Scalable reader-writer locks

Felix Klingelhoefer

April 2020

Abstract

Over the past years, the number of cores and threads has been rapidly increasing, in both personnal and industrial machines. This makes scalability paramount to achieving the best performance, which requires particular attention to the interactions between threads, especially for memory access. In this work we look at efficient solutions for the famous reader-writer lock, in the case of massively parrallel access. We present novel scalable reader-writer locks, and explore the tradeoff between phase-fairness and scalabilty. These algorithms are formally verified in tla+. The performance of these solutions is compared to other reader-writer locks that were not designed for scalability using different benchmarks on a Xeon Phi server.

1 Introduction

One of the main issues of parallel programming is managing concurrent threads trying to access the same shared resource. Letting all the threads work on the common resource simultaneously can lead to very unexpected results, so a portion of code, in the form of a lock, is added to control its access. Mutual exclusion locks are synchronization primitives that are designed to avoid concurrent access to this resource for all threads. A mutex lock can be owned by at most one thread at a time. When a thread gains ownership of the mutex, it is granted access to the shared resource, and can execute its *critical section*, which is the portion of code it wants to execute on the resource (eg. a store). In their review paper [1], the authors identify several critical properties for a mutual exclusion lock to be considered correct. Given a number of threads that may run at arbitrary speed and in arbitrary order:

1. Only one thread may be in a critical section at a time; this is called *mutual exclusion*.

2. If a thread is not in the entry or exit code of a lock, it may not prevent other threads from accessing the critical section, which would be a *deadlock*.

3. In selecting a thread for entry to the critical section, the choice may not be postponed indefinitely. Otherwise, there is a *livelock*.

4. Any thread that arrives in the entry section must eventually get through its critical section. If it is not the case, the thread is subject to *starvation*.

There is a *weak fairness* assumption on the actions of threads: all threads will eventually execute their next step. This is different from *strong fairness*, which guarantees that a thread that is infinitely often in a given state will take a step from that state. To exemplify the difference, imagine a binary clock (switches from 0 to 1 and vice versa infinitely). If a thread can only access its critical section if the clock has value 1, under weak fairness assumptions it is never guaranteed to get through: it may test the value of the clock only when it has value 0. This leads to starvation, which would not be the case under strong fairness assumptions.

There is a different fairness notion for the lock, which is a measure of how many other threads may overtake a given waiting thread. If all are serviced on a First In First Out (FIFO) basis, the lock is said to be *task-fair*. While this gives the best theoretical complexity guarantees, it can be a hindrance to performance, and is not a necessary property of mutual exclusion locks.

Reader Writer (RW) locks were developed on the observation that some tasks (reads) can be executed simultaneously without a problem, as long the data is not modified at the same time (writes). Mutual exclusion locks will unnecessarily serialize all these tasks. Mutual exclusion is therefore relaxed for RW locks: exclusion among writers and between readers and writers is maintained, but needless exlusion between readers is dropped.

When the thread count increases, especially for short critical sections, having an entry and exit code that scale efficiently to large numbers of actors becomes paramount. This is the notion of *scalability*. In this work, we propose a highly scalable RW lock, and explore the tradeoff between fairness and scalabilty. In order to scale well to a large number of threads, particular attention must be given to the use of shared resources, as well as blocking and non blocking memory accesses.

2 Algorithms

2.1 Cohort based Reader Writer lock (C-RW)

This algorithm is adapted from an algorithm in [2]. It is based on the technique of lock cohorting, which allows the lock to take advantage of non uniform memory access. To do so, it uses a cohort lock [3], that makes threads which are on the same NUMA node compete for a local ticket lock. The owners of the local lock then compete for a top-level partitioned ticket lock. While writers compete

for this mutual exlusion lock, readers wait for the lock to be unlocked to get through, which can lead to reader starvation, since the mutex may always be locked when a reader attempts to enter. This would be the case if other writers barge ahead every time the cohort lock is opened. In the original paper, the authors solve this problem by introducing a patience limit for the readers, after which they block writers from reacquiring the lock. However, this is done in a way that introduces writer starvation: a given writer thread may repeatedly try to enter when the patience-based barrier is up, and never get in.

