TWA – Ticket Locks Augmented with a Waiting Array

Dave Dice(✉) and Alex Kogan

Oracle Labs, Burlington, MA, USA
{dave.dice,alex.kogan}@oracle.com

Abstract. The classic *ticket lock* is simple and compact, consisting of `ticket` and `grant` fields. Arriving threads atomically fetch-and-increment `ticket` to obtain an assigned ticket value, and then wait for `grant` to become equal to that value, at which point the thread holds the lock. The corresponding unlock operation simply increments `grant`. This simple design has short code paths and fast handover (transfer of ownership) under light contention, but may suffer degraded scalability under high contention when multiple threads busy wait on the `grant` field – so-called *global spinning*.

We propose a variation on ticket locks where long-term waiting threads – those with an assigned `ticket` value far larger than `grant` – wait on locations in a *waiting array* instead of busy waiting on the `grant` field. The single waiting array is shared among all locks. Short-term waiting is accomplished in the usual manner on the `grant` field. The resulting algorithm, TWA, improves on ticket locks by limiting the number of threads spinning on the `grant` field at any given time, reducing the number of remote caches requiring invalidation from the store that releases the lock. In turn, this accelerates handover, and since the lock is held throughout the handover operation, scalability improves. Under light or no contention, TWA yields performance comparable to the classic ticket lock. Under high contention, TWA is substantially more scalable than the classic ticket lock, and provides performance on par or beyond that of scalable queue-based locks such as MCS by avoiding the complexity and additional accesses incurred by the MCS handover operation while also obviating the need for maintaining queue elements.

We provide an empirical evaluation, comparing TWA against ticket locks and MCS for various user-space applications, and within the Linux kernel.

Keywords: Locks · Mutexes · Mutual exclusion · Synchronization · Concurrency control

1 Introduction

The classic ticket lock [16,17] is compact and has a very simple design. The acquisition path requires only one atomic operation – a fetch-and-add to increment the ticket – and the unlock path requires no atomics. Under light or no

© Springer Nature Switzerland AG 2019
R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 334–345, 2019.
https://doi.org/10.1007/978-3-030-29400-7_24

contention, the handover latency, defined as the time between the call to unlock and the time a successor is enabled to enter the critical section, is low. Handover time impacts the scalability as the lock is held throughout handover, increasing the effective length of the critical section [11]. A ticket lock is in *unlocked* state when `ticket` and `grant` are equal. Otherwise the lock is held, and the number of waiters is given by `ticket - grant - 1`. Ignoring numeric rollover, `grant` always lags or is equal to `ticket`. The increment operation in unlock either passes ownership to the immediate successor, if any, and otherwise sets the state to unlocked.

Ticket locks suffer, however, from a key scalability impediment. All threads waiting for a particular lock will busy wait on that lock's `grant` field. An unlock operation, when it increments `grant`, invalidates the cache line underlying `grant` for all remote caches where waiting threads are scheduled. In turn, this negatively impacts scalability by retarding the handover step. Ticket locks use global spinning, as all waiting threads monitor the central lock-specific `grant` variable.

In Fig. 1 we show the impact of readers on a single writer. We refer to the number of participating caches as the *invalidation diameter* [8]. The `Invalidation Diameter` benchmark spawns T concurrent threads, with T shown on the X-axis. A single writer thread loops, using an atomic fetch-and-add primitive to update a shared location. The other $T-1$ threads are readers. They loop, fetching the value of that location. The shared variable is sequestered to avoid false sharing and is the sole occupant of its underlying cache sector. We present the throughput rate of the writer on the Y-axis. As we increase the number of concurrent readers, the writer's progress is slowed. This scenario models the situation in ticket locks where multiple waiting threads monitor the `grant` field, which is updated by the current owner during handover. The benchmark reports the writer's throughput at the end of a 10 s measurement interval. The data exhibited high variance due to the NUMA placement vagaries of the threads and the home node of the variable. As such, for each data point show, we took the median of 100 individual runs, reflecting a realistic set of samples. The system-under-test is described in detail in Sect. 4.

