Fast Consensus Using Bounded Staleness for Scalable Read-Mostly Synchronization

Haibo Chen, *Senior Member, IEEE*, Heng Zhang, Ran Liu, Binyu Zang, and Haibing Guan

Abstract—Reader-mostly synchronization schemes, such as rwlocks and RCU, aim to maximize parallelism among readers, but many existing designs either cause readers to contend, or significantly extend writer latency, or both. This paper attributes such a problem to the lack of a fast consensus protocol between readers and writers, by which the two parts cooperate to obey the semantics of a synchronization construct. This paper describes FCP, a fast consensus protocol among readers and writers that provides scalable read-side performance as well as small writer latency for TSO architectures. The heart of FCP is a version-based consensus protocol between multiple non-communicating readers and a pending writer. FCP leverages *bounded staleness* of memory consistency to avoid atomic instructions and memory barriers in readers' common paths, and uses message-passing (e.g., IPI) for straggling readers so that the writer latency can be bounded. To demonstrate the effectiveness of FCP, this paper applies FCP to construct a scalable reader-writers lock (rwlock) and a scalable RCU implementation. Evaluation on a 64-core machine shows that FCP significantly boosts the performance of the Linux virtual memory subsystem, a concurrent hashtable and an in-memory database. Micro-benchmarks show that FCP achieves smaller reader-side latency and lower writer-side latency when compared to state-of-the-art rwlocks and RCU implementation.

Index Terms—Consensus, reader-mostly synchronization, reader-writer lock, RCU

1 Introduction

Read-mostly synchronization targets usage scenarios where accesses to shared data structures are read-mostly. For such scenarios, using mutual exclusion locks, such as spin locks or mutex locks can significantly constrain parallelism among readers. To this end, researchers have described synchronization constructs to unleash parallelism by allowing readers to proceed in parallel in the absence of writers. Currently, there are two widely used read-mostly synchronization constructs, namely reader-writer locks (rwlock) and read-copy update (RCU).

Reader-writer locking is an important synchronization primitive that allows multiple threads with read accesses to a shared object when there is no writer, and blocks all readers when there is an inflight writer [13]. While ideally rwlock should provide scalable performance when there are infrequent writers, it is widely recognized that traditional centralized rwlocks have poor scalability [9], [10], [34]. For example, it is explicitly recommended to not use rwlocks unless readers hold their locks for sufficiently long time [9].

Researchers sometimes relax semantic guarantees by allowing readers to see stale data. Specifically, RCU [26] allows a writer to proceed in parallel even if there are still readers in progress, but at the cost that a writer must use the single-pointer update to change a data structure at a whole and memory cannot be reclaimed until all readers have finished referencing the old data. Since its invention, RCU has been widely used in Linux kernel for some relatively simple data structures. However, it would require non-trivial effort for some complex kernel data structures and may be incompatible with some existing kernel designs [10], [11]. Hence, there are still thousands of usages of rwlocks inside Linux kernel [25].

After a careful study of the design of rwlock and RCU, we find that the consensus among readers and writers is the key limiting factor to the performance and scalability of both schemes. Specifically, rwlock needs to know whether there are any readers pending until a writer can enter the writer-side critical section. While in RCU, a writer (or a dedicated reclamation thread) must know if all readers have discarded references to a stale object before it can free the memory.

However, current consensus protocols for both rwlock and RCU still suffer from some performance issues. Typical implementations of rwlock usually use a shared counter among readers and writers to do consensus, which, however, causes severe performance and scalability issues on a relatively large number of cores. While there have been intensive efforts to improve the scalability of rwlocks, prior approaches either require memory barriers and atomic instructions in readers [22], [28], or significantly extend writer latency [5], or both [2], [12]. For RCU, typical implementations usually use scheduler-based consensus (also called quiescence detection) such that all CPU cores must have executed past a context or ring switch, and thus may have latency that is at least a schedule quantum (e.g., 10 ms in Linux by default) in the degenerate case. Besides, some prior designs cannot cope with OS semantics like sleeping inside critical section, preemption and supporting conditional synchronization (e.g., wait/signal) [2], [12].

• *The authors are with the Shanghai Key Laboratory for Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {haibochen, shinedark, byzang, hbguan}@sjtu.edu.cn, naruilone@gmail.com.*

Manuscript received 30 June 2015; revised 24 Feb. 2016; accepted 3 Mar. 2016. Date of publication 9 Mar. 2016; date of current version 16 Nov. 2016. Recommended for acceptance by X. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2539953

TABLE 1
A Comparison of Consensus Protocol in Read-Mostly Synchronization

	Traditional	brlock1	brlock2	C-SNZI	Cohort	RMLock	PRW	Percpu-rwlock	RCU
No memory barrier in read						✓	✓	✓	✓
No atomic instruction in read			✓			✓	✓	✓	✓
No comm. among readers		✓	✓			✓	✓	✓	✓
Sleep inside critical section	✓			✓	✓	✓	✓	✓	✓
Condition wait	✓			✓		✓	✓	✓	-
Writer preference	✓		✓	✓	✓	✓	✓	✓	-
Reader preference	✓				✓				-
Short writer latency w/ small #thread	✓			✓	✓		✓	*	-
Unchanged rwlock semantic	✓	✓	✓	✓	✓		✓	✓	

** The writer latency of Percpu-rwlock is extremely long in most cases.*

This paper describes the FCP, a fast consensus protocol for read-mostly synchronization for Total Store Ordering (TSO) architectures. Like prior consensus protocols such as those in brlock [2], [12], instead of letting writers actively bookkeep status regarding inflight readers, FCP decentralizes such information to each reader and only makes a consensus among readers when a writer explicitly inquires. By leveraging the ordered store property of TSO architectures, such as x86 and x86-64, FCP achieves truly scalable reader performance. On TSO, it not only requires no atomic instructions or memory barriers on the common path, but it also limits writer latency when there are concurrent readers.

The key of FCP is a version-based consensus protocol between multiple non-communicating readers and a pending writer. A writer advances the lock version and waits for other readers to see this version to ensure that they have left their read-side critical sections. Unlike prior designs such as brlocks, this design is based on our observation that *even without explicit memory barriers, most readers are still able to see a most-recent update of the lock version from the writer within a small number of cycles*. We call this property *bounded staleness*. For straggling readers not seeing and reporting the version update, FCP uses a message-based mechanism based on inter-processor interrupts (IPIs) to explicitly achieve consensus. Upon receiving the message, a reader will report to the writer whether it has left the critical section. As currently message passing among cores using IPIs is not prohibitively high (around 1,500 cycles as shown in Section 3.1) and only very few straggling readers require message-based consensus, a writer only needs to wait shortly to proceed. Hence, FCP closely follows the design philosophy of "common case fast, rare case correct".

As a reader might sleep in the read-side critical section, it may not be able to receive messages from the writer. Hence, a sleeping reader might infinitely delay a writer. To address this issue, FCP falls back to a shared counter to count sleeping readers. As sleeping in read-side critical sections is usually rare, the counter is rarely used and contention on the shared counter will not be a performance bottleneck even if there are a small number of sleeping readers.

FCP is built with a parallel wakeup mechanism to improve performance when there are multiple sleeping readers waiting for an outstanding writer. As traditional wakeup mechanisms (like Linux) usually use a shared queue for multiple sleeping readers, a writer needs to wake up multiple readers sequentially, which becomes a scalability bottleneck with the increasing number of readers. Based on the observation that multiple readers can be woken up in parallel with no priority violation in many cases, FCP introduces a parallel wakeup mechanism such that each reader is woken up by the core where it slept from.

We have implemented FCP as a kernel mechanism for Linux, which compromises around 300 lines of code (LoC). We have also applied FCP to construct a scalable rwlock and an RCU implementation inside Linux kernel. To further benefit user-level code, we also created a user-level FCP library (comprising about 500 LoC) and added it to a user-level RCU library (about 100 LoC changes). The rwlock version of FCP can be used in the complex Linux virtual memory system (which currently uses a semaphore), with only around 30 LoC changes. The implementation is stable enough and has passed the Linux Test Project [1]. We have also applied the rwlock version of FCP by substituting a contended rwlock in the Kyoto Cabinet database [18].

Performance evaluation on a 64-core AMD machine shows that rwlock-FCP has extremely good performance scalability for read-mostly workloads and still good performance when there are quite a few writers. The performance speedup of FCP over stock Linux is 2.85X, 1.55X and 1.20X for three benchmarks on 64 cores and FCP performs closely to a recent effort in using RCU to scale Linux virtual memory [10]. Evaluation using micro-benchmarks and the in-memory database shows that rwlock-FCP consistently outperforms rwlock in Linux (by 7.37X for the Kyoto Cabinet database). Our evaluation also shows that FCP also improves the performance and quiescence detection latency of RCU.

The rest of this paper is organized as follows. The next section gives a brief overview of the rwlock and summarizes related work. Section 3 describes the design of FCP by illustrating design rationale, its key data structure, and the fast consensus mechanism. Section 6 describes the parallel wakeup mechanism. Section 7 describes the implementation issues and usages. Section 8 evaluates the performance and scalability of FCP and Section 9 concludes this paper.

2 Background and Related Work

The key to read-mostly synchronization is a fast consensus protocol that incurs low overhead in the reader side and requires only bounded latency in the writer side. Table 1 shows a comparative study of different consensus designs, using a set of criteria related to performance and functionality. The first three rows list the criteria critical to reader performance, including memory barriers, atomic instructions

and communication among readers. The next four rows depict whether each design can support sleeping inside critical section (which also implies preemption) and conditional wait (e.g., wait until a specific event such as queue is not empty), and whether the consensus is writer or reader preference. The last two rows indicate whether the writer in each design has short writer latency when there are a small number of threads, and whether the design retains the original semantics of rwlock.

2.1 Consensus in Reader/Writer Locks

The reader/writer problem was described by Courtois et al. [13] and has been intensively studied afterwards. However, most prior rwlocks implement consensus by sharing states among readers and thus may result in poor critical section efficiency on multicore. Hence, there have been intensive efforts to improve consensus of rwlocks.