In the original work, the authors use a simple counter for the read indicator. It is incremented by the arrive() method, and decremented by depart(). WaitUntilIsEmpty() then simply busy waits for the indicator's value to be 0. In order to achieve better scalabilty, we replaced it with a distributed indicator, in which each thread had its own binary indicator, which is set to 1 by arrive() and 0 by depart(). WaitUntilIsEmpty() then busy waits iteratively for each thread's indicator to be 0.

Algorithm 1 C-RW

```
reader:
  while True do
    while CohortLock.isLocked() do
      Pause
    end while
    ReadIndicator.arrive()
    if not CohortLock.isLocked() then
      break
    end if
    ReadIndicator.depart()
  end while
  Critical Section
  ReadIndicator.depart()
writer:
  CohortLock.Lock()
  ReadIndicator.WaitUntilIsEmpty()
  Critical Section
  CohortLock.Unlock()
```

2.2 Handover based Reader Writer Lock (H-RW)

Many writer threads trying to acquire the mutex lock simultaneously can hurt performance a lot. One way of reducing the impact, in situations with high contention, is to handover the lock between writers a number of times before releasing it. We developed the Handover Reader Writer lock based on this idea. In essence, the writer threads look for other writers to hand over the mutex to when they are releasing the lock. They then tell the given thread, thanks

to a distributed indicator, that they now own the lock and can go through the critical section. In order to avoid starvation, this handover is offered once to every other thread, one ater the other. Then the writers back off for a reader phase, and open the lock.

One main property of this lock is that it spaces out read phases, by up to *nThreads* write phases. It has the advantage of letting read requests "pile up" before being all executed at once, but the drawback of offering less good complexity guarantees on read requests, compared to other RW locks. In section 3 we will present derivations of this algorithm which alternate between read and write phases.

2.2.1 Reader phase

Reader threads communicate their intentions to the writers on a distributed indicator (ReadIndicator), by setting their state to away (default), reading, attempting barrier, or barrier.

A reader phase is initiated either when the writer mutex lock is unlocked, or a writer sets the backoff variable to True. In the first case, readers can simply access the critical section until a writer arrives, and signal their presence by setting their state to anything other than away. When a writer thread takes the mutex, it will first wait for all reading threads to depart.

A writer initiates a reader phase by baking off when it sees any reader thread that put up a barrier (either barrier or attempting barrier state). There are then two parts to the phase. While backoff is set to true, all readers in the barrier or attempting barrier state will go through the critical section. The writer thread waits for this to happen by itiratively busy waiting for each individual thread to have a state different than barrier or attempting barrier ((*waitUntilBarrierDown()*). Any new readers that arrive during this time will access the lock, but in the read state. The writer threads then retracts backoff (by setting its value to False), and no new readers will be able to access the critical section. It then iteratively waits for all the remaining readers to leave (no longer be in the read or attempting barrier state) with the *waitUntilReadEmpty()* method.

When a reader attempts to enter the lock, it will first try to go through in the reading state. If it cannot (mutex locked or backoff set to false), it will attempt to go through in the attempting barrier state. If still not possible it will set the barrier state, and wait to finally be able to get in. The intermediate attempting barrier state is a hybrid of both reading and barrier. It is needed to avoid mutual exclusion violation because even when the writer waited for all threads in the attempting barrier or barrier state to no longer be in them, a thread that avaluated the if condition prior to backoff being set could pass from the reading state to the attempting barrier state.

