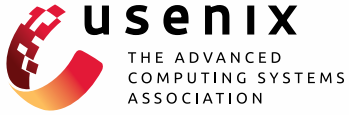

Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks

Ran Liu, *Fudan University and Shanghai Jiao Tong University;* Heng Zhang and Haibo Chen, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/liu

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks

Ran Liu ‡ † , Heng Zhang †, Haibo Chen †

‡*Software School, Fudan University*
†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

ABSTRACT

Reader-writer locks (rwlocks) aim to maximize parallelism among readers, but many existing rwlocks either cause readers to contend, or significantly extend writer latency, or both. Further, some scalable rwlocks cannot cope with OS semantics like sleeping inside critical sections, preemption and conditional wait. Though truly scalable rwlocks exist, some of them cannot handle preemption, sleeping inside critical sections, or other important functions required inside OS kernels. This paper describes a new rwlock called the passive reader-writer lock (prwlock) that provides scalable read-side performance as well as small writer latency for TSO architectures. The key of prwlock is a version-based consensus protocol between multiple non-communicating readers and a pending writer. Prwlock 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 writer lock acquisition latency can be bounded. Evaluation on a 64-core machine shows that prwlock significantly boosts the performance of the Linux virtual memory subsystem, a concurrent hashtable and an in-memory database.

1 INTRODUCTION

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, 25, 10]. For example, it is explicitly recommended to not use rwlocks unless readers hold their locks for a sufficiently long time [9].

While there have been a number of efforts to to improve the scalability of rwlocks, prior approaches either require memory barriers and atomic instructions in readers [22, 18], or significantly extend writer latency [5], or both [12, 2]. Further, many prior designs cannot cope with OS semantics like sleeping inside critical section, preemption and supporting condition synchronization (e.g., wait/signal) [12, 2]. Hence, researchers sometimes relax semantic guarantees by allowing readers to see stale data (i.e., RCU [21]). While RCU has been widely used in Linux kernel for some relatively simple data structures, it, however, 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 or rwlocks inside Linux kernel [20].

This paper describes the prwlock, a scalable rwlock design for read-mostly synchronization for TSO (Total Store Ordering) architectures. Like prior designs such as brlock [12, 2], instead of letting readers actively maintain status regarding inflight readers, prwlock decentralizes such information to each reader and only makes a consensus among readers when a writer explicitly enquires. By leveraging the ordered store property of TSO architectures, such as x86 and x86-64, Prwlock 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 prwlock is a version-based consensus protocol between multiple non-communicating readers and a pending writer. A writer advances the lock version and waits 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, prwlock 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 [4] and only very few straggling readers require message-based consensus, a writer only needs to wait shortly to proceed.

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, prwlock 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.

Prwlock 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, prwlock introduces a parallel wakeup mechanism such that each reader is woken up by the core where it slept from.

We have implemented prwlock as a kernel mechanism for Linux, which compromises around 300 lines of code (LoC). To further benefit user-level code, we also created a user-level prwlock library (comprising about 500 LoC) and added it to a user-level RCU library (about 100 LoC changes). Prwlock can be used in the complex Linux virtual memory system (which currently uses rwlock), with only around 30 LoC changes. The implementation is stable enough and has passed the Linux Test Project [1]. We have also applied prwlock by substituting for a rwlock in the Kyoto Cabinet database [17].

Performance evaluation on a 64-core AMD machine shows that prwlock has extremely good performance scalability for read-mostly workloads and still good performance when there are quite a few writers. The performance speedup of prwlock over stock Linux is 2.85X, 1.55X and 1.20X for three benchmarks on 64 cores and prwlock 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 prwlock consistently outperforms rwlock in Linux (by 7.37X for the Kyoto Cabinet database).

2 Background and Related Work

2.1 Reader/Writer Lock

The reader/writer problem was described by Courtois et al. [13] and has been intensively studied afterwards. However, most prior rwlocks require sharing states among readers and thus may result in poor critical section efficiency on multicore. Hence, there have been intense efforts to improve rwlocks. Table 1 shows a comparative study of different 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 condition wait (e.g., wait until a specific event such as queue is not empty), and whether the lock 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.

	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

Table 1: A comparison of synchronization primitives.

Big-reader Lock (brlock): The key design of brlock is trading write throughput for read throughput. There are two implementations 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 prwlock, 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 inside inside 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.