Big-reader lock (brlock). The key idea of doing consensus for brlock is trading write throughput for read throughput. There are two designs of brlock: 1) requiring each thread to obtain a private mutex to acquire the lock in read mode and to obtain all private mutexes to acquire the lock in write mode (brlock1); 2) using an array of reader flags shared by readers and writer (brlock2). However, brlock1 requires heavyweight operations for both reader and writer sections, as the cost of acquiring a mutex is still non-trivial and the cost for the writer is high for a relatively large number of cores (i.e., readers).

Brlock2, like FCP, uses per-core reader status and forces writers to check each reader's status, and thus avoids atomic instructions in reader side. However, it still requires memory barriers in readers' common paths. Further, both do not support sleeping inside read-side critical sections as there is no centralized writer condition to sleep on and wake up. Finally, they are vulnerable to deadlock when a thread is preempted and migrated to another core. As a result, brlocks are most often used with preemption disabled.

FCP borrows the per-core reader status design from brlock2, but uses a version-based consensus protocol instead of a single flag to avoid memory barriers in readers' common paths and to shorten writer latency. Further, by leveraging a hybrid design with counters as a fallback solution for sleeping readers, FCP can cope with complex semantics like sleeping and preemption, making it viable to be used in complex systems like the virtual memory subsystem in Linux kernel.

C-SNZI. Lev et al. [22] use scalable nonzero indicator (SNZI) [17] to implement consensus for rwlocks. The key idea is instead of knowing exactly how many readers are in progress, the writer only needs to know whether there are any inflight readers. This, however, still requires actively maintaining reader status in a tree and thus may have scalability issues under a relatively large number of cores [8] due to the shared tree among readers.

Cohort lock. Irina et al. leverage the lock cohorting [15] technique to implement several NUMA-friendly rwlocks, in which writers tend to pass the lock to another writer within an NUMA node. While writers benefit from better NUMA locality, its readers are implemented using per-node shared counters and thus still suffer from cache contention and atomic instructions. FCP is orthogonal to this design and can be plugged into it as a read indicator without memory barriers in reader side.

Percpu-rwlock. Linux community is redesigning a new rwlock, called percpu rwlock [5]. Although, like FCP, it avoids unnecessary atomic instructions and memory barriers, its writer requires RCU-based quiescence detection and can only be granted after at least one grace period, where all cores have done a mode/context switch. Hence, according to our evaluation (Section 8), it performs poorly when the ratio of writers over readers is non-negligible (e.g., 1/15 for psearchy), and thus can only be used in the case of having extremely rare writers.

Read-mostly lock. From version 7.0, the FreeBSD kernel includes a new rwlock named reader-mostly lock (rmlock). Its readers enqueue special tracker structures into per-cpu queues. A writer lock is acquired by instructing all cores to move local tracker structures to a centralized queue via IPI, then waiting for all the corresponding readers to exit. Like FCP, it eliminates memory barriers in reader fast paths. Yet, its reader fast path is much longer compared to FCP, resulting in inferior reader throughput. Moreover, as IPIs always need to be broadcasted to all cores, and ongoing readers may contend on the shared queue, its writer lock acquisition is heavyweight (Section 8.2.2). In contrast, FCP leverages bounded staleness of memory consistency to avoid IPIs in the common case.

2.2 Consensus in Read-Copy Update

RCU [27] increases concurrency by relaxing the semantics of locking. Writers are still serialized using a mutex lock, but readers can proceed without any lock. RCU delays freeing memory until there is no reader referencing the object.

Typically, RCU implements consensus by either using scheduler-based or epoch-based quiescence detection that leverage context or mode switches. In contrast, the quiescence detection (or consensus) mechanism in FCP does not rely on context or mode switches and is thus faster due to its proactive nature.

RCU's relaxed semantics essentially break the all-or-nothing atomicity in reading and writing a shared object. Hence, it also places several constraints on RCU-compliant data structures, including single-pointer update and readers can only observe a pointer once (i.e., non-repeatable read). This constrains data structure design and complicates programming, since programmers must handle races and stale data and cannot always rely on cross-data-structure invariants. For example, a recent effort in applying RCU to page fault handling shows that several subtle races need to be handled manually [10], which make it very complex and resource-intensive [11]. In contrast, though the rwlock of FCP can slightly restrict parallelism by preventing readers from proceeding concurrently with a single writer, it still preserves the clear semantics of rwlocks. Hence, it is trivial to completely integrate the rwlock of FCP into complex subsystems, such as address space management.

There are also efforts in providing user-level RCU [14]. FCP also has a user-level version as an alternative to rwlock where strong semantics must be preserved with good performance, as evaluated in Section 8. Further, we show that the version-based consensus protocol using bounded

TABLE 2
IPI Latency in Different Machines

	IPI Latency (Cycles)	StdDev
AMD 64Core (Opteron 6274 * 4)	1316.3	171.4
Intel 40Core (Xeon E7-4850 *4)	1447.3	205.8

staleness can further improve the performance of a user-level RCU library.

2.3 Consensus in Safe Memory Reclamation

While RCU can be naturally used as a safe memory reclamation techniques, there have also been a set of other approaches. Hart et al. [20] made a comprehensive comparison of a set of schemes, including quiescent-state-based reclamation (i.e., RCU [27]), epoch-based reclamation [19], lock-free reference counting [30], [35] and hazard-pointer-based reclamation [29]. Their analysis (Section 5) shows that the greatest source of the performance differences is the number of atomic instructions used.

The key issue in safe memory reclamation is how to determine if a variable is still in use by other threads in dynamic-sized lock-free data structures [21], [29]. Herlihy et al. [21] model the problem as the repeat offender problem and describe a general algorithm called "Pass the Buck", which uses a set of lightweight guards to identify the status of dynamic variables (like readers in FCP) and a heavyweight wait-free *Liberate* thread to reclaims the variables. Another concurrent work, hazard pointers [29], share some similarity with the "Pass the Buck" algorithm for safe memory reclamation for dynamic-size lock-free data structures. Unlike the use of guard variables in [21], the work in [29] use a linked list of hazard pointers, which is updated by readers and scanned by the reclaimer to determine if a pointer is safe to be reclaimed.

Compared to FCP (whose RCU version can be similarly applied for safe memory reclamation), both the reclaimer side of "Pass the Buck" or hazard pointers are wait-free, but require more heavyweight read-side critical sections like more instructions to execute and requiring memory barriers and/or atomic instructions in the read side. Thus, they represent a different set of tradeoffs in the context of memory reclamation.

3 Consensus Using Bounded Staleness

3.1 Design Rationale

The essential design goal of reader-writer consensus is that readers should proceed concurrently, and thus should not share anything with each other. Hence, a scalable consensus design should require no shared state among readers and no explicit or implicit memory barriers when there are no writers pending (in the case of rwlock). However, typical rwlocks rely on atomic instructions to coordinate readers and writers. On many processors, an atomic instruction implies a memory barrier, which prevents reordering of memory operations across critical section boundary and Intel processors even completely drain the store buffer upon a memory barrier. In this way, readers are guaranteed to see the newest version of data written by the last writer. However, such memory barriers are unnecessary when no writer is present, as there is no memory ordering dependency among readers. Such unnecessary memory barriers may cause notable overhead for short reader critical sections. For example, a recent measurement shows that the cost of a memory barrier on a 4-core Intel processor ranges from 20-200 cycles, depending on the number of pending entries in the store buffer [16].

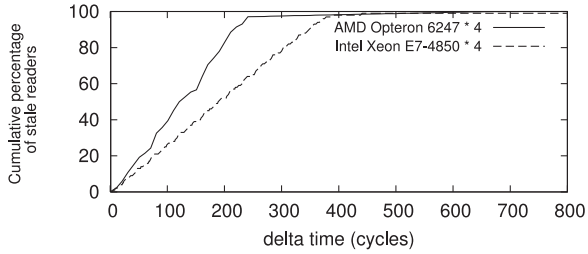


Fig. 1. Cumulative percentage of stale readers.

Message passing is not prohibitively expensive. Commodity multicore processors resemble distributed systems [4] in that each core has its own memory hierarchy. Each core communicates with others using message passing in essence, but hardware designers add an abstraction (i.e., cache coherence) to emulate a shared memory interface. Such an abstraction usually comes at a cost: due to serialization of coherence messages, sharing contended cache lines is usually costly (up to 4,000 cycles for a cache line read on a 48-core machine [6], [7]). In such contended cases, the cost may even exceed explicit message passing like inter-processor interrupts (IPIs). Table 2 illustrates the pairwise IPI latency on two recent large SMP systems, which is 1,316 and 1,447 cycles accordingly. This latency is not prohibitively expensive to be used in rwlocks, whose writer latency may exceed several tens of thousands of cycles under load.

Further, delivering multiple IPIs to different cores can be parallelized so that the cost of parallel IPI is "indistinguishable" from point-to-point interrupt [31]. This may be because point-to-point cache line movement may involve multiple cores depending on the cache line state, while an IPI is a simple point-to-point message.

Bounded staleness without memory barriers. In an rwlock, a writer needs to achieve consensus among all its readers to acquire the lock. Hence, a writer must let all readers see its current status to proceed. Typical rwlocks either use an explicit memory barrier or wait for a barrier [5] to make sure the version updates in the reader/writer are visible to each other in order. However, we argue that these are too pessimistic in either requiring costly memory barriers that limit read-side scalability or in significantly extending the writer latency (e.g., waiting for a grace period in percpu rwlock [5]).

We observe that in commodity processors such as x86 (-64), multiple memory updates can usually be visible to other cores in a very short time. We use a micro-benchmark to repeatedly write a memory location and read the location on another core after a random delay. We then collect the intervals of readers that see the stale value. Fig. 1 shows the cumulative percentage of stale readers along with time; most readers can see the writer's update in a very short time (i.e., less than 400 cycles). This is because a processor will actively flush its store buffer due to its limited size. It is

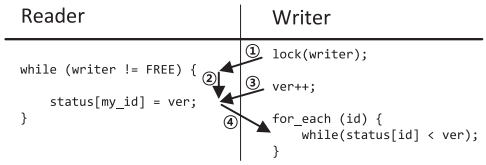


Fig. 2. Simple reader-writer lock with version report.

reasonable to simply wait a small amount of time until a reader sees the updated version for the common case, while using a slightly heavyweight mechanism to guarantee correctness.