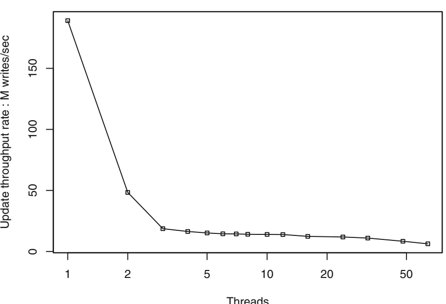

Fig. 1. Invalidation diameter

The *MCS lock* [16] is the usual alternative to ticket locks, performing better under high contention, but also having a more complex path and often lagging behind ticket locks under no or light contention. In MCS, arriving threads use an atomic operation to append an element to a queue of waiting threads, and then busy wait on a field in that element. The lock's tail variable is explicit and the head – the current owner – is implicit. When the owner releases the lock it reclaims the element it originally enqueued and sets the flag in the next element, passing ownership. Specifically, to convey ownership, the MCS unlock operator

must identify the successor, if any, and then store to the location where the successor busy waits. The handover path is longer than that of ticket locks and accesses more distinct shared locations. MCS uses so-called local waiting where at most one thread is waiting on a given location at any one time. As such, an unlock operation will normally need to invalidate just one location – the flag where the successor busy waits. Under contention, the unlock operator must fetch the address of the successor node from its own element, and then store into the flag in the successor's element, accessing two distinct cache lines, and incurring a dependent access to reach the successor. In the case of no contention, the unlock operator must use an atomic compare-and-swap operator to detach the owner's element.

Ticket locks and TWA require no such indirection or dependent accesses in the unlock path and also avoid the need for queue elements and the management thereof. The queue of waiting threads is implicit in ticket locks and TWA, and explicit in MCS. MCS, ticket locks and TWA all provide strict FIFO admission order.

Ticket locks are usually a better choice under light or no contention, while MCS locks are more suitable under heavy contention [2,3]. By employing a waiting array for long-term waiting, TWA achieves the best of the two worlds, as demonstrated by our empirical evaluation with multiple user-space applications and within the Linux kernel.

2 The TWA Algorithm

TWA builds directly on ticket locks. We add a new *waiting array* for long-term waiting. The array is shared amongst all threads and TWA locks in an address space. Arriving threads use an atomic fetch-and-increment instruction to advance the `ticket` value, yielding the lock request's assigned ticket value, and then fetch `grant`. If the difference is 0 then we have uncontended acquisition and the thread may enter the critical section immediately. (This case is sometimes referred to as the lock acquisition *fast-path*). Otherwise TWA compares the difference to the `LongTermThreshold` parameter. If the difference exceeds `LongTermThreshold` then the thread enters the long-term waiting phase. Otherwise control proceeds to the short-term waiting phase, which is identical to that of normal ticket locks; the waiting thread simply waits for `grant` to become equal to the ticket value assigned to the thread. While `LongTermThreshold` is a tunable parameter in our implementation, we found a value of 1 to be suitable for all environments, ensuring that only the immediate successor waits in short-term mode. All data reported below uses a value of 1.

A thread entering the long-term waiting phase first hashes its assigned ticket value to form an index into the waiting array. Using this index, it then fetches the value from the array and then recheck the value of `grant`. If the observed `grant` value changed, it rechecks the difference between that new value and its assigned ticket value, and decides once again on short-term versus long-term waiting. If `grant` was unchanged, the thread then busy waits for the waiting

array value to change, at which point it reevaluates `grant`. When `grant` is found to be sufficiently near the assigned ticket value, the thread reverts to normal short-term waiting. The values found in the waiting array have no particular meaning, except to conservatively indicate that a `grant` value that maps to that index has changed, and rechecking of `grant` is required for waiters on that index.