Prwlock can be viewed as a type of brlock. However, it 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, prwlock can cope with complex semantics like sleeping and preemption, making it viable to be used in complex systems like virtual memory.

C-SNZI: Lev et al. [18] use scalable nonzero indicator (SNZI) [16] to implement 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 a 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. Prwlock 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 prwlock, 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 6), it performs poorly when there are a few writers, 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 prwlock, it eliminates memory barriers in reader fast paths. Yet, its reader fast path is much longer compared to prwlock, resulting in inferior reader throughput. Moreover, as IPIs need always to be broadcasted to all cores, and ongoing readers may contented on the shard queue, its writer lock acquisition is heavyweight (section 6.2.4). In contrast, prwlock leverages bounded staleness of memory consistency to avoid IPIs in the common case.

2.2 Read-Copy Update

RCU increases concurrency by relaxing the semantics of locking. Writers are still serialized using a mutex lock, but readers can proceed without any lock. As a result, readers may see stale data. RCU delays freeing memory until there is no reader referencing to the object, by using scheduler-based or epoch-based quiescence detection that leverage context or mode switches. In contrast, the quiescence detection (or consensus) mechanism in prwlock 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 the 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 prwlock can degrade scalability 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 it into complex subsystems, such as address space management.

2.3 Prwlock's Position

As prwlock strives to achieve scalable reader performance with low reader-side latency, it is designed with a simple yet fast reader fast path, which eliminates the need of reader-shared state and even memory barriers. Yet by leveraging bounded staleness for common cases and IPIs for rare cases, its writer latency is still bounded, especially when readers are frequent.

Prwlock targets the territory of RCU where extremely low reader latency is preferred. Compared to RCU, it trades obstruction-free reader access for a much stronger and clearer semantic and much shorter writer latency. Hence, it can be used to improve performance with trivial effort for cases where RCU is hard to apply.

3 Design of prwlock

3.1 Design Rationale

The essential design goal of reader-writer lock (rwlock) is that readers should proceed concurrently, and thus should not share anything with each other. Hence, a scalable rwlock design should require no shared state among readers and no explicit or implicit memory barriers when there are no writers pending. 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. 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 are no memory ordering dependency among readers. Such unnecessary memory barriers may cause significant overhead for short reader critical sections.

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

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

Table 2: IPI latency in different machines

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]) and sometimes the cost significantly exceeds explicit message passing like inter-processor interrupts (IPIs). Table 2 illustrates the pairwise IPI latency on 2 recent large SMP systems, which is 1,316 and 1,447 cycles accordingly. This latency is low enough to be used in rwlocks, whose writer latency usually exceeds several tens of thousands of cycles.

Further, delivering multiple IPIs to different cores can be parallelized so that the cost of parallel IPI is "indistinguishable" from point-to-point interrupt [23]. 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 in order 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).

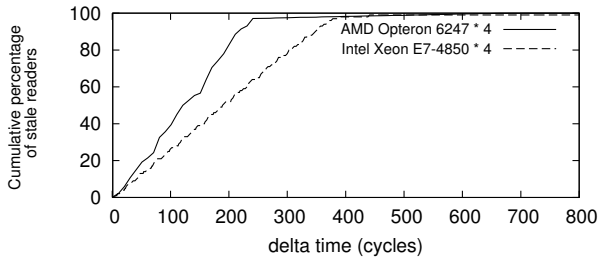

Figure 1: Cumulative percentage of stale readers

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 microbenchmark 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. Figure 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 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 a 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 to serialize the pipeline and flush the store buffer.. 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 prwlock readers never contain explicit memory barriers, and thus might appear to violate this "law of order", prwlock uses IPIs to serialize reader execution with respect to writers, and IPI handling has the same effect as a memory barrier.

3.2 Basic Design

Consensus using bounded staleness: Prwlock 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 Figure 2, ver creates a series of *happens-before* dependencies between readers and writers. A writer can only proceed after all readers have seen its new version. This ensures 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.

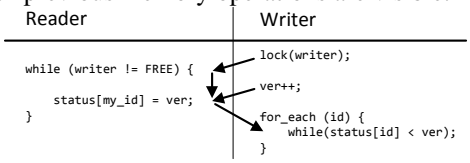

Figure 2: Simple reader-writer lock with version report

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, prwlock introduces a message-based consensus protocol to let the writer actively send consensus requests to readers when necessary. The design is motivated by the relatively small cost for message passing in contemporary processors. Hence, prwlock 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, prwlock uses a hybrid design by combining the above mechanism with traditional counter-based rwlocks. Prwlock 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 before 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.