Memory barrier not essential for mutual exclusion. To reduce processor pipeline stalls caused by memory accesses or other time-consuming operations, modern processors execute instructions out of order and incorporate store buffer to allow the processor to continually execute after write cache misses. This leads to weaker memory consistency. To achieve correct mutual exclusion, expensive synchronization mechanisms like memory barriers are often used. This may cause notable performance overhead for short critical sections.

Attiya et al., proved that it is impossible to build an algorithm that satisfies mutual exclusion, is deadlock-free, and avoids both atomic instructions and memory barriers (which avoid read-after-write anomalies) in all executions on TSO machines [3]. Although FCP readers never contain explicit memory barriers, and thus might appear to violate this "law of order", FCP uses IPIs to serialize reader execution with respect to writers for straggling readers, and IPI handling has the same effect as a memory barrier.

3.2 Basic Design

Consensus using bounded staleness. FCP introduces a 64-bit version variable (ver) to the lock structure. Each writer increases the version and waits until all readers see this version. As shown in Fig. 2, *ver* creates a series of *happens-before* dependencies between readers and writers. Suppose a writer sees the status of a reader equaling to its version, and then dependency 4 is established because the only one who modifies the status variable is the reader itself. Since the reader updates its status to the writer's version, the reader must see the modification of the version, which means dependency 3 is established. The dependency 2 is established because the modification of the status depends on the condition becomes true. Since the condition becoming true, the reader must see the lock operation of the writer and thus dependency 1 is established. A writer can only proceed after all readers have seen its new version. This ensures the correct rwlock semantic on a machine with total-store order (TSO) consistency, since a certain memory store can be visible only after all previous memory operations are visible.

However, there are still several issues with such an approach. First, a writer may never be able to enter the write-side critical section if a supposed reader never enters the read-side critical section again. Second, a reader may migrate from one core to another core so that the departing core may not be updated. Hence, such an approach may lead to arbitrarily lengthy latency or even starvation in the write side.

Handling straggling readers. To address the above issues, FCP introduces a message-based consensus protocol to let

```
Function ReadLock(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 st.reader ← PASSIVE;
3 while lock.writer ≠ FREE do
4 |   st.reader ← FREE;
5 |   st.version ← lock.version;
6 |   WaitUntil(lock.writer == FREE);
7 |   st ← PerCorePtr(lock.rstatus, CoreID);
8 |   st.reader ← PASSIVE;
9 /* Barrier needed here on non-TSO architecture */;
```

```
Function ReadUnlock(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 if st.reader = PASSIVE then
3 |   st.reader ← FREE;
4 else
5 |   AtomicDec(lock.active);
6 /* Barrier needed here on non-TSO architecture */;
7 st.version ← lock.version;
```

```
Function ScheduleOut(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 if st.reader = PASSIVE then
3 |   AtomicInc(lock.active);
4 |   st.reader ← FREE;
5 st.version ← lock.version;
```

Fig. 3. Pseudocode of reader algorithms.

the writer actively send consensus requests to readers when necessary. The design is motivated by the modest cost for message passing in contemporary processors. Hence, FCP uses IPIs to request straggling readers to immediately report their status.

This design solves the straggling reader problem. However, if a reader is allowed to sleep in a read-side critical section, a sleeping reader may miss the consensus request so that a writer may be blocked infinitely.

Supporting sleeping readers. To address the sleeping reader issue, FCP uses a hybrid design by combining the above mechanism with traditional counter-based rwlocks. FCP tracks two types of readers: passive and active ones. A reader starts as a passive one and does not synchronize with others, and thus requires no memory barriers. A passive reader will be converted into an active one upon sleeping. A shared counter is increased during this conversion. The counter is later decreased after an active reader released its lock. Like traditional rwlocks, the writer uses this counter to decide if there is any active reader.

We assume that sleeping in a reader-side critical section will be rare. Hence, FCP enjoys good performance in the common case, yet still preserves correctness in a rare case where there are sleeping readers.

4 Scalable Reader-Writer Lock

To demonstrate the usefulness of the fast consensus protocol using bounded staleness, we have implemented a scalable reader-writer lock, namely passive reader-writer lock (or prwlock) for both Linux kernel and user-level applications. We term the rwlock passive because prwlock only passively maintains reader's status to be queried by the writer.

4.1 Prwlock Algorithms

Figs. 3 and 4 show a skeleton of the read-side and write-side algorithms of prwlock. For exposition simplicity, we assume that there is only one lock and preemption is disabled within these functions so that they can use per-cpu

Function WriteLock(lock)

```
1  lastState ← Lock (lock.writer) ;
2  if lastState = PASS then
3  |   return;
4  |   /* Lock passed from another writer */
5  newVersion ← AtomicInc(lock.version);
6  coresWait ← ∅;
7  for ID ∈ AllCores do
8  |   if Online (lock.domain, ID) ∧ ID ≠ CoreID then
9  |   |   if PerCorePtr (lock.rstatus, CoreID) .version ≠ newVersion then
10 |   |   |   AskForReport (ID) ;
11 |   |   |   Add (ID, coresWait) ;
12 for ID ∈ coresWait do
13 |   while PerCorePtr (lock.rstatus, ID) .version ≠ newVersion do
14 |   |   Relax () ;
15 while lock.active ≠ 0 do
16 |   Schedule () ;
```

Function WriteUnlock(lock)

```
1  if SomeoneWaiting (lock.writer) then
2  |   Unlock (lock.writer, PASS) ;
3  else
4  |   Unlock (lock.writer, FREE) ;
```

Function Report(lock)

```
1  st ← PerCorePtr (lock.rstatus, CoreID) ;
2  if st.reader ≠ PASSIVE then
3  |   st.version ← lock.version;
```

Fig. 4. Pseudocode of writer algorithms.

states safely. In Section 7.1, we further show how to support multiple locks.

Read-side algorithm. Passive readers are tracked in a distributed way by a per-core reader status structure (*st*), which remembers the newest seen version and the passive status of a prwlock on each core. A reader should first set its status to *PASSIVE* before checking the writer lock, or there would be a time window at which the reader has already seen that the writer lock is free but has not yet acquired the reader lock. If the consensus messages (e.g., IPI) were delivered in this time window, the writer could also successfully acquire the lock and enter the critical section, which would violate the semantic of rwlock. If the reader found that this lock is writer locked, it should set its status back to *FREE*, wait until the writer unlocks and try again (lines 4-8).

Depending on the expected writer duration, prwlock could either choose to spin on the writer status, or put the current thread to sleep. In the latter case, reader performance largely depends on the sleep/wakeup mechanism (Section 6).

If a reader is holding a lock in *passive* mode while being scheduled out, the lock should be converted into an active one by increasing the active counter (*ScheduleOut*). To unlock a reader lock, one just needs to check whether the lock is held in passive mode and unlock it accordingly (*ReadUnlock*).

Hence, no atomic instructions/memory barriers are necessary in reader common paths on TSO architectures. Moreover, readers do not communicate with each other as long as they remain PASSIVE, thus guaranteeing perfect reader scalability and low reader latency.

Write-side algorithm. Writer lock acquisition can be divided into two phases. A writer first locks the writer mutex and increases the version to enter phase 1 (lines 6-20). Then it checks all online cores in the current domain to see if the core has already seen the latest version. If so, it means that reader is aware of the writer's intention, and

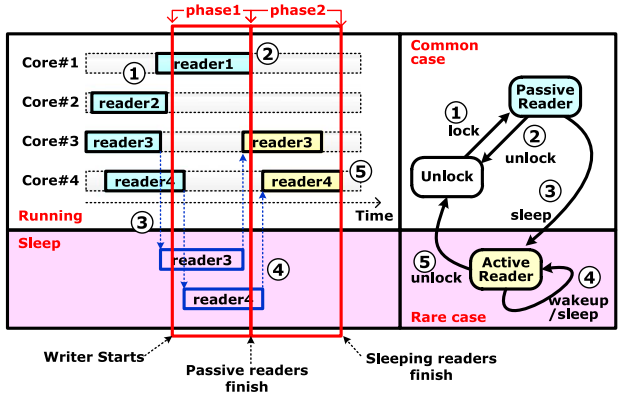


Fig. 5. An example execution of readers (left) and the state machine of reader (right). Writer is not shown here.

will not acquire reader lock until the writer releases the lock. For cores not seeing the newest version, the writer sends an IPI and asks for its status. Upon receiving an IPI, an unlocked reader will report to the writer by updating its local version (*Report*). A locked reader will report later after it leaves the read-side critical section or falls asleep. After all cores have reported, the consensus is done among all passive readers. The writer then enters phase 2 (lines 21-23). In this phase, the writer simply waits until all active readers exit. For a writer-preference lock, a writer can directly pass the lock to a pending writer, without achieving a consensus again (lines 1-2 in *WriteUnlock* and lines 2-4 in *WriteLock*).

Example. The right part of Fig. 5 shows the state machine for prwlock in the reader side. A reader in passive mode may switch to the active mode if the reader goes to sleep. It cannot be directly switched back to passive mode until the reader releases the lock. The following acquisition of the lock will be in passive mode again.

The left part of Fig. 5 shows an example execution of readers and how the consensus is done. Before a writer starts to acquire the lock, reader2 has finished its read critical section, while reader3 sleeps in its read critical section due to waiting for a certain event. Reader1 and reader4 have just started their read critical sections but have not finished yet.

In phase 1, there is a writer trying to acquire the lock in write mode, which will increase the lock version and block all upcoming readers. It will send IPIs to current active readers that have not seen the newest lock version. If reader2 in core2 has done a context switch and another thread is running right now, no IPI is required for core2. Reader4 in core4 may go to sleep to wait for a certain event, which will switch to be an active reader. No IPI is required for core4 as there is no reader in core4 at that time. At the end of phase1, all passive readers have left the critical sections. Thus, in phase 2, the writer waits for all active readers to finish their execution and finally the lock can be granted in write mode. For a writer-preference prwlock, the writer can directly pass the lock to next writer, which can avoid unnecessary consensus among readers for consecutive writers.

