i))
```

Figure 11: Pseudocode of the `BRAVO` algorithm. We omit the code to get CPU ID for brevity.

bypassing it (lines 24-25) so that other readers can directly acquire the read lock by setting their respective indicators. In the case of writer acquisition, a writer (`W`) first acquires the underlying writer lock. `W` will only acquire the lock when there are no active readers that hold the underlying lock. After that, `W` further checks for the read-biased mode (line 32). If it is active, `W` first disables it and waits for all readers to exit the critical section that used the per-CPU indicator (lines 34–35).

Evaluation. Figure 12 compares the stock `rwlock` with our version using SynCord (⁎per-CPU `rwlock`). We evaluate these locks with a `page_fault1` microbenchmark from `will-it-scale` [7] and Metis [48], a MapReduce framework, contending on the reader side of `mmap_sem`. Figure 12 shows that ⁎per-CPU `rwlock` outperforms stock by 2.2x and maintains the performance with increasing core count.

5.5 Dynamic lock profiling

Motivation. Several works [8, 19, 35, 37, 52] have shown that kernel locks mostly determine the scalability of applications. Hence, lock profiling tools are critical to understanding a lock's performance. Unfortunately, only few tools exist for kernel lock profiling, and even those have limited analysis capability. For example, developers often use Linux perf [1] to measure the aggregated CPU cycles in each code region. While this is useful to find lock contention, it does not pro-

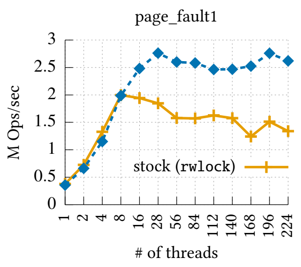


Figure 12: Comparison between Linux `rwlock` and our distributed per-CPU `rwlock` with SynCord. Refer to Table 2 for workload details.

policy	acquisitions	x-socket	avg-batch	violation
SCL+Reorder$_{Bully}$	4042	277	14.59	1659
SCL+Backoff$_{Bully}$	918765	8678	105.87	538

Table 3: Statistics collected by SynCord to analyze our two different implementations of NUMA-SCL: SCL+Reorder$_{Bully}$ and SCL+Backoff$_{Bully}$. We collect this result from the same benchmark used in §5.3 with 224 threads.

vide any lock-specific performance stats, such as the time spent in the critical section. As a result, Linux perf is not always the right tool for understanding lock performance. On the other hand, Linux provides another tool: `lockstat` [73] which exposes various statistics of kernel locks. However, it profiles all the kernel locks together and only shows the system-wide statistics. Moreover, a user can neither choose the lock instance nor specific lock data to profile. To make matter worse, `lockstat` requires a kernel to be compiled with a specific configuration, which significantly increases the size of every lock data structure and introduce performance overhead. For example, a kernel with `lockstat` uses 423MB of extra memory over the stock version even from the booting.

Design. SynCord can patch locks at various granularities, from an individual lock instance to a set of locks. In addition, SynCord provides the ability to profile any lock instance with arbitrary lock-specific performance stats. In particular, a user can now customize which set of lock instances to profile with specific statistics (even the algorithm-specific ones). For example, a user can profile only the rename lock and count the number of lock acquisitions across socket instead of collecting a set of statistics for all the locks in the kernel. We reproduce the statistics provided by `lockstat` using SynCord. Since the hooking points of SynCord General APIs (Table 1) have the same context as those of lockstat, the implementation is straightforward with simple updates of the counters and timestamp in each API.

Evaluation. We compare the overhead of lock profiling between `lockstat` and SynCord. `lockstat` keeps track of 10 counters for each lock. We implement two versions of SynCord-based `lockstat` having identical 10 counters and

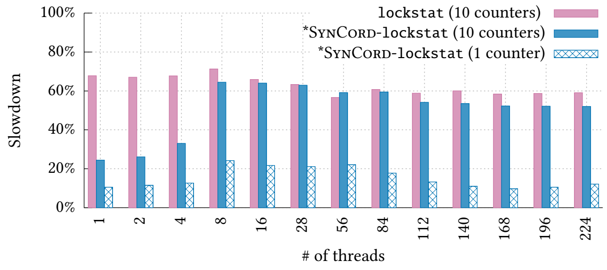


Figure 13: Performance overhead of lock profiling in MWRL for both `lockstat` and our custom implementation of `lockstat` with SynCord. The overhead increases with increasing counters.