As sleeping in reader-side critical section is rare, prwlock enjoys good performance in the common case, yet still preserves correctness in a rare case where there are sleeping readers.

3.3 Prwlock Algorithms

Figure 3 and Figure 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 states safely.

Read-side algorithm: Passive readers are tracked distributively 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 (line 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 4).

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 (line 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 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 (line 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 (line 1-2 in *WriteUnlock* and line 2-4 in *WriteLock*).

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;
```

Figure 3: Pseudocode of reader algorithms

Example: The right part of Figure 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 Figure 5 shows an example execution

```
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, CoreID).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;
```

Figure 4: Pseudocode of writer algorithms

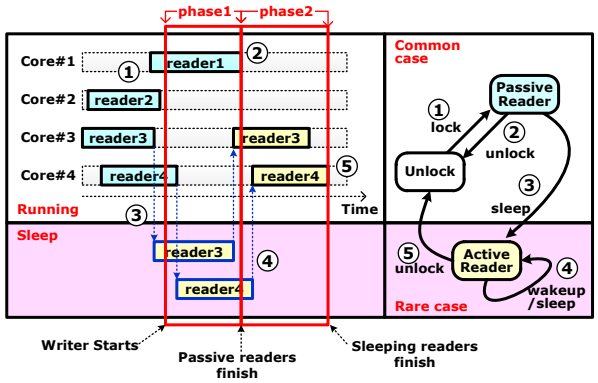

Figure 5: An example execution of readers (left) and the state machine of reader (right). Writer is not shown here.

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 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 enface a strong visibility guarantee between readers and writers. This is guaranteed in prwlock with the help of TSO consistency model.

Once a reader sees an 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 Figure 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.

3.4 OS Kernel Incorporation

There are several issues in incorporating prwlock to an OS kernel. First, the scope of a prwlock could be either global or process-wide and there may be multiple prwlocks in each scope. Each prwlock could be shared by multiple tasks. To reduce messages between readers and writers, prwlock uses the *lock domain* abstraction to group a set of related prwlocks that can do consensus together. A domain tracks CPU cores that are currently executing tasks related to a prwlock. Currently, a domain could be process-wide or global. We now describe how prwlock uses the domain abstraction:

Domain Online/Offline: It is possible that the scope

for a set of prwlocks 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, prwlock 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. Figure 6 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 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 rwlock 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, prwlock 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 prwlocks in the slower *ACTIVE* state. We choose to leave the choice to lock users as they may have more insight on the workload.

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;
```

Figure 6: Domain management algorithms

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. prwlock uses task online/offline to handle such operations. When a task is about to be switched out while holding a prwlock in PASSIVE mode, it will change its lock status to be ACTIVE and increase the active reader counter if it previously holds a prwlock 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: 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.

3.5 User-level Support

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 Figure 3 impossible.

To solve this problem, prwlock instead relies on some kernel support. The idea behind is simple: when it is necessary to perform any operation on per-core state, prwlock enters kernel and lets kernel handle it.

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 lock is maintained in per-core status while active locks are maintained in the shared counter; checking and changing the passive lock mode should be done atomically.

For example, line 2-3 of ReadUnlock algorithm in Figure 7 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 line 2 and line 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 imbalanced 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 Figure 7, 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.

```
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;
```

Figure 7: Pseudocode of unlock algorithm with preemption detection

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, 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).

3.6 Performance Analysis

Memory barrier: In the common path of read-side critical section, prwlock requires no memory barrier when there is no outstanding writer. 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. However, domain online/offline operations are rare in typical execution. Hence, prwlock 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, the cost of IPIs and mode switches are small and usually in the scale of several hundreds 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.

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 6.2.4). 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 prwlock is essentially a distributed rwlock, it needs $O(n)$ space for a lock instance. More specifically, current implementation needs 12 bytes (8 for version and 4 for reader status) per core per lock in order 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), prwlock imposes similar or lower space overhead.

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

4 Decentralized Parallel Wakeup

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.

