Deadlock-Free Try Reader-Writer Locks

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

ABSTRACT

A Reader-Writer lock is a common concurrency construct that provides simultaneous read-only access for multiple threads (readers) or exclusive read-modify access to a single thread at a time (writer). Many algorithms to implement a reader-writer lock exist, and most, if not all, are capable of `trylock()` functionality, returning in a finite number of steps whether the lock was successfully acquired or not. For most of these algorithms, the `trylock()`s may fail spuriously, where by multiple threads simultaneously attempt to acquire an unlocked lock and all fail. Because the `trylock()` methods are wait-free, reader-writer locks without spurious `trylock()` failures can be used to implement wait-free pools and other wait-free linearizable mechanisms.

We present two new reader-writer lock algorithms where `trylock()`s are guaranteed to succeed unless there is a linearizable history for which a thread already has the lock or is guaranteed to acquire it, i.e. our algorithms do not have spurious failures for `trylock()`, a property sometimes referred to as *Deadlock Freedom*. The first algorithm, named RWTryLock-DF, uses a single word of memory and is simple but has low scalability for readers. The second algorithm, named RWTryLock-DF-ReaderState, provides good scalability for read-mostly workloads at the cost of more memory usage. Both our algorithms have reader-preference.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

mutual exclusion, reader-writer locks, locks

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Algorithm 1 trylock()s for C-RW-RP

```
1  bool sharedTryLock() {
2      ReadIndr.arrive();
3      if (CohortLock.isLocked()) {
4          ReadIndr.depart();
5          return false;
6      }
7      return true;
8  }
9
10 bool exclusiveTryLock() {
11     if (!CohortLock.trylock()) return false;
12     if (ReadIndr.isEmpty()) return true;
13     CohortLock.release();
14     return false;
15 }
```

1. INTRODUCTION

Reader-Writer locks are the simplest way to synchronize between one *writer* and multiple *readers* and they have a wide variety of applications [5]. Reader-Writer locks have been studied for several decades [1, 3, 6, 10, 7, 8, 9], with the latest research pointing to NUMA-awareness [2] as the best way to improve scalability for read-mostly workloads.

When comparing with the state of the art in reader-writer locks, the most obvious comparison is the reader-preference algorithm C-RW-RP in figure 4 of [2]. Although the original paper does not explain how to implement the `trylock()`s, it is reasonable to expect that the implementation for C-RW-RP would look like the one shown in Algorithm 1.

Like most reader-writer locks, C-RW-RP can have spurious failures on its `trylock()`s. One reader and one writer can concurrently attempt to acquire the lock and both fail to take the lock. This scenario happens when the reader arrives (line 2 of Algorithm 1), the writer acquires the cohort lock (line 11), then the reader sees the cohort lock has been acquired (line 3), the writer sees the ReadIndr is not empty (line 12) and returns false (line 14), and the reader also returns false (line 5).

Reader-writer lock algorithms whose `trylock()` methods do not have spurious failures are said to have *Deadlock Freedom* for this API. The property of Deadlock Freedom guarantees that *the critical section will not*

become inaccessible to all processes. This means that if a number of processes attempt to execute their critical sections, then after a finite number of steps, some process will be allowed to do so. Reader-writer locks are linearizable objects and therefore, there must exist a linearizable history in which (at least) one thread executed its critical section, i.e. successfully acquired the lock. Notice that all *correct* reader-writer lock algorithms must have deadlock freedom at least for their `lock()/unlock()` API.

The next sections will show two different algorithms for reader-writer locks with deadlock freedom for their `sharedTryLock()` and `exclusiveTryLock()` methods, named RWTryLock-DF and RWTryLock-DF-ReaderState. Both algorithms are capable of providing linearizable trylock functionality without spurious failures [4].

The motivation for the design of a reader-writer lock with deadlock free `tryock()`s, is to provide wait-free progress in scenarios where multiple resources are available. For example, suppose N threads are competing for one of N resources, where they may need the resource in *shared* or *exclusive* mode. A sensible approach is to protect each resource with a reader-writer lock instance and access it using `trylock()` methods, where all threads start their attempt to acquire the lock from the *first* instance going up to the *last* instance following a pre-defined order. If the `trylock()` has no spurious failures, each thread is guaranteed one of the N locks and its respective resource in at most N calls to `trylock()`, thus resulting in a wait-free bounded progress. However, if the `trylock()` has spurious failures, then N resources may not be enough, and either an unbounded amount of reader-writer lock instances (and resources) would be required, which is clearly unrealistic, or a thread reaching the *last* instance without having successfully acquired a lock must restart from the *first* instance, thus resulting in lock-free progress, and prone to starvation.