Correctness on TSO architecture. The main difference between rwlocks and other weaker synchronization primitives is that rwlocks enforce a strong visibility guarantee between readers and writers. This is guaranteed in prwlock with the help of TSO consistency model.

```
Function ReadUnlock(lock) for user-level prwlock
1 st ← PerThreadPtr(lock.rstatus);
2 st.reader ← FREE;
3 if st.preempted then
4 |   AtomicDec(lock.active);
5 |   st.preempted ← FALSE;
6 st.version ← lock.version;

Function ScheduleOut(lock)
1 st ← PerThreadPtr(lock.rstatus);
2 if st.reader = PASSIVE then
3 |   AtomicInc(lock.active);
4 |   st.preempted ← TRUE;
5 |   st.reader ← FREE;
6 st.version ← lock.version;
```

Fig. 6. Pseudocode of unlock algorithm with preemption detection.

Once a reader sees a FREE prwlock, we can be sure that: 1) That FREE was set by the immediate previous writer, as writers will always ensure all reader see its LOCKED status before continuing; 2) As memory writes become visible in order under TSO architectures, updates made by the previous writer should also be visible to that reader. The same thing goes with earlier writers; 3) A writer must wait until all readers to see it, so no further writers can enter critical section before this reader exits. Thus, prwlock ensures a consistent view of shared states.

These three properties together guarantee that a reader should always see the newest consistent version of shared data protected by prwlock. Moreover, as all readers explicitly report the newest version during writer lock acquisition, writers are also guaranteed to see all the updates (if any) made by readers to other data structures.

On non-TSO architectures, two additional memory barriers are required in reader algorithm as marked in Fig. 3. The first one ensures that readers can see the newest version of shared data after acquiring the lock in the fast path. The second one makes readers' memory updates visible to the writer before releasing reader locks.

4.2 User-Level Prwlock

While it is straightforward to integrate prwlock in the kernel, there are several challenges to implementing it in user space. The major obstacle is that we cannot disable preemption during lock acquisition at user space. That is to say, we can no longer use any per-core data structure, which makes the algorithm in Fig. 3 impossible.

To solve this problem, prwlock instead relies on some kernel support to maintain per-thread information for readers and performance quiescence detection for writers. The idea behind is simple: prwlock registers a user-level buffer to the per-thread kernel struct (e.g., *task_struct* in Linux kernel) to coordinate synchronization state.

Instead of using a per-core data structure to maintain passive reader status, we introduce a per-thread data structure in user space. Each thread should register an instance of it to the kernel before performing lock operations, since there is only one thread running on each core at any time. Such per-thread data structures resemble a per-core data structure used in the kernel algorithm.

For performance considerations, the reader critical paths should be entirely in user space, or the syscall overhead would ruin prwlock's advantage of short latency. As a user application may be preempted at any time, our reader lock may experience several TOCTTOU problems. Recall that in prwlock a passive reader is tracked using per-core status while active readers are tracked by the shared counter; checking and changing the passive lock mode should be done atomically.

```
Function RCUReadLock(rcu)
1 st ← PerCorePtr(rcu.rstatus, CoreID);
2 st.reader ← INFLIGHT;
3 /* Barrier needed here on non-TSO architecture */;

Function RCUReadUnlock(rcu)
1 st ← PerCorePtr(rcu.rstatus, CoreID);
2 if likely(st.reader = INFLIGHT) then
3 |   st.reader ← FREE;
4 else
5 |   AtomicDec(rcu.active);
6 /* Barrier needed here on non-TSO architecture */;
7 old_ver ← st.version;
8 new_ver ← rcu.version;
9 if unlikely(old_ver < new_ver) then
10 |   AtomicCAS (st.version, old_ver, new_ver);

Function ScheduleOut(lock)
1 st ← PerCorePtr(rcu.rstatus, CoreID);
2 if st.reader = INFLIGHT then
3 |   AtomicInc(rcu.active);
4 |   st.reader ← FREE;
5 old_ver ← st.version;
6 new_ver ← rcu.version;
7 if st.reader = INFLIGHT then
8 |   AtomicCAS (st.version, old_ver, new_ver)
```

Fig. 7. Pseudocode of reader algorithms in FCP-RCU.

For example, lines 2-3 of ReadUnlock algorithm in Fig. 6 check if a reader is a passive one, and if so, release the passive lock by setting status to FREE. If the thread is preempted between lines 2 and 3, the lock might be converted into an active lock and the active count is increased. When it is later scheduled, the active count will not be decreased since the decision has already been made before. As a result, the rwlock becomes unpaired and a writer can never acquire the lock again.

To overcome this problem, we add a preemption detection field into the per-thread data structure. As is shown in Fig. 6, the reader first sets the status to PASSIVE and checks if it has been preempted while locking passively. If so, it decreases the active counter since the lock is now an active lock.

For the write-side algorithm, since it is not possible to send IPIs in user space, almost all writers should enter kernel to acquire the lock. Fortunately, the mode switch cost between kernel and user space (around 300 cycles) is typically negligible compared to writer lock acquisition time (usually more than 10,000 cycles under load).

5 Read-Copy Update

We have also applied the fast consensus protocol to read-copy update (RCU), to implement fast quiescence detection in the writer side to reduce the latency of RCU and accelerate the reclamation of stale objects.

5.1 RCU Algorithms

Figs. 7 and 8 show a skeleton of the read-side algorithms of FCP-RCU as well as how to detect a quiescent state in RCU (i.e., *SynchronizedRCU*).

```
Function SynchronizeRCU(rcu)
1  newVersion ← AtomicInc (rcu.version);
2  Lock (rcu.writer);
3  coresWait ← 0;
4  for ID ∈ AllCores do
5  |   if Online (lock.domain, ID) ∧ ID ≠ CoreID then
6  |   |   if PerCorePtr (rcu.rstatus, CoreID) .version ≠ newVersion then
7  |   |   |   AskForReport (ID);
8  |   |   |   Add (ID, coresWait);
9  for ID ∈ coresWait do
10 |   while PerCorePtr (rcu.rstatus, ID) .version ≠ newVersion do
11 |   |   Relax();
12 while lock.active ≠ 0 do
13 |   Schedule();
14 Unlock (rcu.writer);

Function Report(lock)
1  st ← PerCorePtr (rcu.rstatus, CoreID);
2  if st.reader ≠ INFLIGHT then
3  |   st.version ← lock.version;
```

Fig. 8. Pseudocode of SynchronizeRCU algorithms.

Read-side algorithm. Similar to prwlock designs, RCU readers are similarly tracked with a per-core reader status structure (*st*). Unlike prwlock, *RCUReadLock* only needs to set the local status as *INFLIGHT*, which indicates that a reader is in progress, instead of checking whether a writer is in progress or not. Under *RCUReadUnlock*, if the CPU context is the same for the RCU reader, which is the common case (lines 2 and 3 of *RCUReadUnlock*), the reader simply restores the reader's status as *FREE*. Otherwise, the original CPU has been switched out and thus we should mark this RCU reader as an active one and track it using the per-RCU lock's active reader counter (line 5). Then, the reader snapshots the current per-core version and the current RCU's global version (lines 7 and 8); it then checks if the version has been changed afterwards. If so (which is unlikely), the reader atomically snapshots the global version to the per-core local version so that a writer can use this to determine if this reader is aware of the inflight writer.

In summary, FCP-RCU's read-side critical section contains no memory barrier and atomic instruction in the common path and the critical section itself only comprises a few number of instructions. Hence, it has high critical section efficiency.

Similarly, the *ScheduleOut* function is invoked when a core running a passive reader experienced a context switch to another core. In this case, the core to be scheduled out needs to invoke the *ScheduleOut* function to increase the active RCU reader and reset the CPU core's status as *FREE*. It then snapshots the RCU's global version if such a version has been increased.

Synchronize RCU. The key function to detect a quiescent state for RCU is the *SynchronizeRCU* function. Prior RCU implementations usually rely on scheduler-based invariant, by which each core having experienced a context or ring switch means that all CPU cores have dropped the references to stale objects. However, with the pervasive usage of dynamic ticks and increasing number of cores, this approach would suffer from extended latency, or more complexity, or both.

FCP-RCU leverages the fast quiescence detection of FCP to implement a fast quiescence detection algorithm, as shown in Fig. 8. Specifically, the *SynchronizeRCU* function first advances the global version of an RCU lock and then acquires the RCU's writer lock. It then checks if there is any straggling CPU cores not seeing the new version (lines 4-5). Due to the bounded staleness properties of TSO architectures, most CPU cores should have seen the new version. For those straggling readers, an explicit request is issued to ask this core to report its status (line 6) and add the core ID to the *coresWait* (line 7). Note that one can group multiple IPIs together to send them in a batch. For exposition clarity, we did not show such a case here. For straggling readers in *coresWait*, we then check again and wait until the reader has seen the new version. Finally, we check if there is any active RCU readers pending and wait until they have finished execution (lines 11-12) and unlocks the RCU writer lock.

Generalization. Note that, for exposition clarity, we only show a single RCU lock here. Similar to the design and implementation of RCU in Linux kernel, one can also group multiple RCUs to use one quiescence detection round. This helps a lot in reducing the consensus overhead and the resulted number of IPIs.

Correctness argument. The key to the correctness of quiescence detection in an RCU implementation is that a writer (or a reclaimer) can only reclaim an object after all RCU readers have dropped references to that object. FCP-RCU guarantees this through the global version increased by an RCU synchronizer and the happen-before relationship enforced by FCP.

Specifically, the RCU synchronizer first increases the *rcu.version* (line 1 in the SynchronizeRCU function), which essentially creates a new epoch for the RCU. If all RCU readers have seen and snapshot this new version, this indicates that all RCU readers have entered into the new epoch and thus have dropped the old references of any stale objects. This is true because its version is a snapshot from the *rcu.version* when invoking *RCUReadUnlock*, which naturally delimitates the references to RCU objects. Besides, even if an RCU reader falls to sleep, FCP-RCU still leverages a counter to count such active readers to finish. Hence, FCP-RCU can safely reclaim stale objects without causing dangling references.