policy	# threads	acqst-fast	acqst-slow
AMP+Reorder$_{Fast}$	8	20,340,943	8,192,987
AMP+Backoff$_{Slow}$	8	24,825,644	5,267,531
AMP+Reorder$_{Fast}$	224	571,712	571,354
AMP+Backoff$_{Slow}$	224	37,937,529	54,941

Table 4: Statistics collected by SynCord to analyze our two different implementations of AMP: AMP+Reorder$_{Fast}$ and AMP+Backoff$_{Slow}$. We collect the result for MWRL with the same environment as in §5.2.

one counter. Figure 13 shows that `lockstat` constantly incurs a 60% application slowdown. We observe that the overhead of SynCord profiling increases with an increased number of counters. With 10 counters, SynCord profiler incurs a similar overhead as that of conventional `lockstat`, while it is only 24% for one counter. Furthermore, unlike `lockstat`, SynCord opens the door for lock profiling even on production servers, as SynCord can dynamically turn on the profiling feature and does not introduce overhead when the profiling is uninstalled.

Simplifying performance analysis. We now illustrate an example of how SynCord's custom profiling can significantly simplify the performance analysis. Our initial implementation of the AMP (§5.2) and SCL (§5.3) algorithms only utilized the reordering mechanism of CNA and ShflLock (§5.1). In particular, we only used the `should_reorder` API to enforce both NUMA-awareness to handle hardware characteristics and SCL and AMP algorithms for software requirements. However, the reordering API alone has limitations in strictly enforcing the policy, thus it is difficult to have desired performance as the number of threads increases.

To understand the NUMA-SCL algorithm, we collect the following statistics for the reordering-based algorithm (SCL+Reorder$_{Bully}$), and the current algorithm-based on bully backoff with NUMA-aware reordering (SCL+Backoff$_{Bully}$): the total number of lock acquisitions, the number of lock acquisitions across socket, the average times of lock passing within a socket, and the number of policy violations (a bully thread acquires the lock when it shouldn't). Table 3 shows that SCL+Reorder$_{Bully}$ fails to enforce the policy, leading to a high violation count. This happens because the reordering approach guarantees that a waiter will acquire the lock once it is part of the waiting queue. Thus, a bully can still grab the lock even though it is penalized, if a victim has not yet joined the waiting queue. Instead, the backoff-based approach (SCL+Backoff$_{Bully}$) avoids this problem by preventing the bully thread from joining the wait queue, thereby effectively enforcing the policy.

Table 4 shows the collected statistics for AMP locks using two variations: AMP+Reorder$_{Fast}$ which uses reordering only to enforce both fast-core preference and same-socket preferences, and AMP+Backoff$_{Slow}$ shown in §5.2. We collect the number of lock acquisitions separately on the fast and slow cores. Under low contention (eight threads), both algorithms have similar throughput. However, under a highly contended scenario (224 threads), the reordering approach cannot correctly enforce the fast-core preference anymore, leading to a performance drop. The same reasoning from SCL holds true here too.

5.6 Experience with SynCord

Policy	LoC	Time
NUMA-aware spinlocks (§5.1)	6	3 hours
Scheduler-cooperative locks (§5.3)	30	18 hours
Asymmetric multicore-aware locks (§5.2)	15	8 hours
Scalable reader-writer locks (§5.4)	36	5 hours
Lock profiling (§5.5)	36	1 hour

Table 5: Development effort of the use cases.

We now discuss the efforts and lessons we have learned in developing the use cases with SynCord.

Lock development effort. Table 5 summarizes the development effort of all use cases. We spent most of the development time understanding, debugging, and testing the lock algorithm. Implementing the algorithms in SynCord involves only tens of lines of code and is relatively straightforward. In addition, SynCord allows users to modify the lock without kernel installation and rebooting, which dramatically reduces the overall development effort.