Our further investigation uncovers that the main performance scalability issue comes from the cascading wakeup phenomenon, as shown in Figure 8. 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

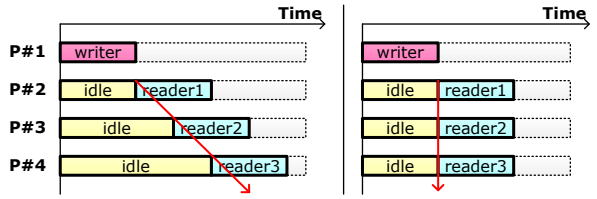

Figure 8: Issue with centralized, sequential wakeup (left) and how decentralized parallel wakeup solve this problem (right).

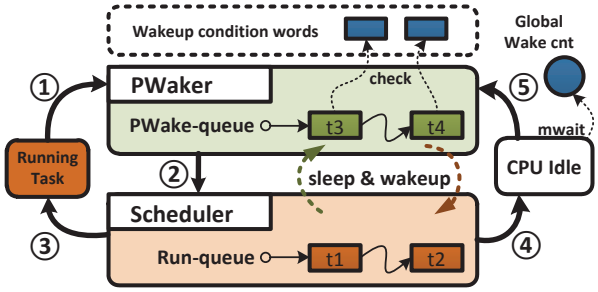

Figure 9: Key data structure and state transition graph of decentralized parallel wakeup in each core.

linearly with the number of readers.

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

Figure 9 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 to 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 wakened up.

5 Implementation and Applications

We have implemented prwlock on several versions of Linux, and integrated it with the Linux virtual memory system by replacing the default rwlock. The porting effort among different versions of Linux is trivial and one student can usually finish it in less than one hour.

Linux address space: As prwlock is still an rwlock, it can trivially replace the original rwlock in Linux virtual memory subsystem. 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 prwlock library comprises of less than 300 LoC and requires manual change of 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 on anonymous memory mapping only) [10], yet with a complete replacement.

User-level prwlock and RCU: We have also implemented user-level prwlock, which comprise about 500 LoC. We further used the consensus protocol of prwlock to implement quiescence detection to implement a user-level RCU; this has better read-side throughput and faster quiescence detection than previous user-level quiescence detection mechanisms (section 6.3). We modified a famous database system named Kyoto Cabinet [17], by replacing a rwlock with prwlock to protect its data tables.

6 Evaluation

6.1 Evaluation Setup

Kernel prwlock: We use three workloads that place intensive uses of virtual memory: Histogram [24], which is a MapReduce application that counts colors from a 16GB bitmap file; Metis [19] from MOSBENCH [6], which computes a reverse index for a word from a 2GB 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. We also implemented a concurrent hashtable [25] in kernel as a micro-benchmark to characterize prwlock and its alternatives.

User-space prwlock: We use several micro-benchmarks to compare prwlock with several alternatives like brlock and user-level RCU. As prwlock has a user-level RCU library, we also compare its performance

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

to traditional signal-based user space RCU [14]. To show that prwlock can scale up user-space applications, we also evaluated the Kyoto Cabinet database using prwlock and the original rwlock.

As the performance characteristic that prwlock 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. For each benchmark, we evaluate the throughput in a fixed time and collect the arithmetic mean of five runs.

6.2 Kernel-level prwlock

6.2.1 Application Benchmarks

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. 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 prwlock 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 prwlock, as shown in Figure 10. 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% respectively on 64 cores.

It was quite surprising that prwlock 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 Figure 11, prwlock performs near linearly to 64 cores with a speedup over percpu-rwlock and original Linux by 27% and 55% 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, which not only taxes page fault handler, but also mmap operations. Due to extended mmap latency, percpu-rwlock cannot scale beyond 4 cores, as shown in Figure 12. In contrast, prwlock performs similarly with Linux before 32 cores and eventually outperforms Linux after 48 cores, with a speedup of 20% 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, prwlock performs worse than RCUVM as RCUVM allows readers to proceed in parallel with writers. Under 64 cores, prwlock is around 6% 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.

6.2.2 Benefits of Parallel Wakeup

Figure 13 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% when there are multiple readers waiting. prwlock improves the performance of original Linux by 47.6%. 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 prwlock, which are 934us, 1014us and 567us, 344us. With the fast wakeup mechanism, the cost for Linux has decreased to 697us and 354us.