5.2 User-Level RCU

User-level RCU can be implemented by simply replacing per-core states with per-thread states and using signals to call for report. However, this solution does not achieve short latency if the threads number is much more than the cores number. To achieve better performance, we add a module in OS to provide services for user-level RCU. In this module, we pre-allocate an RCU array in the kernel and add a syscall that cooperates with user-level RCU. Before a program uses user-level RCU, it invokes a syscall with *REG* flag to allocate a corresponding RCU data structure from the pre-allocated array and return its identifier. The following operations like locking and unlocking work using the identifier. To track the readers that are preempted and become active, reader threads invoke the added syscall with a *READREG* flag to add *preempt_notifiers* in their *task_structs*. Because read-side is lightweight, operations of readers should not invoke syscalls. To ensure that readers are visible by *preempt_notifiers*, we add a user-space address pointer in *task_struct* which is set up during the syscall with a *READREG* flag. The address

is filled with the corresponding RCU identifier in *RCUReadLock* and is reset to *EMPTY* in *RCUReadUnlock*.

Because programs cannot disable preemption in user-level, we add a preemption flag to detect whether readers are preempted before completing unlocking. To guarantee correctness, a compiler barrier is necessary between resetting the address and check the preempted flag. *SynchronizeRCU* is protected by a mutex and invokes the syscall with a *SYNC* flag. The syscall with a *SYNC* flag increases the global version and waiting for all cores' reporting. Because the report needs to work in kernel-level, the syscall overhead may decrease the performance of readers. Of course, we can allow the user-level program to operate the local version directly by locating the local versions in a shared page, but it is dangerous because the kernel control path flow depends on them. Hence, our solution is that readers do not report frequently but at regular intervals or only report after receiving IPIs.

5.3 Performance Analysis

After describing how rwlock and RCU can be implemented based on FCP, this section presents a performance analysis of both implementations.

Memory barrier. In the common path of read-side critical sections of both prwlock and FCP-RCU, FCP requires no memory barrier in the common path, i.e., when there is no outstanding writer or SynchronizeRCU. The only memory barrier required is when a CPU core is about to leave a lock domain, e.g., switch to another task and make current lock domain offline or online, or when another core is trying to reclaim stale objects by invoking SynchronizeRCU. However, such operations are relatively rare in typical execution. Hence, FCP enjoys good performance scalability in common cases.

Writer cost. It appears that using IPIs may significantly increase the cost of writes, due to the IPI cost, possible mode switches and disturbed reader execution. However, thanks to the properties of bounded staleness, the straggling readers are usually very few. However, the cost of IPIs and mode switches are small and usually in the scale of several hundred to one thousand cycles. Further, as a writer usually needs to wait for a while until all readers have left the critical section, such costs can be mostly hidden. Though there may be a few cold cache misses due to disturbing reader execution, such misses on uncontended cache lines would be much smaller than the contention on shared states between readers and writers in traditional rwlocks.

Besides, there is one interesting property for prwlock. In contrast to traditional rwlocks, the more readers are currently executing in the read-side critical section, the faster that a write can finish the consensus and get the lock in write mode (Section 8.2.2). This is because readers will likely see the writer, and thus report immediately. Such a feature fits well with the common usage of rwlocks (more readers than writers).

Space overhead. Since FCP is essentially a distributed consensus protocol, it needs $O(n)$ space for a lock instance. More specifically, the current implementation needs 12 bytes (eight for version and four for reader status) per core per lock to maximize performance. It is also possible to pack a 7 bit version and a 1 bit status into one byte to save space. Another several bytes are needed to store writer status, whose exact size depends on the specific writer synchronization mechanism used. Further, an additional 1 byte per core is needed to store domain online status to support the lock domain abstraction.

By using the Linux kernel's per-cpu storage mechanism, a lock's per-cpu status could be packed into the same cache line as other per-cpu status words. Compared with other scalable rwlock algorithms (e.g. brlock, SNZI rwlock, read-mostly lock), FCP imposes similar or lower space overhead.

Applicability. FCP is not a panacea to read-mostly synchronization and performs better for a relatively short reader critical section where contention on shared data determines performance scalability. For extremely long read-side critical sections, FCP would perform similarly with traditional read-mostly synchronization as the cost of message-based consensus will be much smaller than the cost of the critical section itself. Different with brlock and many other scalable reader writer lock algorithms, prwlock is essentially a write-preference sleepable rwlock that tries to provide small write latency, which fits well with many kernel synchronization scenarios. Finally, FCP prefers machines and operating systems with fast inter-core messages so that the consensus can be done more quickly.

Memory consistency model requirement. As FCP relies on a series of happened-before relationships on memory operations, it requires that memory store operations are executed and become visible to others in the issuing order (TSO consistency). Fortunately, this assumption holds for many commodity processor architectures like x86(64), SPARC and zSeries.

6 Decentralized Parallel Wakeup

Many read-mostly synchronization mechanisms need to cope with complex OS semantics like sleeping inside a reader-side critical section. This requires the OS to wake up such straggling readers when certain conditions have been satisfied. On a large-scale multicore machine, our study shows that existing wakeup mechanism may spend quite a large amount of time to wake readers. To this end, this section describes a decentralized parallel wakeup mechanism to optimize the performance of both traditional synchronization mechanisms as well as those enabled by FCP.

Issues with centralized sequential wakeup. Sleep/wakeup is a common OS mechanism that allows a task to temporarily sleep to wait until a certain event happens (e.g., an I/O event). Operating systems such as Linux, FreeBSD and Solaris use a shared queue to hold all waiting tasks. It is usually the responsibility of the signaling task to wake up all waiting tasks. To do this, the signaling task first dequeues the task from the shared task queue, and then does something to prepare waking up the task. Next, the scheduler chooses a core for the task and inserts the task to the percpu runqueue. Finally, the scheduler sends a rescheduling IPI to the target core so that the awakened task may get a chance to be scheduled. The kernel will repeat sending IPIs until all awakened tasks have been rescheduled.

There are several issues with such a centralized, sequential wakeup mechanism. First, the shared waiting queue may become a bottleneck as multiple cores trying to sleep may contend on the queue. Hence, our first step involves using a lock-free wakeup queue so that the lock contention can be mitigated. However, this only marginally improves performance.

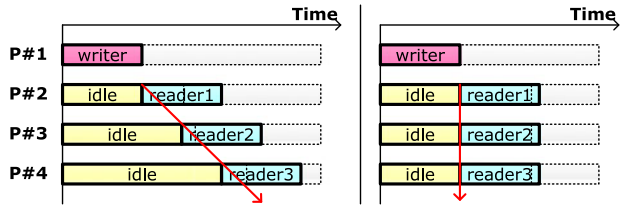


Fig. 9. Issue with centralized, sequential wakeup (left) and how decentralized parallel wakeup solve this problem (right).

Our further investigation uncovers that the main performance scalability issue comes from the cascading wakeup phenomenon, as shown in Fig. 9. When a writer leaves its write-side critical section, it needs to wake up all readers waiting for it. As there are multiple readers sleeping for the writer, the writer wakes up all readers sequentially. Hence, the waiting time grows linearly with the number of readers.

Decentralized parallel wakeup. To speed up this process, FCP distributes the duty of waking up tasks among cores. In general, this would risk priority inversion, but all FCP readers always have equal priority.

Fig. 10 shows the key data structure used in the decentralized parallel wakeup. Each core maintains a per-core wakeup queue (PWake-queue) to hold tasks sleeping on such a queue, each of which sleeps on a wakeup condition word. When a running task is about to sleep (step 1), it will be removed from the per-cpu runqueue and inserted into the per-cpu wakeup queue. Before entering the scheduler, if the kernel indicates that there is a pending request (e.g., by checking the wakeup counter), each core will first peek the PWake-queue to see if there is any task to wake up by checking the status word. If so, it will then insert the task to runqueue. This may add some cost to the per-cpu scheduler when there are some pending wakeup requests. However, as there are usually only very few tasks waiting in a single core, the cost should be negligible. Further, as all operations are done locally in each core, no atomic instructions and memory barriers are required. Finally, as a task generally wakes up on the core that last executed it, this task may benefit from better locality in both cache and TLBs. After checking the PWake-queue, each core will execute its scheduler (step 2) to select a task to execute (step 3).

As the new wakeup mechanism may require a core to poll the wakeup queue to reschedule wakeup tasks in the per-core scheduler, it may cause waste of power when there are no runnable tasks in a processor. To address this problem, our wakeup mechanism lets each idle core use the mwait mechanism[1] to sleep on a global word (step 4). When a writer finishes its work and signals to wake up its waiting tasks, the writer touches the word to wake up idle cores, which will then start to check if any tasks in the wakeup queue should be woken up.

The main advantage of the parallel wakeup mechanism is that it incurs very little latency to wake up a task. Further, it avoids the sequential wakeup problem so that multiple waiting tasks can be woken up to execute in parallel. The downside is that there might be some imbalance if the task distribution has changed dramatically since a task sleeps.

1. mwait/monitor are x86 instructions that setup and monitor if an memory location has been touched by other cores.

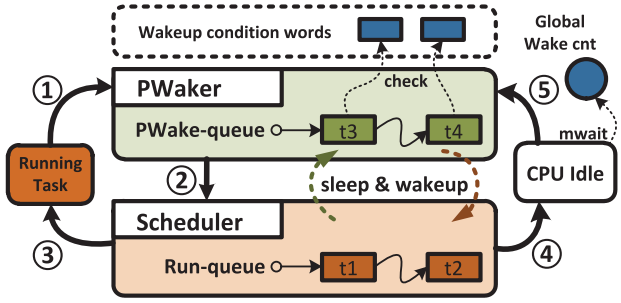


Fig. 10. Key data structure and state transition graph of decentralized parallel wakeup in each core.

This can be mitigated using the load balance mechanism in the CPU scheduler. Besides, as a core in mwait state is less energy-efficient than in a complete sleeping state like C-state, the decentralized parallel wakeup may be less energy-efficient when the tasks may need to wait a long time. In such cases, waking up from a deeper sleeping state would have higher latency than from the mwait state, which indicates that the parallel wakeup mechanism can be more quickly than the serialized wakeup mechanism, though at the cost more energy consumption during sleeping. Nevertheless, tasks using read-mostly synchronization in our test usually tend to sleep only for a short period.