2. RWTRYLOCK-DF

Our first algorithm with deadlock freedom for trylocks is RWTryLock-DF and it requires a single atomic word, which we named `state`. A `state` with a value of zero (`NOLOCK`) means the lock is not held by any thread. When the `state` has a value between zero and `WLOCK` it means that the lock is being held in *shared* mode by one or multiple threads (readers). If the `state` has a value of `WLOCK` or higher, then the lock is being held in *exclusive* mode by a single thread (writer). The corresponding C++ source code is shown in Algorithm 2.

To acquire the lock in shared mode, a reader will increase the value of the `state` by 1 using a Fetch-And-Add (FAA) operation (line 9). The FAA will return the current value of `state`, and if it is already at `WLOCK` or higher, then the lock is in exclusive mode and the shared attempt must fail. Notice that no rollback of the value is needed because when the writer releases the lock, it will set the value of `state` back to zero (`NOLOCK`). When the reader wants to release its shared lock, it will decrement the `state` by 1 using an FAA (line 13). In practice, this means that all bits below the `WLOCK` bit are being used to count the number of readers currently attempting to or holding the lock in shared mode.

To acquire the lock in exclusive mode, a writer will attempt to do a Compare-And-Swap (CAS) from `NOLOCK` to `WLOCK` (line 19), which will fail if there are any readers that have incremented `state` or if another writer was successful in its CAS. To release the exclusive mode, the state will be set to `NOLOCK` (line 23). It is possible for a lock in exclusive mode to be *downgraded* to shared mode by doing a store on `state` of `1` (line 27) to indicate the presence of a single reader.

The `sharedLock()` and `exclusiveLock()` methods can be implemented by doing a spin-loop with `sharedTryLock()` and `exclusiveTryLock()` respectively.

Although excellent for deployments where memory is scarce, or when a large number of lock instances are needed, the RWTryLock-DF algorithm has little scalability for read-mostly workloads due to contention on the `state` variable caused by the readers. The RWTryLock-DF-ReaderState algorithm in the next section addresses this issue through the usage of a modified ReadIndicator where each reader has its own separate state.

Algorithm 2 RWTryLock-DF in C++

```
1  class RWTryLockDF {
2    static const int64_t NOLOCK = 0;
3    static const int64_t WLOCK = 1LL << 62;
4    std::atomic<int64_t> state {NOLOCK};
5
6  public:
7    bool sharedTryLock() {
8      if (state.load() >= WLOCK) return false;
9      return (state.fetch_add(1)+1 < WLOCK);
10   }
11
12   void sharedUnlock() {
13     state.fetch_add(−1);
14   }
15
16   bool exclusiveTryLock() {
17     if (state.load() != NOLOCK) return false;
18     int64_t unlocked = NOLOCK;
19     return state.compare_exchange_strong(unlocked, WLOCK);
20   }
21
22   void exclusiveUnlock() {
23     state.store(NOLOCK, std::memory_order_release);
24   }
25
26   void downgrade() {
27     state.store(1, std::memory_order_release);
28   }
29 };
```

3. RWTRYLOCK-DF-READERSTATE

To help improve the scalability on the readers side, we

present a new algorithm named RWTryLock-DF-ReaderState which uses a concept similar to ReadIndicators [2], where each reader needs to publish its state independently, in our implementation, an exclusive entry in an array whose index is given by the thread's id `tid`.