2.2.2 Writer phase

Similarly to readers, writers communicate their presence to other writers through a distributed indicator (WriteIndicator), but it only has two states: away (de-

Algorithm 2 H-RW - Readers

```
reader:
  ReadIndicator.read(self)
  if Lock.isLocked() and not Backoff.load() then
    ReadIndicator.attemptBarrier(self)
    if Lock.isLocked() and not Backoff.load() then
      ReadIndicator.barrier(self)
      while Lock.isLocked and not Backoff.load() do
        Pause()
      end while
    end if
  end if
  Critical Section
  ReadIndicator.depart()
```

Algorithm 3 H-RW - Writer backoff

```
SetBackoff():
  if ReadIndicator.isBarrier() then
    Backoff.store(True)
    ReadIndicator.waitUntilBarrierDown
  end if
RetractBackoff():
  if Backoff.load() then
    Backoff.store(False)
    ReadIndicator.waitUntilReadEmpty()
  end if
```

fault) and waiting. Their access to the lock is controlled by a mutex, *Lock*. Its method *takeLock* atomically (through a compare-and-swap instruction), sets the lock to locked and returns true if it was unlocked, or returns false if it was already locked.

A writer can take ownership of the mutex lock, and therefore the reader-writer lock in two ways. The first is the *takeLock* method, which atomically (through a compare-and-swap instruction), sets the lock to locked and returns true if it was unlocked, or returns false if it was already locked. The second is by implicitely being granted ownership through handover. This is done when a writer set its state to waiting, and then the owner of the lock sets the state of the first to away. Therefore, to see if the lock was handed over to it, a writer tests wether its state in the indicator is still waiting. When a writer arrives, it will therefore start by setting its state to waiting, then try to take the lock. If unsucessfull, it busy waits for the lock to be opened or to be handed over to it. After it stops waiting, if it was not handed over the mutex it will attempt to take the lock again. If this also fails, it now knows that another thread took the lock after it started waiting, so it is guaranteed to be handed over the lock at some point, and busy waits untill that time.

Now that the writer got ownership of the lock, the two cases must be distinguished. If it took it directly, it will start by doing a full reader phase, because while the lock was open readers may have accessed the lock in any state, and it will initialise the parameters of the handover cycle (*Start* and *End*). In case of a handover, the previous owner set the backoff, and the writer only needs to retract it and wait for the readers to leave the lock.

After going through the critical secction, writers will try to hand over the lock to the next thread numerically, in the cycle of length the number of threads. It will do so only if the other thread is in the waiting state. Once a handover cycle is complete, the writer will set backoff to initiate a reader phase, and then start a new backoff cycle or free the lock if there is no writer waiting. This guarantees a linear wait for readers, who once they executed their entry section can be delayed by at most the number of threads writers.

Algorithm 4 H-RW - Writer lock

```
ExclusiveLock():
  WriteIndicator.wait(self)
  if not Lock.takeLock() then
    while WriteIndicator.isWaiting(self) and Lock.load() do
      Pause
    end while
    if not WriteIndicator.isWaiting(self) then
      RetractBackoff()
      Start = self
      return
    end if
    if not Lock.takeLock() then
      while WriteIndicator.isWaiting(self) do
        Pause
      end while
      RetractBackoff()
      Start = self
      return
    end if
  end if
  SetBackoff()
  Backoff.store(False)
  ReadIndicator.waitUntilReadEmpty()
  WriteIndicator.depart(self)
  Start = self
  End = self
```

Algorithm 5 H-RW - Writer unlock

```
writer:
  if HandoverWriter() then
    return
  end if
  if ReadIndicator.isBarrier() then
    Backoff.store(True)
    ReadIndicator.waitUntilBarrierDown
    End = Start
    if HandoverWriter() then
      return
    end if
    Backoff.store(False)
    Lock.unlock()
  end if
```

Algorithm 6 H-RW - HandoverWriter()

```
HandoverWriter()
  for i = (start + 1) mod nThreads, with i ≠ end, step i = (i+1) mod
  nThreads do
    if WriteIndicator.isWaiting(i) then
      WriteIndicator.depart(i)
      return True
    end if
  end for
  return False
```

3 Phase-Fairness

Phase fairness is a concept exposed in [4] to extend the notion of fairness to reader-writer locks. Indeed, task fairness is an ill-suited notion for RW locks, since it would require all readers to be executed sequentially. This would completely remove the advantages of the relaxed mutual exclusion over mutex locks. We take their definition of phase-fairness.