Hardware is (still) the key factor of performance. We include the NUMA grouping policy even for SCL and AMP algorithms to make them perform well on a NUMA machine. Initially we did not plan to include the NUMA grouping algorithm since we thought the performance gain achieved by task-specific scheduling should dominate. For example, due to the big performance gap between fast and slow cores in AMP machines, scheduling fast cores should achieve good enough performance without NUMA grouping. However, this has proven to not be the case. LibASL performs even worse than ShflLock on our emulated AMP NUMA machine due to the cache-line bouncing among sockets. Hence, we believe that a lock developer still needs to consider and prioritize the underlying hardware when designing a lock.

Avoid overloading APIs. As discussed in §5.5, our initial implementation of the SCL and AMP algorithms only used reordering mechanisms to enforce both customized policy and the NUMA-grouping algorithm. However, the profiling results show that complex policy in the reordering API does not work at a high thread count. The root cause of this issue is that the node reordering mechanism in ShflLock and CNA cannot strictly prevent certain nodes from acquiring a lock once they join the queue. Specifically, the reordering mechanism makes the scheduling decision as soon as it encounters the first suitable candidate, without considering whether a better candidate exists in the entire waiting queue. With the current lock implementation, we believe the best way to address this issue is to avoid overloading APIs. Specifically, a lock developer should not specify too complicated policies in one API and, if possible, divide the policies into small pieces and enforce each one of them with the suitable APIs. For example, in our AMP implementation, we enforced the NUMA grouping policy through reordering and the fast core scheduling policy when cores enter the wait queue.

6 Discussion

6.1 Generality of SynCord

SynCord's current implementation focuses on non-blocking locks, but the fundamental concept can be applied to other synchronization primitives, such as blocking locks (mutex) [55], RCU [50], seqlocks [27], and wait events [68]. SynCord can similarly modularize key decisions and behaviors of these synchronization primitives and expose them as APIs. For example, SynCord may expose the condition to wake up or park a thread as APIs for a blocking lock. Moreover, with *lock bypass* APIs, which grant privileged users complete control of the kernel lock, the implementation of kernel locks can even be moved to user space.

6.2 Support for multi-tenancy

The current SynCord prototype targets an environment where one user or a set of users trust each other to share the machine. In this scenario, SynCord can resolve patch conflicts (i.e., a lock instance is patched by multiple patches) by allowing a privileged user to provide a final patch (§3.3). However, this approach no longer works in a multi-tenancy cloud environment.

To extend SynCord to a multi-tenancy environment, we plan to apply the widely used *cgroup* and *namespace* concepts to kernel synchronization primitives. For example, a synchronization cgroup controls the set of kernel synchronization primitives that an application can change. A synchronization namespace virtualizes the underlying synchronization primitives. For example, for a shared kernel lock, a corresponding virtual lock is created in every synchronization namespace. The kernel implements an arbitration mechanism, such as time-sharing, to decide which virtual lock can hold the physical lock. Hence, the synchronization namespace enables applications to modify synchronization primitives while still enforcing performance isolation. With this change, we can drop privilege for SynCord to each namespace instead of limiting it to system administrators.

6.3 Easier programming of lock policy

In the current SynCord prototype, a user needs to provide the entire code for each policy. For example, both AMP (§5.2) and SCL (§5.3) policies include NUMA grouping code to achieve better scalability. We can extend SynCord to compose multiple policies into one (*i.e.*, merge NUMA-grouping and AMP to get NUMA-aware AMP) unless there is a duplicate use of the API between policies.

SynCord currently only supports C to write customized lock code, but it can be further extended to support more languages. With several toolchains [24, 30, 64] that allow writing `eBPF` programs in languages other than C, SynCord can support more expressive and memory-safe languages such as Python or Rust with better libraries and ecosystems.

6.4 Patching time

With our environment, the patching time is typically 10-40ms, and at a maximum of 5 seconds in an extreme case: patching every spinlock in the kernel. Livepatch applies the patch by checking each thread's stack whether the thread has invoked any patched functions. If so, Livepatch waits until all threads exit the patched function. One of the reasons for long patching time derives from a few tasks blocking the completion of a patching operation. The five-second patching time looks unreasonably long to us and further reducing the patching time is possible by sending a fake signal to the remaining blocking tasks.

7 Related Work

In section §2.2, we covered several works that customize kernel from user space. While SynCord is the first work to expose the concurrency control to user space, the need for different locking designs depending on hardware or software requirements has been also discussed in previous works.