Note that the decentralized parallel wakeup mechanism should be viewed as an optimization instead of a complete substitution over existing wakeup mechanisms. Since there are multiple queues in an OS kernel and the mwait mechanism can only mwait on a single address range, a set of tasks which want to enjoy the benefit of parallel wakeup should sleep on the per-core PWake-queue instead on its own queue. Other tasks can still use the original global wait queues without the benefit of parallel wakeup.

7 Implementation and Applications

7.1 OS Kernel Incorporation

While Sections 4 and 5 use a simplified way to illustrate how FCP work by assuming only one reader/writer lock per core; there are several issues in incorporating FCP to an OS kernel, where there are multiple processes, threads, and locks.

The scope of a FCP-based rwlock or RCU could be either global or process-wide and there may be multiple FCP-based locks in each scope. For example, each FCP-based lock could be shared by multiple tasks. To reduce messages between readers and writers, FCP uses the *lock domain* abstraction to group a set of related FCP-based locks that can do consensus together. A domain tracks CPU cores that are currently executing tasks related to a FCP-based locks. Currently, a domain could be process-wide or global. We now describe how FCP-based locks uses the domain abstraction:

Domain online/offline. It is possible that the scope for a set of FCP-based locks may be switched off during OS execution. For example, for a set of locks protecting the address space structure for a process, the structure may be switched off during an address space switch. In such cases, FCP uses the domain abstraction to avoid unnecessary consensus messages. A domain maintains a mapping from cores to its online/offline status. Only CPU cores within an active domain will necessitate the sending of messages. Fig. 11

```
Function DomainOnline(dom)
1 coreSt ← PerCorePtr(dom.cores, CoreID);
2 coreSt.online = TRUE;
3 MemoryBarrier();

Function DomainOffline(dom)
1 coreSt ← PerCorePtr(dom.cores, CoreID);
2 MemoryBarrier();
3 coreSt.online = FALSE;
```

Fig. 11. Domain management algorithms.

shows how to dynamically adjust the domain. The algorithm is simple as the consensus protocol can tolerant inaccurate domains.

When a domain is about to be online on a core, it simply sets the mapping and then performs a memory barrier (e.g., mfence). As the writer (or a reclaimer) always sets its status before checking domains, it is guaranteed that either a writer could see the newly online core, or incoming readers on that core can see the writer is acquiring a lock. In either case, the lock semantic is maintained. To correctly make a domain offline from a core, a memory barrier is also needed before changing the domain to ensure that all previous operations are visible to other cores before offline.

Currently, for domains that correspond to processes, FCP makes domains online/offline before and after context switches. However, it is possible to make a domain offline at any time if readers are expected to be infrequent afterward. When outside a domain, readers must acquire all FCP-based locks in the slower *ACTIVE* state. We choose to leave the choice to lock users as they may have more insight on the workload.

Using domain online/offline also naturally records CPU online and offline when CPU hotplug is supported. This may notably reduce the amount of IPIs during consensus.

Task online/offline. A task (e.g., a thread) may be context switched to other tasks and a task may also be migrated from one core to another core. FCP uses task online/offline to handle such operations. When a task is about to be switched out while holding a prwlock in PASSIVE or INFLIGHT mode, it will change its lock status to be ACTIVE and increase the active reader counter if it previously holds a FCP in passive read mode. This makes sure that a writer will wait until this task is scheduled again to leave its critical section to proceed. A task needs to do nothing when it is scheduled to be online again.

Downgrade/upgrade of rwlocks. Typical operating systems usually support downgrading an rwlock from write mode to read mode and upgrading from read mode to write mode. Prwlock similarly supports lock downgrading by setting the current task to be in read mode and then releasing the lock in write mode. Unlike traditional rwlocks, upgrading a prwlock from read mode to write mode may be more costly in a rare case when the upgrading reader is the only reader, due to the lack of exact information regarding the number of readers. To upgrade a lock from read to write mode, prwlock tries to acquire the lock in write mode in the read-side critical section, but counts one less readers (excluding the upgrading reader itself) when acquiring the lock.

7.2 Applications

Scaling address space management in Linux. Many operating systems such as Linux, FreeBSD, Solaris use a tree-like data structure to organize an address space, where page fault handlers lookup the tree to fill page tables and the mmap/munmap calls update the tree. An rwlock (e.g., *mmap_sem*) is used to allow concurrent page faults but serialize updates to updates (e.g., mmap, munmap) to an address space.

Prior research [6], [10], [33] has shown that there is a serious contention over this per-process rwlock in Linux. Unfortunately, it requires non-trivial efforts to remove or replace this contending rwlock as it is widely used in address space related operations in address space management, device drivers, file systems and process management, resulting in more than 600 references to this rwlock. Though Clements et al. [10] has attempted to scale up page fault handling by integrating a variant of sleepable RCU, they only replace *mmap_sem* for page fault handling on anonymous memory mapping and leave other parts such as memory-mapped files, copy-on-write faults, and device drivers still holding the *mmap_sem*.

As prwlock is essentially an rwlock, it can be used to replace the original rwlock straightforwardly. We write a script to replace more than 600 calls to *mmap_sem*. We add several hooks to process fork, exec, exit, wakeup and context switch. The FCP library comprises of less than 300 LoC and requires manually changing less than 30 LoC other than the automatically replaced calls to *mmap_sem*. This is significantly less than the prior effort (around 2,600 LoC for page fault handling only).

User-level prwlock and Kyoto Cabinet. We have also implemented FCP into user space, as a large number of parallel applications make use of rwlocks for synchronization. We implemented the kernel support for FCP into Linux kernel by adding a new syscall and around 500 LOCs. User applications may create/destroy or operate on FCP through the added syscall.

To show the performance advantage of user-level FCP. We modified a famous database system named Kyoto Cabinet [18], which use an rwlock to protect its data tables. We modified Kyoto Cabinet to use prwlock by replacing the rwlock. Our evaluation shows that prwlock boosts performance significantly due to the reduced contention on the reader-side critical section.

User-level FCP-RCU. The kernel support of FCP can be easily retargeted to implement our user-level RCU mechanism. It is traditionally considered non-trivial to implement RCU in user space [14]. Traditional user-level RCU either provides inefficient read-side primitives or restrict application architecture. By using the version-based consensus protocol of FCP to implement quiescence detection, our user-level RCU implementation shows better read-side throughput and faster quiescence state detection than previous algorithms (Section 8.3).

8 Evaluation

We have implemented FCP on several versions of Linux. Prwlock was initially implemented in Linux 3.2.6 and has been integrated with the Linux virtual memory system by replacing the default rwlock. Prwlock was also ported to

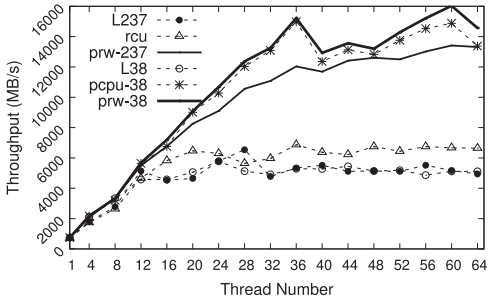


Fig. 12. Histogram throughput scalability for original Linux, percpu-rwlock, FCP on Linux 3.8.

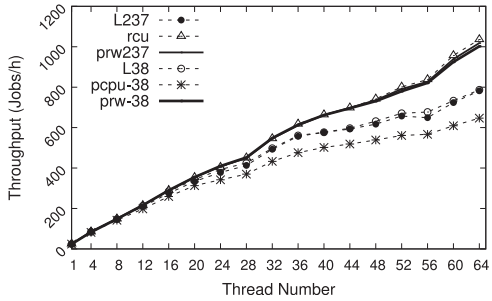


Fig. 13. Metis throughput scalability for original Linux, percpu-rwlock, FCP on Linux 3.8.

Linux 2.6.37 to compare its performance with an RCU-based virtual memory system [10], and a recent version Linux 3.8 to compare its performance with percpu reader-writer lock [5]. The porting effort among different versions of Linux is trivial and one student can usually finish it in less than one hour. To study the kernel-level FCP-RCU performance, we port a hashtable into the kernel and compare it with the default RCU implementation as well as a special RCU implementation called (RCU Expedited) which allows extremely fast quiescence detection.

8.1 Evaluation Setup

Kernel-level prwlock. We use three workloads that place intensive uses of virtual memory: Histogram [32], which is a MapReduce application that counts colors from a 16 GB bitmap file; Metis [24] from MOSBENCH [6], which computes a reverse index for a word from a 2 GB Text file residing in memory; and Psearchy [6], a parallel version of searchy that does text indexing. They represent different intensive usages of the VM system, whose ratio between write (memory mapping) and read (page fault) are small, medium and large.

Kernel-level FCP-RCU. We also implemented a concurrent hashtable [34] in kernel as a micro-benchmark to characterize FCP and its alternatives.

User-space prwlock and FCP-RCU. We use several micro-benchmarks to compare FCP with several alternatives like brlock and user-level RCU. As FCP has a user-level RCU library, we also compare its performance to traditional signal-based user space RCU [14]. To show that FCP can scale up user-space applications, we also evaluated the Kyoto Cabinet database using FCP and the original rwlock.

As the performance characteristic that FCP relies on are similar for Intel and AMD machines, we mainly run our tests on a 64-core AMD machine, which has four 2.4 GHz 16-core chips and 128 GB memory. We use Linux kernel version 3.8 as the default kernel. For each benchmark, we evaluate the throughput in a fixed time and collect the arithmetic mean of five runs.

8.2 Kernel-Level FCP

8.2.1 Application Benchmarks for Prwlock

We compare the performance of prwlock with several alternatives, including the default rwlock in Linux for virtual memory, percpu read-write lock [5], and an RCU-based VM design [10] (RCUVM). We are not able to directly compare prwlock with brlock as it has no sleeping support. As RCUVM is implemented in Linux 2.6.37, we also ported prwlock to Linux 2.6.37. We were not able to use the same version to do the experiments as porting the RCU-based VM design is difficult and resource-intensive, as acknowledged by the authors [11].

As different kernel versions have disparate mmap and page fault latency, we use the Linux 2.6.37 kernel as the baseline for comparison. For the three benchmarks, we present the performance scalability for Linux-3.8 (L38), percpu-rwlock (pcpu-38) and prwlock on Linux 3.8 (prw-38), as well Linux 2.6.37 (L237), RCUVM (rcu) and FCP on Linux 2.6.37 (prw-237) accordingly.