A RW lock is said to be phase fair, if and only if it satisfies the following properties:

PF1: reader phases and writer phases alternate

PF2: writers are subject to first-in-first-out (FIFO) ordering with regard to other writers

PF3: during each reader phase, all read requests that were unsatisfied at its start will be executed

PF4: during a reader phase, newly-issued read requests are satisfied only if there are no unsatisfied write requests pending.

These properties give bounds on the complexity of read and write requests: a reader can be blocked by at most one read and one write phase, whereas a writer can be blocked by at most (nThreads - 1) writer and reader phases.

Phase fairness can be seen as the middle ground between writer preference locks, where arriving barge ahead of readers and can lead to reader starvation, but guarantee liner wait for writers, and reader preference locks, which ensure constant wait for readers but can starve writers. In this type of lock, both readers and writers cede to each other after every phase.

3.1 Phase Fair Cohort Reader Writer lock (PF-C-RW)

This algorithm was initially created as a phase-fair version of the cohort reader-writer, but modified so much it no longer has much resemblance. First of all, in order to ensure strict FIFO ordering for writers, the mutex used by the writers was changed to a ticket lock. Indeed, the cohort lock only guarantess linear wait, and not FIFO ordering, except in degenerate cases (where one node is either a single thread or all the threads) where it becomes an overcomplicated ticket lock.

The writer part of this lock is fairly simple: a thread arrives, competes for the ticket lock, does one backoff phase, goes through its critical section and relinquishes control of the lock. Meanwhile, the reader either goes through its critical section directly if the lock is unlocked, or gets to the barrier state otherwise. In order to avoid mutual exclusion violation, an attempting barrier state is used again, in which the reader will retract the barrier if it sees the writer set backoff before it could confirm its barrier. Thanks to this, the reader is guaranteed, once in the barrier state, that the writer will wait for it during its next backoff phase, so it can go through the critical section when backoff is set.

Let us show that this algorithm verifies all the properties required for phase-fairness. First of all, the FIFO ordering of writers (PF2) is guaranteed by the ticket lock. It is also simple to see that phases alternate (PF1): before each writer phase there must be a reader phase during the time it sets backoff. A reader phase starts either when the lock is unlocked, or backoff is set. Any reader that executed all its entry section must be in the while loop that becomes untrue in both these cases, and also in the barrier state. Since the writer waits for all readers to exit this state, all these threads must go through their critical section before the end of the reader phase (PF4). Finally, when a writer arrives, he immediately sets backoff after taking the lock. After that point, no newly arriving reader can get through its critical section (PF3), since it will be blocked in the first while loop.

3.2 Phase Alternating Handover Reader Writer lock (PA-H-RW)

We made two modifications of H-RW, that are both not phase-fair, but satisfy more of the required properties. This first algorithm is made to have alternating read and write phases (PF1). It also guarantees any reader that is waiting at the beginning of a reader phase to get through (PF3). It does not however respect the FIFO ordering requirement (PF2), nor does it guarantee that newly arriving readers do not get through in a given reader phase if there is a writer waiting (PF4).

Compared to H-RW, the only modification is in the writer unlock. The fact that a backoff phase is initiated at every handover makes it simpler: there is no more need to track how many successive handovers were made, and whether a backoff phase is necessary.