Dice and Kogan [18] presented the `BRAVO` lock which can dynamically switch between a centralized reader-writer lock and a lock using distributed reader indicators. When the `BRAVO` algorithm detects a *read-biased* workload pattern, it improves scalability between readers by using distributed reader indicators, but at a cost of a potential slow down when released from the read-biased mode. This clear trade-off shows that the logic to turn on the read-biased mode is critical to performance, but `BRAVO` relies on heuristic parameters because it was impossible to change lock algorithms on the fly. Once ported to SynCord, when to turn on the read-biased mode can be an open problem for users.

Recently, Chehab *et al.* [17] proposed CLoF, which generates hundreds of possible combinations of spinlocks to create NUMA-aware hierarchy locks. CLoF first generates a set of locks and then selects the best performing lock for the target environment. The work emphasizes the need for different lock designs depending on the underlying hardware, which strengthens our motivation. Constructing and finding the best performing lock is an orthogonal topic to our work.

Lock profiling. Synchronization primitives play a significant role in application scalability, thus sophisticated lock profiling tools can help users understand the performance bottleneck. In addition to the `perf` [1] and `lockstat` [73] introduced in §5.5, there are several more works to improve lock profiling.

SyncPerf [3] hooks `pthread` related functions and provide a synchronization analysis tool with low overhead. Tallent *et al.* [67] used a sampling approach to quantify lock contention and SyncProf [71] collects profiling data through repetitive execution of an application to find the source of contention. Although these studies contributed to a better understanding of lock contention, all three works profile locks used in the user space, not the kernel locks.

LockDoc [43] traced the usage of kernel locks and automatically generates documentation describing the order in which each lock should be used. The work mainly focused on inferring locking rules instead of the performance aspect of each lock. wPerf [72] analyzes waiting events to find the source of a performance bottleneck, but does not provide lock-specific statistics.

8 Conclusion

Kernel synchronization primitives greatly impact application performance and scalability. However, the current kernel design prevents application developers from controlling the kernel synchronizations. This paper proposes *application-informed kernel synchronization primitives* which allow users to customize the kernel locks on the fly. To showcase the idea, we implemented SynCord, a framework for user-defined custom lock code. SynCord allows a privileged user to deploy custom code into the kernel lock safely and efficiently without recompiling or rebooting the kernel. We show that applications can leverage SynCord to achieve significant performance gains by developing hardware- or workload-specific lock algorithms. Furthermore, SynCord enables users to perform custom, fine-granularity lock profiling, which can greatly simplify the performance analysis of lock algorithms.

9 Acknowledgment

We sincerely thank our shepherd Adam Belay and the anonymous reviewers for their insightful feedback. This work is supported by Huawei.

References

[1] perf: Linux profiling with performance counters , 2014. https://perf.wiki.kernel.org/index.php/Main$_p$age.

[2] J. Ahn, C. Kim, J. Han, Y. ri Choi, and J. Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, 2012. USENIX Association.

[3] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 298–313, 2017.

[4] Apple. Small chip. Giant leap., 2020. https://www.apple.com/mac/m1/.

[5] ARM. Processing Architecture for Power Efficiency and Performance, 2021. https://www.arm.com/why-arm/technologies/big-little.

[6] A. Bijlani and U. Ramachandran. Extension Framework for File Systems in User space. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 121–134, Renton, WA, July 2019.

[7] A. Blanchard. will-it-scale, 2013. https://github.com/antonblanchard/will-it-scale.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, Oct. 2010.

[9] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[10] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 973–990, Virtual, Nov. 2020.

[11] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 157–166, Shenzhen, China, Feb. 2013.

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

[13] J. Corbet. MCS locks and qspinlocks, 2014. https://lwn.net/Articles/590243/.

[14] J. Corbet. User-space page fault handling, 2015. https://lwn.net/Articles/636226/.

[15] J. Corbet. Controlling the CPU scheduler with BPF, 2021. https://lwn.net/Articles/873244/.

[16] I. Cutress. Intel Alder Lake: Confirmed x86 Hybrid with Golden Cove and Gracemont for 2021, 2020. https://www.anandtech.com/show/15979/intel-alder-lake-confirmed-x86-hybrid-with-golden-cove-and-gracemont-for-2021.

[17] R. L. de Lima Chehab, A. Paolillo, D. Behrens, M. Fu, H. Härtig, and H. Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[18] D. Dice and A. Kogan. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 315–328, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[19] D. Dice and A. Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 12:1–12:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6281-8.