Algorithm 3 RWTryLock-DF-ReaderState in C++

```
1  class RWTryLockDFReaderState {
2    const int maxThreads;
3    static const int64_t NOLOCK = -1;
4    static const int64_t RLOCK = -2;
5    static const int64_t WLOCK = 1LL << 62;
6    std::atomic<int> wstate {NOLOCK};
7    RIStaticPerThread ri {maxThreads};
8
9  public:
10   bool sharedTryLock(const int tid) {
11     if (wstate.load() == WLOCK) return false;
12     ri.arrive(tid);
13     int64_t ws = wstate.load();
14     if (ws != NOLOCK && ws != WLOCK && ws != RLOCK) {
15       if (wstate.compare_exchange_strong(ws, NOLOCK)) {
16         return true;
17       }
18       ws = wstate.load();
19     }
20     return (ws != WLOCK || !ri.rollbackArrive(tid));
21   }
22
23   void sharedUnlock(const int tid) {
24     ri.depart(tid);
25   }
26
27   bool exclusiveTryLock(const int tid) {
28     if (!ri.isEmpty()) return false;
29     int64_t ws = wstate.load();
30     if (ws == RLOCK || ws == WLOCK) return false;
31     if (ws != tid) {
32       if (!wstate.compare_exchange_strong(ws, tid)) return false;
33     }
34     if (!ri.isEmpty()) return false;
35     int64_t curr = tid;
36     return wstate.compare_exchange_strong(curr, WLOCK);
37   }
38
39   void exclusiveUnlock() {
40     wstate.store(RLOCK);
41     ri.abortRollback();
42     wstate.store(NOLOCK);
43   }
44
45   void downgrade(const int tid) {
46     wstate.store(RLOCK);
47     ri.abortRollback();
48     ri.arrive(tid);
49     wstate.store(NOLOCK);
50   }
51 };
```

Our ReadIndicator implementation in Algorithm 4 has two extra methods: `rollbackArrive()` called by a reader to do a rollback of the `arrive()`; `abortRollbacks()` called by a writer to abort all ongoing rollbacks of the readers;

Each RWTryLock-DF-ReaderState instance contains one `wstate` variable and one ReadIndicator instance, as shown in lines 6 and 7 of Algorithm 3. The `wstate` variable can have different values:

- `NOLOCK` when no thread is currently holding the lock, or if one or multiple threads have the lock in shared mode;
- `Thead's id` if a writer is attempting to acquire the lock in exclusive mode;
- `WLOCK` when there is a writer holding the lock in exclusive mode;
- `RLOCK` when a writer is currently releasing the lock from exclusive mode;

A schematic with the state transitions for `wstate` is shown in Figure 1.

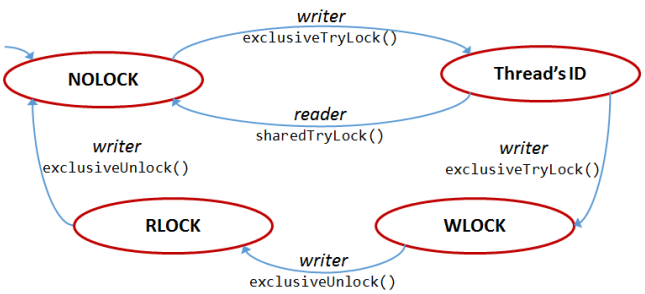

Figure 1: State-machine of the `wstate` variable

A unique intermediate state is needed for each writer, which is achieved with the thread's id, `TID`. The intermediate state must be unique so as to prevent ABA issues when transitioning from the intermediate state to the final state `WLOCK`, hence the usage of the thread's id. For example, if the intermediate states were not unique per writer, then a reader could change `wstate` from the intermediate state to `NOLOCK` and a second writer change it back to intermediate, and the first writer do a successful CAS from intermediate to `WLOCK`, thus nullifying the transition to `NOLOCK` by the reader, due to ABA. Such behavior could invalidate mutual exclusivity and therefore be incorrect.

Whether the lock is being held in shared mode or not depends on the combination of the `wstate` variable and the value of `ri.isEmpty()`. Figure 2 shows the states of a single reader in our modified ReadIndicator.

The `RLOCK` state is meant to let the lock be acquired in shared mode but not in exclusive mode. During this state, the writer currently holding the lock will attempt to prevent ongoing readers from doing a rollback by calling `ri.abortRollbacks()`. This method will let the readers know that the lock is free to be taken in shared mode. Without the `RLOCK` state and the `rollbackArrive()` and `abortRollbacks()` API, it could happen that a reader would do `arrive()`, this would prevent any further writers from taking the lock, the reader would see `wstate` as `WLOCK` and decide to call `ri.rollbackArrive()` but gets preempted before doing so, and then the writer holding the lock would release