The TWA unlock operator increments `grant` as usual from U to $U + 1$ and then uses an atomic operator to increment the location in the waiting array that corresponds to threads waiting on ticket value $U + 1 + LongTermThreshold$, notifying long-term threads, if any, that they should recheck `grant`. An atomic operation is necessary as the location may be subject to hash collisions. We observe that this change increases the path length in the unlock operator, but crucially the store that effects handover, which is accomplished by a non-atomic increment of `grant`, happens first. Given a `LongTermThreshold` value of 1, we expect at most one thread, the immediate successor, to be waiting on `grant`. Updating the waiting array occurs *after* handover and outside the critical section.

All our experiments use a waiting array with 4096 elements, although ideally, we believe the waiting array should be sized as a function of the number of CPUs in the system. Hash collisions in the table are benign, at worst causing unnecessary rechecking of the `grant` field. Our hash function is cache-aware and intentionally designed to map adjacent ticket values to different 128-byte cache sectors underlying the waiting array, to reduce false sharing among long-term waiters. We multiply the ticket value by 127, `EXCLUSIVE-OR` that result with the address of the lock, and then mask with $4096 - 1$ to form an index into the waiting array. We selected a small prime $P = 127$ to provide the equidistribution properties of a *Weyl sequence* [15]. We include the lock address into our deterministic hash to avoid the situation where two locks might operate in an entrained fashion, with ticket and grant values moving in near unison, and thus suffer from excessive inter-lock collisions. A given lock address and ticket value pair always hashes to the same index.

TWA leaves the structure of the ticket lock unchanged, allowing for easy adoption. As the instance size remains the same, the only additional space cost for TWA is the waiting array, which is shared over all locks, reflecting a one-time space cost.

The TWA fast-path for acquisition remains unchanged relative to ticket locks. The unlock path adds an increment of the waiting array, to notify long-term waiters, if any. that they should transition from long-term to short-term waiting. We note that TWA doesn't reduce overall coherence traffic, but does act to reduce coherence traffic in the critical handover path, constraining the invalidation diameter of the store in unlock that accomplishes handover. TWA thus captures the desirable performance aspects of both MCS locks and ticket locks.

Listing 1.1 depicts a pseudo-code implementation of the TWA algorithm. Lines 7 through 16 reflect the classic ticket lock algorithm and lines 20 through 71 show TWA. TWA extends the existing ticket lock algorithm by adding lines 41 through 57 for long-term waiting, and line 71 to notify long-term waiters to shift to classic short-term waiting.

Listing 1.1: Simplified Python-like Implementation of TWA

```
1  ## Classic Ticket Lock
2
3  class TicketLock :
4    int Ticket = 0         ## Next ticket to be assigned
5    int Grant  = 0         ## "Now Serving"
6
7  TicketAcquire (TicketLock * L) :
8    ## Atomic fetch-and-add on L.Ticket
9    auto tx = FetchAdd (L.Ticket, 1)
10   while tx != L.Grant :
11     Pause()
12
13 TicketRelease (TicketLock * L) :
14   ## succession via direct handoff ...
15   ## Increment does not require atomic instructions
16   L.Grant += 1
17
18 ## ==================================================
19
20 ## TWA : Ticket lock augmented with waiting array
21
22 ## tunable parameters
23 ## short-term vs long-term proximity threshold
24 LongTermThreshold = 1                                  |
25 ArraySize         = 4096                               |
26
27 ## Global variables :
28 ## Long-term waiting array, initially all 0
29 ## Shared by all locks and threads in the address space
30
31
32 uint64_t WaitArray [ArraySize]                         |
33
34 TWAAcquire (TWA * L) :
35   auto tx = FetchAdd (L.Ticket, 1)
36   auto dx = tx - L.Grant
37   if dx == 0 :
38     ## fast-path return - uncontended case
39     return
40
41   ## slow path with contention -- need to wait         |
42   ## Select long-term vs short-term based on the number |
43   ## of threads waiting in front of us                  |
44   if dx > LongTermThreshold :                          |
45     ## long-term waiting via WaitArray                 |
46     auto at = Hash(L, tx)                              |
47     for                                                |
48       auto u = WaitArray[at]                           |
49       dx = tx - L.Grant                                |
50       assert dx >= 0                                   |
51       if dx <= LongTermThreshold : break               |
52       while WaitArray[at] == u :                       |
53         Pause()                                        |
54     ## This waiting thread is now "near" the front of  |
55     ## the logical queue of waiting threads            |
56     ## Transition from long-term waiting to            |
57     ## short-term waiting                              |
58
59   ## classic short-term waiting on L.Grant field
60   while L.Grant != tx :
61     Pause()
62
63 TWARelease (TWA * L) :
64   ## Notify immediate successor, if any
65   ## such threads will be in short-term waiting phase
66   ## non-atomic increment
67   auto k = ++L.Grant
68
69   ## Notify long-term waiters                           |
70   ## atomic increment required                          |
71   FetchAdd (WaitArray[Hash(L,k + LongTermThreshold)], 1)|
```