Algorithm 7 PF-T-RW

```
reader:
  ReadIndicator.read(self)
  if Lock.isLocked() then
    ReadIndicator.attemptBarrier(self)
    if Backoff.load() then
      ReadIndicator.read(self)
      while Backoff.load() do
        Pause
      end while
    end if
    ReadIndicator.barrier(self)
    while Lock.isLocked and not Backoff.load() do
      Pause()
    end while
  end if
  Critical Section
  ReadIndicator.depart()
writer:
  TicketLock.Lock()
  Backoff.store(True)
  ReadIndicator.waitUntilBarrierDown()
  Backoff.store(False)
  ReadIndicator.WaitUntilReadEmpty()
  Critical Section
  TicketLock.Unlock()
```

Algorithm 8 PA-H-RW - Writer unlock

```
writer:
  SetBackoff()
  for i = (start + 1) mod nThreads, with i ≠ start, step i = (i+1) mod
  nThreads do
    if WriteIndicator.isWaiting(i) then
      WriteIndicator.depart(i)
      return
    end if
  end for
  RetractBackoff()
  Lock.unlock()
```

algorithm	PF1	PF2	PF3	PF4	writer complexity	reader complexity
CRW	no	no	no	yes	$\mathcal{O}(n)$	reader starvation
HRW	no	no	yes	no	$\mathcal{O}(n)$	$\mathcal{O}(n)$
PAHRW	yes	no	yes	no	$\mathcal{O}(n)$	$\mathcal{O}(1)$
SPAHRW	yes	no	yes	yes	$\mathcal{O}(n)$	$\mathcal{O}(1)$
PFCRW	yes	yes	yes	yes	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Table 1: Summary of the properties of the different algorithms

3.3 Short Phase Alternating Handover Reader Writer (SPA-H-RW)

This second algorithm derived from H-RW verifies the same properties as the first, but also that any reader that arrives during a reader phase will not go through if a writer is waiting. In order for this to be true, the reader unlock is modified to be that of PF-T-RW, which was designed with that objective in mind. The writer unlock is the same as that of PA-H-RW, which gives alternating phases.

The only remaining property of phase-fairness that is not verified is the FIFO ordering amongst writers. The most efficient way of adding this property is using a ticket system, which would make it into PF-T-RW.

We summarized the properties of all our algorithms in Table 1.

Algorithm 9 SPA-H-RW - reader

```
reader:
  ReadIndicator.read(self)
  if Lock.isLocked() then
    ReadIndicator.attemptBarrier(self)
    if Backoff.load() then
      ReadIndicator.read(self)
      while Backoff.load() do
        Pause
      end while
    end if
    ReadIndicator.barrier(self)
    while Lock.isLocked and not Backoff.load() do
      Pause()
    end while
  end if
  Critical Section
  ReadIndicator.depart()
```

4 Formal verification

All the algorithms we presented were verified in the tla+ formal verification language. They were translated from PlusCal specifications. PlusCal is an imperative language that enables algorithms to be defined similarly to other programming languages. They are then automatically transpiled into tla+. In order to verify the corectness of the algorithms, properties and invariants can be written, which are then checked in all reachable states when the model is run. Since the number of reachable states evolves exponentially with the number of threads, we only verified the algorithms for up to 4 threads.

The default model in tla+ already checks the translated algorithm for liveness (no livelocks), so we only wrote properties to verify mutual exclusion and starvation. Each line of code in pluscal is defined by a label, and we named the entry of the read section r1, and that of the write w1. The read and write critical sections were labelled respectively csr and csw. The two invariants to check for mutual exclusion were then:

$$\forall i \in \{0,1,2,3\}, \forall j \in \{0,1,2,3\}, \forall i \neq j, pc[i] \neq csw \vee pc[j] \neq csw \tag{1}$$

which ensures strict mutual exclusion among writers,

$$\forall i \in \{0,1,2,3\}, \forall j \in \{0,1,2,3\}, pc[i] \neq csw \vee pc[j] \neq csr \tag{2}$$

which guarantees mutual exclusion between readers and writers. These are checked to be true in all reachable states, with *pc[i]* indicating the current label of the i-th thread.

Two properties, which are checked to be true for every execution were used to guarantee that there is no starvation:

$$\forall i \in \{0,1,2,3\}, pc[i] = r1 \sim> pc[i] = csr \tag{3}$$

for reader starvation,

$$\forall i \in \{0,1,2,3\}, pc[i] = w1 \sim> pc[i] = csw \tag{4}$$

for writer starvation. The symbol $\sim>$ designs *leads to*, which means that in any given execution if the first property is verified at some step, the second must be verified at a later step.