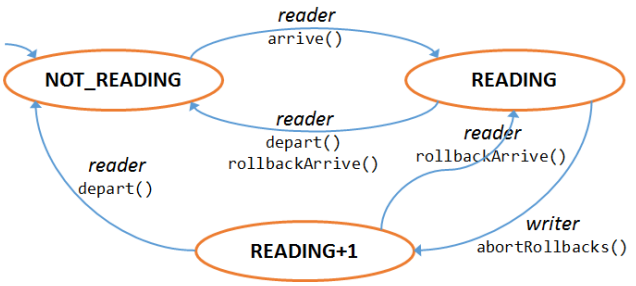

Figure 2: States of each individual reader in our ReadIndicator

it by setting `wstate` back to `NOLOCK`, and eventually the reader would complete its call to `ri.rollbackArrive()`, all the while neither this reader nor no other writer would be able to acquire the lock, which would result in a `trylock()` *spurious failure*. Without such mechanism, it does not seem possible to use multiple variables (one per reader) to represent the reader's states, and without such an approach, there is no way to obtain scalability for read-mostly workloads. Reader-writer locks with good read scalability, like C-RW-RP, have multiple variables in different cache lines to represent the reader's states, so as to spread contention over these cache lines, as opposed to have all contention in a single cache line like RWTryLock-DF. However, using different cache lines means we no longer have *atomicity* for the reader's and writer's states, and a more complex state machine was devised to handle this case in RWTryLock-DF-ReaderState.

Regarding the `downgrade()` method in RWTryLock-DF-ReaderState, a naive approach could assume that calling `ri.arrive()` followed by setting `wstate` to `NOLOCK` would be sufficient, though sadly it is not. It could happen that an ongoing reader would decide to rollback its ReadIndicator because it saw `wstate` as being `WLOCK`, then the writer downgrades and unlocks (the shared lock). If a new writer attempts to acquire the lock it will fail because there is still an ongoing reader (about to rollback), which means that it will not acquire the lock even though no thread has the lock, a behavior which is not deadlock free. The correct solution is shown in lines 45-50 of Algorithm 3, where writers are prevented from taking the lock by setting it to `RLOCK` and then reader's rollbacks are aborted to make sure that any ongoing reader rollback will not prevent a future writer from taking the lock, and afterwards we can follow the procedure similar to a reader, by doing `ri.arrive()` and moving `wstate` to `NOLOCK`.

3.1 RWTryLock-DF-ReaderState Correctness

For exclusive mode, the correctness of a reader-writer trylock is satisfied when the guarantee is given that a writer thread can only access its critical section in isolation, thus making sure no other thread has the lock. For

Algorithm 4 Our ReadIndicator implementation

```
1  class RIStaticPerThread {
2    static const uint64_t NOT_READING = 0;
3    static const uint64_t READING = 1;
4          std::atomic<uint64_t> states[MAX_THREADS];
5
6  public:
7    RIStaticPerThread(int maxThreads) : maxThreads{maxThreads} {
8      for (int tid = 0; tid < maxThreads; tid++) {
9        states[tid].store(NOT_READING, std::memory_order_relaxed);
10     }
11   }
12
13   void arrive(const int tid) noexcept {
14     states[tid].store(READING);
15   }
16
17   void depart(const int tid) noexcept {
18     states[tid].store(NOT_READING);
19   }
20
21   bool isEmpty() noexcept {
22     for (int tid = 0; tid < maxThreads; tid++) {
23       if (states[tid].load() != NOT_READING) return false;
24     }
25     return true;
26   }
27
28   void abortRollbacks() noexcept {
29     for (int tid = 0; tid < maxThreads; tid++) {
30       if (states[tid].load() != READING) continue;
31       uint64_t read = READING;
32       states[tid].compare_exchange_strong(read, READING+1);
33     }
34   }
35
36   bool rollbackArrive(const int tid) noexcept {
37     return (states[tid].fetch_add(-1) == READING);
38   }
39 };
```

shared mode, a reader thread can only acquire the lock if no writer thread has it, but it can share its critical section with other readers.

From Algorithm 3, we can see that a writer will check the reader's state and give up if any reader has already arrived (line 28 of Algorithm 3). It will also give precedence to readers even after marking `wstate` with its intermediate state, `TID` (line 32), which gives a read preference to this algorithm. On the other hand, readers after arriving, will make sure there is no writer that has already grabbed the lock or is in the process of acquiring it, with `wstate` marked as `TID` (line 14). In case the lock is not yet in exclusive mode, readers will try to make sure the writer will not have the lock, just in case the writer has not seen its arrival. The CAS on `wstate` (line 15) guarantees that only one, reader or writer, will get the lock, making sure only a writer or multiple readers access the critical section. This guarantees correctness for the trylock methods but unfortunately does not guarantee that in all scenarios for all threads attempting to acquire the lock there will always be at least one thread that will acquire it successfully.