6.2.3 Benefits of Eliminating Memory Barriers

We use a concurrent hashtable [25] to compare prwlock with RCU, rwsem and brlock. Figure 16 illustrates the performance. RCU has a nearly zero reader overhead and outperform all rwlocks. The throughput of rwsem vanishes because of cache contention. Thanks to elimination of memory barriers, prwlock shows higher throughput than brlock. More tests reveal that the lookup overhead mainly comes from cache capacity misses while accessing hash buckets. Prwlock's speedup over brlocks would

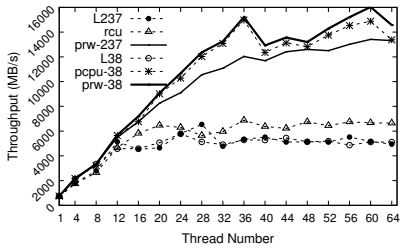

Figure 10: Histogram throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

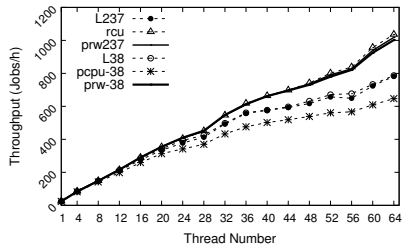

Figure 11: Metis throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

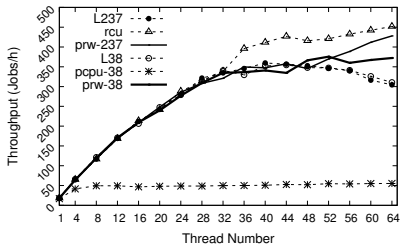

Figure 12: Psearchy throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

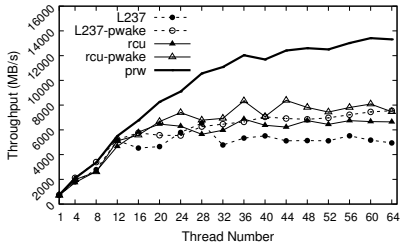

Figure 13: Benefit of parallel wakeup for Histogram.

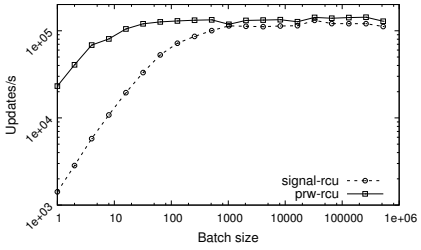

Figure 14: Update performance with batch size

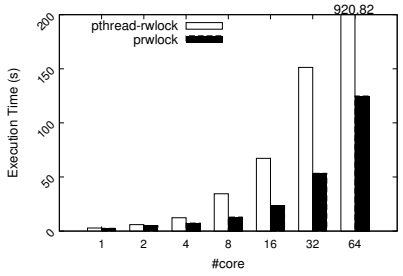

Figure 15: Benefit of prwlock for an in-memory DB

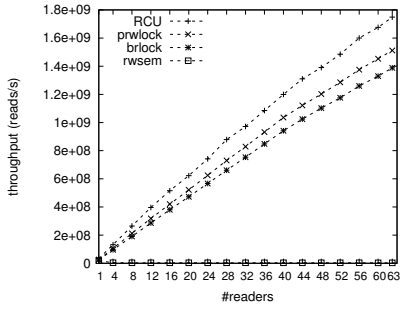

Figure 16: Lookup performance of hashtable

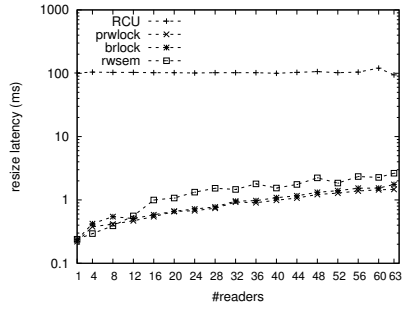

Figure 17: Resize latency of hashtable

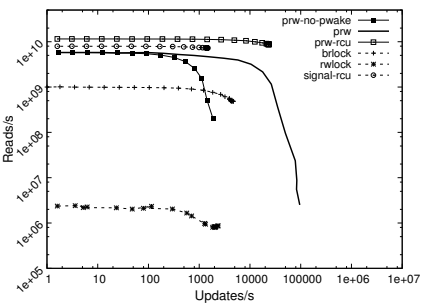

Figure 18: Relation between reader/writer throughput

be much larger if there was a cache hit (not shown here).