In order to test whether there are any deadlocks in the algorithms, we made the process a tripartite choice between executing the reader section, the writer section, or an "idle" section, which simulates a thread being away. That way, a thread could repeatedly execute the idle section, and if there were a deadlock, properties 3 or 4 would be violated.

5 Empirical Evaluation

In this section, we present the results from our experimental benchmarks, in which we compare the different algorithms we presented. The benchmark we

use is the same as [2]. We ran experiments on a Xeon Phi server, with x86 architecture, which possesses 64 cores, and 4 threads per core. The clock speed of the cpus is of 1.30 MHz. For practical reasons, we limited our experiments to 192 threads, of the 256 available. We compare our locks, and the reader writer lock from the pthread library, *PThreadRW*.

5.1 Throughput test

The first benchmark measures the throughput of the lock (total number of threads executing their critical section), over a given amount of time (20s in our experiments). During that time, a number of threads (which varies as a parameter of the benchmark), randomely choose to execute either a read or a write request, according to a specified probability. After executing their critical section, threads execute a non-critical section before returning with a new request. The length of both the critical section for readers and writers, as well as the non-critical section can be adjusted, but we fixed the parameters to RCSLen = 4, WCSLen = 4, and NCSLen = 32 in this experiment.

Figure 1 shows the results of the Throughput test. We can see that the handover RW lock performs best in all tests when the number of threads is high. It is also competitive even for loxer amounts of threads. Since read requests can be executed in a parrallel fashion, it is no surprise to see much higher throughput for lower percentages of writers. The limitation of hardware appears as of 64 threads: in the 100% writer test, the performance is near constant up to that point, and for the 0% writer case, the linear increase (which appears exponential with the logscale on the x-axis) becomes logarithmic (linear in appearance). H-RW manages to leverage the parrallelization of read requests to improve throughput up to that point in all tests where there are readers, while for the other RW locks the increased complexity of control code outshadows those benefits. We can therefore see that it appears to be the only truely scalable lock in our experiments.

When comparing it to its variations, it appears that satisfying more properties of phase-fairness comes at the cost of performance on this benchmark: PA-H-RW outperforms SPA-H-RW, which beats PF-T-RW.

5.2 Dedicated test

In the dedicated benchmark, each thread is originally assigned a role: either reader or writer. Exactly half of the threads are assigned to each. The test is then the same as a throughput test, only a reader will always issue read requests (and a writer writes). The total of reads and writes is then measured over a given time (here 20s), and reported in Figure 2, for a varying number of total threads.

In this test, we can see that PA-H-RW executes by far the most reads. This is because contrary to the previous case, executing many write requests in a row will not lead read requests to pile up, as all threads have a dedicated role. The benefit of our two phased backoff, in which newly arriving readers can enter the