Histogram. As histogram is a page-fault intensive workload and the computation is very simple, it eventually hits the memory wall after 36 cores on Linux 3.8 for both percpu-rwlock and FCP, as shown in Fig. 12. Afterwards, both prwlock and percpu-rwlock show similar performance thrashing, probably due to memory bus contention. Percpu-rwlock scales similarly well and is with only a small performance gap with prwlock; this is because both have very good read-side performance. In contrast, the original Linux cannot scale beyond 12 cores due to contention on *mmap_sem*. As a result, prwlock outperforms Linux and percpu-rwlock by 2.85X and 9 percent, respectively on 64 cores.

It was quite surprising that FCP significantly outperforms RCUVM. This is because currently RCUVM only applies RCU to page fault on anonymous pages, while histogram mainly faults on a memory-mapped files. In such cases, RCUVM retries page fault with the original *mmap_sem* and thus experiences poor performance scalability. Though RCUVM can address this problem by adding RCU support for memory-mapped files, prwlock provides a much easier way to implement and reason about correctness due to its clear semantic.

Metis. Metis has relatively more mmap operations (mainly to allocate memory to store intermediate data), but is still mainly bounded by page fault handling on anonymous memory mapping. As shown in Fig. 13, prwlock performs near linearly to 64 cores with a speedup over percpu-rwlock and original Linux by 27 and 55 percent in 64 cores accordingly. This is mainly due to scalable read-side performance and small write-side latency. There is a little bit performance gap with RCUVM, as RCUVM further allows a writer to proceed in parallel with readers.

Psearchy. Psearchy has many parallel mmap operations from multiple user-level threads (with a writer/reader ratio around 1/15), which not only taxes page fault handler, but also mmap operations. Due to extended mmap latency,

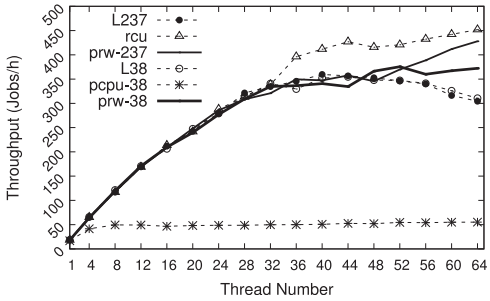


Fig. 14. Psearchy throughput scalability for original Linux, percpu-rwlock, FCP on Linux 3.8.

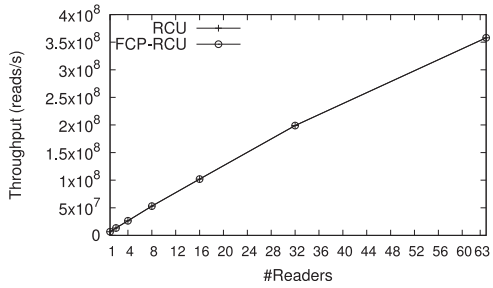


Fig. 16. Lookup performance of hashtable.

percpu-rwlock cannot scale beyond four cores, as shown in Fig. 14. In contrast, prwlock performs similarly with Linux before 32 cores and eventually outperforms Linux after 48 cores, with a speedup of 20 percent and 5.63X over Linux and percpu-rwlock for Linux 3.8. There is a performance churn between 32 and 48 cores for Linux, probably due to the contention pattern changes during this region. For Linux 2.6.37 with smaller mmap latency, prwlock performs similarly with Linux under 48 cores and begins to outperform Linux afterwards. This is due to the contention over rwlock in Linux, while prwlock's excellent read-side scalability makes it still scale up.

As psearchy is a relatively mmap-intensive workload, FCP performs worse than RCUVM as RCUVM allows readers to proceed in parallel with writers. Under 64 cores, prwlock is around 6 percent slower than RCUVM. Psearchy can be view as a worst case for prwlock and we believe this small performance gap is worthwhile for much less development effort.

Summary. It can be seen that prwlock almost consistently outperforms other rwlock designs for Linux for different ratio between write and read. It also has only a very small performance loss than RCUVM. This confirms that prwlock performs stably for different contention patterns.

8.2.2 Microscopic Analysis

Benefits of parallel wakeup. We show how the parallel wakeup mechanism could benefit both kernel operations and user code. Fig. 15 using the histogram benchmark to show how parallel wakeup can improve the performance of both RCUVM and original Linux. Parallel wakeup boosts RCUVM by 34.7 percent when there are multiple readers waiting. FCP improves the performance of original Linux by 47.6 percent. This shows that parallel wakeup can also be separately applied to Linux to improve performance. We also collected the mmap and munmap cost for both Linux and FCP, which are 934, 1014 and 567, 344us. With the fast wakeup mechanism, the cost for Linux has decreased to 697 and 354us.

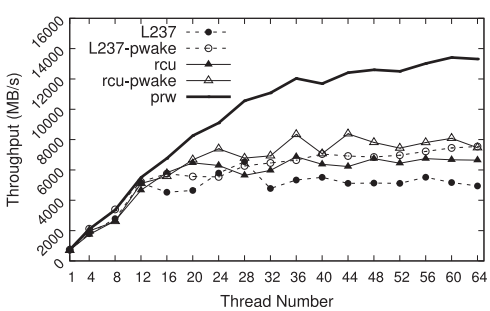


Fig. 15. Benefit of parallel wakeup for Histogram.

To further understand the benefit from the parallel wakeup, we compared the parallel wakeup mechanism with the serial wakeup using the wait queue in Linux 3.8 by measuring the latency of waking up threads. In this evaluation, 63 threads sleep on a variable and 1 thread wakes them up. As Fig. 19 demonstrates, 90 percent threads can be woken up within 30,000 cycles with the help of the parallel wakeup mechanism, which are 30x faster than using wait queue.

Critical section efficiency. To better characterize different rwlocks, we also evaluate their raw critical section overhead (lock/unlock pair latency), which is shown in Table 3. FCP shows best reader performance as its common path is simple and has no memory barriers. Thanks to the domain abstract, threads with few readers can acquire locks in active mode to reduce unnecessary IPIs. In this way, prwlock can achieve short write latency even there are no readers on other threads. Though rmlock (Read-Mostly Lock in FreeBSD) also eliminates memory barriers in reader common paths, its reader algorithm is more complex than prwlock, and thus results in higher reader latency. Writer of rwsem (Linux's rwlock) performs well for few readers, but suffers from contention with excessive readers.

Impact from sleepers. To measure the performance of prwlock with sleep readers. We distribute 64 reader threads among 64 cores and force some readers to invoke schedule functions in critical sections. To make them suffer from context switches, we also distribute 64 idle threads among the 64 cores, each of which invokes the schedule function in a loop. Fig 20 shows that prwlock can benefit a lot from our scalable design even there are 32 sleep reader threads. When all reader threads become sleepy readers, FCP still only performs similarly with rwsem.

8.2.3 Concurrent Hash Table for RCU

We use a concurrent hashtable [34] to compare FCP with traditional RCU. Fig. 16 illustrates the performance. Both FCP-RCU and traditional RCU have a nearly zero reader overhead.

By using FCP-based quiescence detection instead of scheduler-based RCU, resizing the hashtable is much simpler and faster as all readers are blocked during resizing and FCP can detect a quiescent state quickly. Fig. 17 presents average latency of synchronizeRCU used to shrink and grow the hash table on different concurrency levels. FCP shows up to three orders of magnitude shorter resizing latency compared to RCU and three times faster than the expedited one on AMD platform. Fig. 18 presents the same

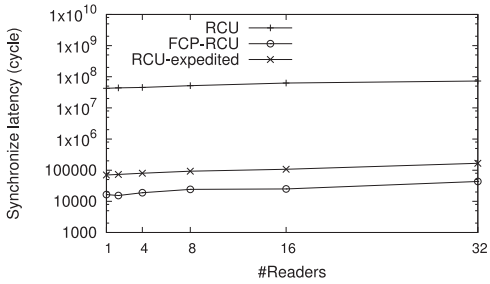


Fig. 17. SynchronizeRCU latency of hashtable (AMD).

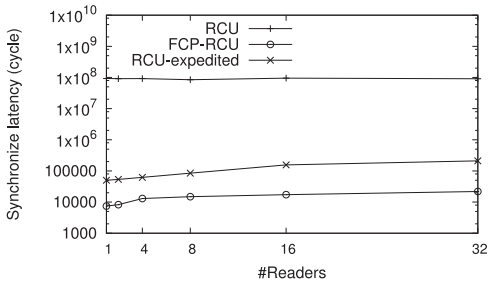


Fig. 18. SynchronizeRCU latency of hashtable (Intel).

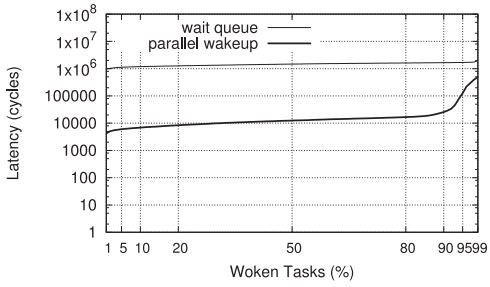


Fig. 19. Woken latency.

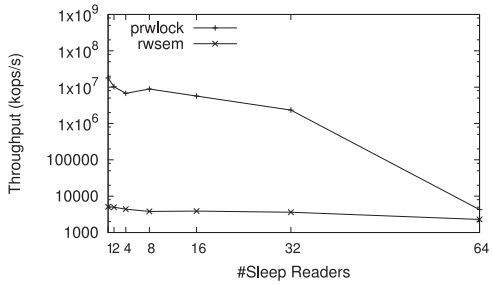


Fig. 20. Throughput with sleep readers.

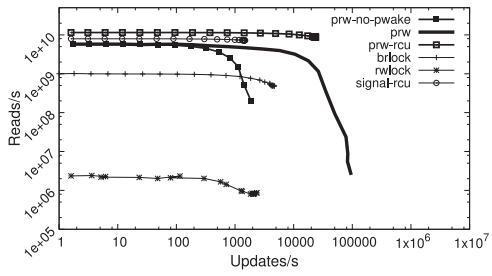


Fig. 21. Relation between reader/writer throughput.