The property of deadlock freedom is only satisfied

with the extra state `RLOCK` and the functions `rollbackArrive()` and `abortRollbacks()`. Without the `RLOCK` state, the scenario that invalidates the deadlock freedom property occurs when both reader and writer are competing for a lock, and the writer wins the lock. At this point, the reader will abort and rollback the `arrive()`, but before doing so, the writer does unlock. Now, another writer, or even the same writer thread, attempts to acquire the lock but will immediately give up because it sees a reader marked as arrived and immediately after, the reader will conclude the abort. In such a scenario, both reader and writer will not acquire the lock and therefore, no thread was successful. The solution is for every writer that acquires the lock to guarantee, before releasing the lock, that ongoing readers will not abort and mistakenly cause future concurrent writes to give up acquiring the lock. This is achieved with a transitional state, `RLOCK`, which will allow readers to enter but no new writer. Also, the writer which is in the process of releasing the lock, will have to abort all possible reader's rollbacks that may haven't seen the transitional state `RLOCK`, by marking all reader's states to a state that will make `rollbackArrive()` fail. After calling `abortRollbacks()`, the writer guarantees that no reader will abort due to its write lock. In a way, every write lock will first downgrade the lock to read mode before releasing the lock, again giving reader preference to the algorithm.

4. MICROBENCHMARKS

To compare the throughput of our reader-writer lock algorithms, we executed a microbenchmark on an AMD Opteron 6272 server with a total of 32 cores, running Ubuntu with GCC 6.2.0, using a procedure similar to [2] except that we have multiple reader-writer lock instances each protecting its own array.

In our microbenchmark there is an array of reader-writer locks whose size equals the number of competing threads. Similarly to [2], each reader-writer lock protects an array of 64 integers. The read-only operation iterates through an inner loop for `RCSLen=4` times where each iteration fetches two randomly selected integers from the shared array. The read-write operation iterates through an inner loop for `WCSLen=4` times where each iteration selects two integers from the shared array, and adds a random value to one integer and subtracts that same value from the other integer. The non-critical section of the main loop updates another thread-private array of 64 integers for `NCSLen=32` iterations.

Unlike [2], in our microbenchmark each thread will attempt to acquire the lock on the first instance using the respective `trylock()` method, `sharedTryLock()` for a read-only operation or `exclusiveTryLock()` for a read-write operation. If the `trylock()` returns false, then it will try on the next instance, and repeat this procedure until the last instance is reached. If the last reader-writer lock instance is reached and the `trylock()` fails, then we consider this as being an *overflow* and continue the attempts from the first instance, repeating the procedure until the `trylock()` is successful.

We executed this procedure for four different reader-writer locks: PThread, the implementation in the pthreads library; C-RW-RP, the algorithm based on C-RW-RP [2] shown in Algorithm 1; RWTryLock-DF, Algorithm 2; RWTryLock-DF-ReaderState, Algorithm 3. The results are shown in figure 3, where each data point is the median of 5 runs of the procedure described above. RWTryLock-DF-ReaderState has good scalability under read-mostly workloads, matching C-RW-RP which has been until now the best in this class.

Of the four different implementations, only the C-RW-RP has overflows, with the other three implementations having zero overflows, i.e. do not have spurious trylocks failures. The pthread implementation in our system uses a single mutex which readers and writers have to acquire, thus creating a linearization point for the trylock methods, but other pthread implementations in other systems may behave differently.

5. CONCLUSION

The `trylock()` methods for reader-writer lock algorithms are usually an afterthought and receive little consideration by most lock designers, with few even explicitly showing the algorithm for these methods in their papers. The importance of this API to practitioners and users of reader-writer locks seems to have been underrated until now, and support is provided on a best effort basis. However, the `trylock()` methods of a reader-writer lock can be used as a building block to wait-free pools and other more complex wait-free bounded techniques, but only if these methods are provided with deadlock freedom, i.e. without spurious failures.

We have shown it is simple to create a reader-writer lock algorithm with a single word of memory, with non-spurious failures for `trylock()`, using CAS and FAA atomic operations, namely RWTryLock-DF. However, its scalability for read-mostly workloads leaves much to be desired and is far below state of the art reader-writer lock algorithms whose trylocks do not provide deadlock freedom like C-RW-RP [2]. To address this scalability issue we designed RWTryLock-DF-ReaderState, a reader-writer lock algorithm capable of providing the same kind of scalability as C-RW-RP, with linearizable consistency and deadlock freedom for all methods including `trylock()`s.