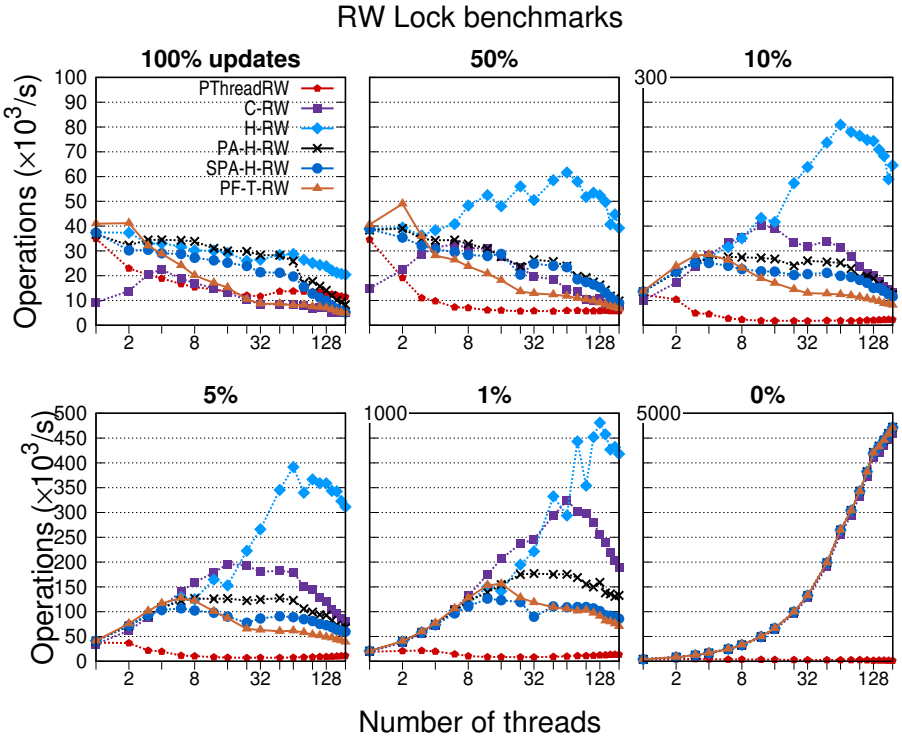

Figure 1: Throughput test for RW locks with varying writer percentages. The number of threads varies from 1 to 192 on a logscale on the X-axis. The Y-axis shows the total number of operations for all threads.

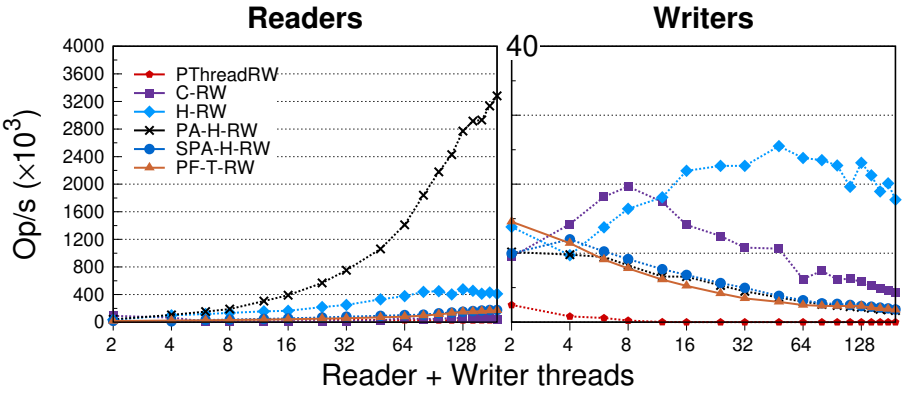

Figure 2: Dedicated test for RW locks with varying writer percentages. The number of readers is always equal to the number of writers, and varies from 1 to 96, for a total of 2 to 192 thread, on a logscale on the X-axis. The Y-axis shows the total number of operations for all reader or writer threads.

critical section while the writer hasn't retracted backoff does become obvious, by the huge advantage of PA-H-RW over SPA-H-RW, which executes at least as many reader phases, but each of them much less efficient. H-RW, which favors writers by chaining their requests compared to the alternating phase algorithms unsurprisingly has the most write requests satisfied.

6 Conclusion

We developed a RW lock that offers highly scaling throughput in most cases (H-RW). It does not, however, ensure fast read request satisfaction, nor the best theoretical guarantees on write requests. However, as we saw in its variations that approach *phase-fairness* as well as in our phase fair lock, these come at a price in empirical performance. If these are worth the cost is entirely dependant on the desired application.

References

[1] Peter Buhr, David Dice, and Wim Hesselink. High-performance n-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27, 05 2014.

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

[3] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[4] Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46:25–87, 2010.

A What work I did

I was provided with the code for the CRWWP algorithm, as well as that for the benchmarks by Andreia Correia and Pedro Ramalhete. The original idea for the H-RW algorithm was also given to me by them, though we revised it substantially together (the initial version turned out to not be correct, and we made other optimizations). I wrote all the specifications for the formal verifications, as well as the other algorithms, and ran all the experiments (I was provided server access).