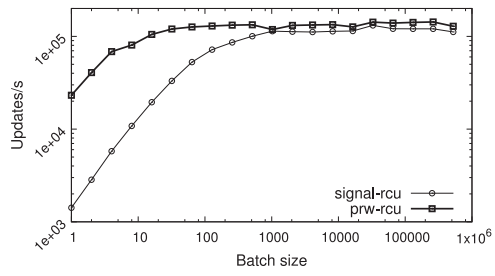


Fig. 22. Update performance with batch size.

TABLE 3
Critical Section Efficiency (Average of 10 Millions Runs)

	brlock	rmlock	rwsem	prwlock
Reader latency (1 reader)	58	46	107	12
Reader latency (64 readers)	58	46	20,730	12
Writer latency (0 reader)	17,709	136	100	1,389
Writer latency (63 readers)	89,403	622,341	3,235,736	23,236

evaluation on Intel platform. FCP also shows the best performance among three RCU Implementations.

8.3 User-Level FCP

We use several micro-benchmarks to evaluate FCP's performance, as well as the Kyoto Cabinet database to demonstrate user-level prwlock's performance advantages.

Fig. 21 shows the impact of writer frequency on reader throughput for several locking primitives, by running 63 reader threads and one writer thread. Writer frequency is controlled by varying the delay between two writes, which is similar done as Desnoyers et al. [14]. Note that one writer is the worst case of prwlock since if there is more than one writer, the writer lock could be passed among writers without redoing consensus. To compare the time for a consensus, we fixed the batch size of both RCU algorithms to 1. That means they must wait a grace period for every update.

FCP achieves the highest writer rate. This confirms that our version-based consensus protocol is more efficient than prior approaches. FCP's read side performance is similar to RCU, and notably outperforms brlock, mainly because prwlock requires no memory barriers in reader side. Parallel wakeup also contributes to prwlock's superior performance. Since it improves reader concurrency, prwlock is able to achieve higher reader throughput when there are many writers. Writer performance is also greatly improved since wakeup is offloaded to each core.

Performance of user-level FCP-RCU. We can also notice that FCP-based RCU performs consistently better than the signal-based user-level RCU. Thanks to FCP's kernel support, the reader-side algorithm of FCP-RCU is simpler, which results in a higher reader throughput. Besides, FCP-RCU has orders of magnitude higher writer rate than signal-based RCU, due to its fast consensus protocol.

We further vary the batch size to study RCU performance, as shown in Fig. 22. FCP-RCU reaches its peak performance before the batch size reaches 100 and performs much better when the batch size is less than 1,000. While typical batch size would be 1,000 to 10,000 or even larger, we believe the faster consensus of FCP allows achieving a good performance even at a small batch size. A small batch

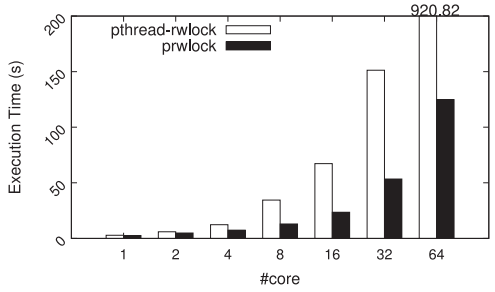


Fig. 23. Benefit of FCP for an in-memory DB.

size helps control the memory footprint since it allows faster reclamation of unused objects. Hence, it can improve memory locality of those application using RCU.

Kyoto Cabinet. Fig. 23 shows the improvement of FCP over using the original pthread-rwlock. As the workload for different number of cores is different, the increasing execution time with core does not mean poor scalability. For all cases, FCP outperforms original rwlock and the improvement increases with core count. Under 64 cores, FCP outperforms pthread-rwlock by 7.37X (124.8 s versus 920.8 s). The reason is that the workload has hundreds of millions read accesses and pthread-rwlock incurs high contention on the shared counter, while FCP places no contention in the reader-side.

8.4 Impact on Single-Threaded Applications

We compared the performance of prwlock and original Linux of building a Linux kernel 3.8 to see the performance impact of prwlock on single-threaded applications. Under a 64-core setting, prwlock and the original Linux take 64.5 and 64.4 s to finish building the kernel, which is nearly not measurable in practice. This shows that prwlock has no impact on single-threaded application performance.

9 Conclusions

This paper described FCP, a fast consensus protocol that leverages the bounded staleness of TSO architectures for read-mostly synchronization constructs. FCP provides fence-free read-side critical sections to achieve high critical section efficiency for readers, and leverages bounded staleness to check if all readers have seen the writer's (or the reclaimer's) new version. We show that FCP can be used to construct a scalable reader-writer lock as well as an RCU implementation with fast quiescence detection. Measurements on a 64-core machine confirmed its performance and scalability using a set of application benchmarks that contend kernel components as well as a database.

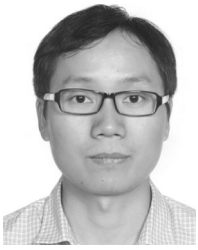
Acknowledgments

A preliminary version of this paper focused on designing scalable reader-writer locks is published in [23]. This work is supported in part by China National Natural Science Foundation (61572314), National Youth Top-notch Talent Support Program of China, and Singapore CREATE E2S2. Haibing Guan is the corresponding author.

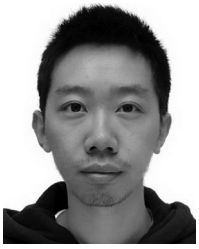
References

[1] Linux test project [Online]. Available: http://ltp.sourceforge.net/
[2] Version-based brlock [Online]. Available: https://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/include/linux/brlock.h
[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," *Proc. 38th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2011, pp. 487–498.
[4] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principle*, 2009, pp. 29–44.
[5] S. S. Bhat. (2013). [Online]. Available: https://patchwork.kernel.org/patch/2157401/
[6] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Proc. 9th USENIX Conf. Operating Syst. Design Implementation*, 2010, pp. 1–8.
[7] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proc. Linux Symp.*, 2012, pp. 119–130.
[8] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, "NUMA-aware reader-writer locks," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 157–166.
[9] B. Cantrill and J. Bonwick, "Real-world concurrency," *Queue*, vol. 6, no. 5, pp. 16–25, 2008.
[10] A. Clements, M. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 199–210.
[11] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 211–224.
[12] J. Corbet. (2010). Big reader locks [Online]. Available: http://lwn.net/Articles/378911/
[13] P. Courtois, F. Heymans, and D. Parnas, "Concurrent control with readers and writers," *Comm. ACM*, vol. 14, no. 10, pp. 667-668, 1971.
[14] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole, "User-level implementations of read-copy update," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, Feb. 2012.
[15] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 247–256.
[16] Y. Duan, A. Muzahid, and J. Torrellas, "Weefence: Toward making fences free in TSO," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 213–224.
[17] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "SNZI: Scalable nonzero indicators," in *Proc. 26th Annu. ACM Symp. Principles Distrib. Comput.*, 2007, pp. 13–22.
[18] (2015). FAL Labs. Kyoto Cabinet. Available: http://fallabs.com/kyotocabinet/
[19] K. Fraser, "Practical lock-freedom," PhD dissertation, Univ. Cambridge, Cambridge, MA, United Kingdom, 2004.
[20] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of memory reclamation for lockless synchronization," *J. Parallel Distrib. Comput.*, vol. 67, no. 12, pp. 1270–1285, 2007.
[21] M. Herlihy, V. Luchangco, and M. Moir, "The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures," in *Proc. 16th Int. Conf. Distrib. Comput.*, 2002, pp. 339–353.
[22] Y. Lev, V. Luchangco, and M. Olszewski, "Scalable reader-writer locks," in *Proc. 21st Annu. Symp. Parallelism Algorithms Archit.*, 2009, pp. 101–110.
[23] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive reader-writer locks," in *Proce. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 219–230.
[24] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for multicore architectures," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2010-020, 2010.
[25] P. Mckenney. RCU usage in the Linux kernel: One decade later. (2012) [Online]. Available: http://www2.rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf
[26] P. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *Proc. Ottawa Linux Symp.*, 2001, pp. 338–367.
[27] P. McKenney and J. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Proc, Int. Conf. Parallel Distrib. Comput. Syst.*, 1998, pp. 509–518.

[28] J. Mellor-Crummey and M. Scott, "Synchronization without contention," in *Proc. 4th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 1991, pp. 269–278.
[29] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.
[30] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," University of Rochester, Rochester, NY, USA, Technical Report 599, DTIC Document, 1995.
[31] D. Petrović, O. Shahmirzadi, T. Ropars, A. Schiper, et al., "Asynchronous broadcast on the Intel SCC using interrupts," in *Proc. Int. Many-Core Appl. Res. Community Symp.*, 2012, pp. 24–29.
[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Performance Comput. Archit.*, 2007, pp. 13–24.
[33] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, "A case for scaling applications to many-core with os clustering," in *Proc. 6th Int. Conf. Comput. Syst.*, 2011, pp. 61–76.
[34] J. Triplett, P. E. McKenney, and J. Walpole, "Resizable, scalable, concurrent hash tables via relativistic programming," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 11–11.
[35] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proc. 14th Annu. ACM Symp. Principles Distrib. Comput.*, 1995, pp. 214–222.



Haibo Chen received the PhD degree in computer science from Fudan University in 2009. He is currently a professor at the School of Software, Shanghai Jiao Tong University. His research interests include operating systems and parallel and distributed systems. He is a senior member of IEEE.



Heng Zhang is currently working toward the master's degree at the School of Software, Shanghai Jiao Tong University. His research interests include operating systems and synchronization constructs.

Ran Liu received the master's degree from the Software School of Fudan University. He was a visiting student at the Institute of Parallel and Distributed Systems, School of Software, Shanghai Jiao Tong University. He is now an engineer at NetEase, Inc. His research interests include operating systems and synchronization constructs.

Binyu Zang received the PhD degree in computer science from Fudan University, in 2000. He is currently a professor at the School of Software, Shanghai Jiao Tong University. His research interests include systems software, compiler design and implementation.

Haibing Guan received the PhD degree from Tongji University in 1999. He is a professor at the School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University. His research interests include distributed computing, network security, network storage, green IT and cloud computing.