By using rwlocks instead of RCU, resizing the hashtable is much simpler and faster as all readers are blocked during resizing. Figure 17 presents the total latency to shrink and grow the hash table on different concurrency levels. Rwlocks shows up to two orders of magnitude shorter resizing latency compared to RCU. As hashtable resizes have negative impact on lookup performance, shorter resize latency is desirable to maintain a stable lookup performance. Prwlock only shows marginally better performance compared to other two rwlocks, as in this test most of the time is spent in critical section rather than writer lock acquisition.

6.2.4 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. prwlock shows best reader performance as its common path is simple and has no memory barriers. It is interesting that prwlock has much higher writer latency when there is no reader, since the writer has to use IPIs to ask every online core to report. 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.

	brlock	rmlock	rwsem	prwlock
Reader latency (1 reader)	58	46	107	12
Reader latency (64 readers)	58	46	20730	12
Writer latency (0 reader)	17709	136	100	65511
Writer latency (63 readers)	89403	622341	3235736	6322

Table 3: Critical section efficiency (average of 10 millions runs)

6.3 User-level Prwlock

Figure 18 shows the impact of writer frequency on reader throughput for several locking primitives, by running 63 reader threads and 1 writer thread. Writer frequency is controlled by varying the delay between two writes, which is similar done as Desnoyers et al. [14]. Note that 1 writer is the worst case of prwlock since if there is more than 1 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 algo-

rithms to 1. That means they must wait a grace period for every update.

Prwlock achieves the highest writer rate. This confirms that our version-based consensus protocol is more efficient than prior approaches. Prwlock'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.

We can also notice that prwlock-based RCU performs consistently better than the signal-based user-level RCU. Thanks to prwlock's kernel support, the reader-side algorithm of prwlock RCU is simpler, which results in a higher reader throughput. Besides, prwlock-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 Figure 14. Prwlock-RCU reaches its peak performance before the batch size reaches 100 and performs much better when the batch size is less than 1000. Small batch size helps control the memory footprint since it allows faster reclamation of unused objects.

Kyoto Cabinet: Figure 15 shows the improvement of prwlock 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, prwlock outperforms original rwlock and the improvement increases with core count. Under 64 cores, prwlock outperforms pthread-rwlock by 7.37X (124.8s vs. 920.8s). The reason is that the workload has hundreds of millions read accesses and pthread-rwlock incurs high contention on the shared counter, while prwlock places no contention in the reader-side.

7 Conclusions and Future Work

This paper has described passive reader-writer lock, a reader-writer lock that provides scalable performance for read-mostly synchronization. Prwlock can be implemented in both kernel and user mode. 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. In future work, we will investigate and optimize prwlock in a virtualized environment (which may have higher IPI cost).

Acknowledgment

We thank our shepherd Eddie Kohler and the anonymous reviewers for their insightful comments and suggestions. This work is supported in part by China National Natural Science Foundation (61003002), Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of China, and Singapore CREATE E2S2.

References

[1] Linux test project. http://ltp.sourceforge.net/.

[2] Version-based brlock. 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. In *PPoPP*, 2011.

[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 *SOSP*, pages 29–44, 2009.

[5] S. S. Bhat. https://patchwork.kernel.org/patch/2157401/, 2013.

[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 *OSDI*, 2010.

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

[8] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *PPoPP*, 2013.

[9] B. Cantrill and J. Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, 2008.

[10] A. Clements, M. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

[11] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys*, 2013.

[12] J. Corbet. Big reader locks. http://lwn.net/Articles/378911/.

[13] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Comm. ACM*, 14(10), 1971.

[14] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-level implementations of read-copy update. *TPDS*, 23(2):375–382, 2012.

[15] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *PPoPP*, 2012.

[16] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC*, pages 13–22, 2007.

[17] FAL Labs. Kyoto Cabinet. http://fallabs.com/kyotocabinet/.

[18] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *SPAA*, 2009.

[19] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for multicore architectures. In *MIT Tech. Rep*, 2010.

[20] P. Mckenney. RCU usage in the Linux kernel: One decade later. http://www2.rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf.

[21] P. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symposium*, 2001.

[22] J. Mellor-Crummey and M. Scott. Synchronization without contention. In *ASPLOS*, pages 269–278. ACM, 1991.

[23] D. Petrović, O. Shahmirzadi, T. Ropars, A. Schiper, et al. Asynchronous broadcast on the Intel SCC using interrupts. In *Many-core Applications Research Community Symposium*, 2012.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

[25] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX ATC*, 2011.