[20] D. Dice and A. Kogan. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 173–183, 2021.

[21] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–256, New Orleans, LA, Feb. 2012.

[22] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[23] M. Fleming. A thorough introduction to eBPF, 2017. https://lwn.net/Articles/740157/.

[24] foniod. RedBPF, 2021. https://github.com/foniod/redbpf.

[25] S. Ghemawat and J. Dean. LevelDB, 2019. URL https://github.com/google/leveldb.

[26] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 649–662, Denver, CO, June 2016.

[27] G. Haskins. seqlock: serialize against writers, 2008. https://lwn.net/Articles/296209/.

[28] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. Ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[29] Intel. Introduction to the Storage Performance Development Kit (SPDK), 2016. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-the-storage-performance-development-kit-spdk.html.

[30] iovisor. BPF Compiler Collection (BCC), 2021. https://github.com/iovisor/bcc.

[31] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998. URL https://arxiv.org/abs/cs/9809099.

[32] S. Jennings. Kernel live patching, 2014. https://lwn.net/Articles/619390/.

[33] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[34] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[35] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[36] S. Kashyap, C. Min, and T. Kim. Scaling Guest OS Critical Sections with eCS. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[37] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[38] S. Kim, H. Kim, J. Lee, and J. Jeong. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST)*, pages 345–358, Santa Clara, CA, Feb. 2017.

[39] A. Kogan. [PATCH 0/3] Add NUMA-awareness to qspinlock, 2019. URL https://lkml.org/lkml/2019/1/30/1191.

[40] Linux. Shadow Variables, 2018. https://www.kernel.org/doc/Documentation/livepatch/shadow-vars.txt.

[41] N. Liu, J. Gu, D. Tang, K. Li, B. Zang, and H. Chen. Asymmetry-aware Scalable Locking. *CoRR*, abs/2108.03355, 2021. URL https://arxiv.org/abs/2108.03355.

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

[43] A. Lochmann, H. Schirmeier, H. Borghorst, and O. Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

[44] W. Long. qrwlock: Introducing a queue read/write lock implementation, 2014. URL https://lwn.net/Articles/579729/.

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

[46] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.*, 33(4):13:1–13:62, Jan. 2016.

[47] Madhavapeddy, Anil and Scott, David J. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM*, page 61–69, Jan. 2014. ISSN 0001-0782.

[48] Y. Mao, R. Morris, and F. M. Kaashoek. Optimizing MapReduce for Multicore Architectures. In *MIT CSAIL, Technical Report*, 2010.

[49] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

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

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

[52] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[53] C. Min, W.-H. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. SOLROS: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

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

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

[56] ORACLE. Ksplice, 2018. https://ksplice.oracle.com.

[57] S. Park, I. Calciu, T. Kim, and S. Kashyap. Contextual Concurrency Control. In *18th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XVIII)*, Virtual, May 2021.

[58] Y. Patel, L. Yang, L. Arulraj, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Virtual, Apr. 2020.

[59] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[60] J. Poimboeuf. kpatch: dynamic kernel patching, 2014. https://lwn.net/Articles/597123/.

[61] T. L. F. Projects. DPDK, 2021. https://www.dpdk.org/.

[62] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1871-0.

[63] RedHat. Achieving high-performance, low-latency networking with xdp, 2021. https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1/.

[64] rust bpf. Rust-bcc, 2021. https://github.com/rust-bpf/rust-bcc.

[65] M. Rybczyńska. Bounded loops in bpf for the 5.3 kernel, 2019. https://lwn.net/Articles/794934/.

[66] J. Slaby. kGraft, 2014. https://lwn.net/Articles/603185/.

[67] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 269–280, jan 2010.

[68] L. Torvalds. Linux Wait Queues, 2005. http://www.tldp.org/LDP/tlk/kernel/kernel.html.

[69] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 11–11, Portland, OR, June 2011.

[70] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[71] T. Yu and M. Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 389–400, 2016.

[72] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: Generic Off-CPU analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, Oct. 2018.

[73] P. Zijlstra. lockstat: documentation, 2003. https://lwn.net/Articles/252835/.