6. REFERENCES

[1] Brandenburg, B. B., and Anderson, J. H. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems 46*, 1 (2010), 25–87.

[2] Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V. J., and Shavit, N.

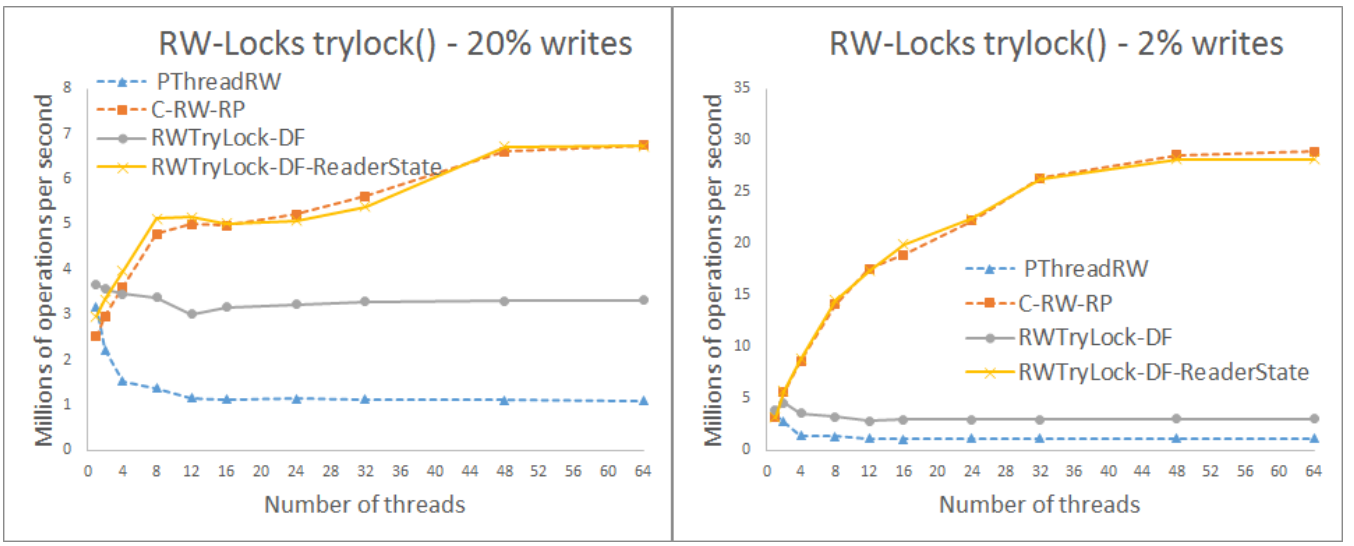

Figure 3: Benchmarks with 20% and 2% writes. C-RW-RP has overflows.

NUMA-aware reader-writer locks. *PPoPP 2013* (2013).

[3] Courtois, P.-J., Heymans, F., and Parnas, D. L. Concurrent control with âĂIJreadersâĂİ and âĂIJwritersâĂİ. *Communications of the ACM 14*, 10 (1971), 667–668.

[4] CPP-ISO-committee. std::mutex::try_lock. http://en.cppreference.com/w/cpp/thread/mutex/try_lock, 2016.

[5] Dice, D., and Shavit, N. Tlrw: return of the read-write lock. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures* (2010), ACM, pp. 284–293.

[6] Hsieh, W. C., and Weihl, W. E. Scalable reader-writer locks for parallel systems. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International* (1992), IEEE, pp. 656–659.

[7] Krieger, O., Stumm, M., Unrau, R., and Hanna, J. A fair fast scalable reader-writer lock. In *Parallel Processing, 1993. ICPP 1993. International Conference on* (1993), vol. 2, IEEE, pp. 201–204.

[8] Lev, Y., Luchangco, V., and Olszewski, M. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (2009), ACM, pp. 101–110.

[9] Mellor-Crummey, J. M., and Scott, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Notices* (1991), vol. 26, ACM, pp. 106–113.

[10] Mellor-Crummey, J. M., and Scott, M. L. Synchronization without contention. *ACM SIGPLAN Notices 26*, 4 (1991), 269–278.