2.1 Example Scenario – TWA in Action

❶ Initially the lock is in *unlocked* state with `Ticket` and `Grant` both 0.

❷ Thread *T1* arrives at Listing 1.1 line 34 attempting to acquire the lock. *T1* increments `Ticket` from 0 to 1, and the atomic `FetchAdd` operator returns the original value of 0 into the local variable `tx`, which holds the assigned ticket value for the locking request. At line 36 *T1* then fetches `Grant` observing a value of 0. Since `tx` equals that fetched value, we have uncontended lock acquisition. *T1* now holds the lock and can enter the critical section immediately, without waiting, via the fast path at line 39.

❸ Thread *T2* now arrives and tries to acquire the lock. The `FetchAdd` operator advances `Ticket` from 1 to 2 and returns 1, the assigned ticket, into `tx` at line 35. *T2* fetches `Grant` and notes that `tx` differs from that value by 1. The `dx` variable holds that computed difference, which reflects the number of threads between the requester and the head of the logical queue, which is the owner. *T2* has encountered contention and must wait. The difference is only 1, and *T2* will be the immediate successor, so *T2* proceeds to line 60 for short-term waiting similar to that used in classic ticket locks shown at line 10. *T2* waits for the `Grant` field to become 1.

❹ Thread *T3* arrives and advances `Ticket` from 2 to 3, with the `FetchAdd` operator returning 2 as the assigned ticket. The difference between that value (2) and the value of `Grant`(0) fetched at line 64 exceeds the `LongTermThreshold` (1), so *T3* enters the path for long-term waiting at line 49. *T3* hashes its observed ticket value of 2 into an index `at`, say 100, in the long-term waiting array and then fetches from `WaitArray[at]` observing U. To recover from potential races with threads in the unlock path, *T3* rechecks that the `Grant` variable remains unchanged (0) at line 49 and that the thread should continue with long-term waiting. Thread *T3* busy waits at lines 52–53 on the `WaitArray` value.

❺ Thread *T4* arrives, advances `Ticket` from 3 to 4, obtaining a value in its `tx` variable of 3. Similar to *T3*, *T4* enters the long-term. *T4* hashes its assigned ticket value of 3 yielding an index of, say, 207, and fetches `WaitArray[207]` observing V. *T4* then busy waits, waiting for `WaitArray[207]` to change from V to any other value.

❻ Thread *T1* now releases the lock, calling `TWARelease` at line 63. *T1* increments `Grant` from 0 to 1 at line 67, passing ownership to *T2* and sets local variable `k` to the new value (1).

❼ Thread *T2* waiting at lines 60–61 notices that `Grant` changed to match its `tx` value. *T2* is now the owner and may enter the critical section.

❽ Thread *T1*, still in `TWARelease` at line 71 then hashes $k + LongTermThreshold$ (the sum is 2) to yield index 100 and then increments `WaitArray[100]` from U to $U + 1$.

❾ Thread *T3* waiting at lines 52–53 observes that change, rechecks `Grant`, sees that it is close to being granted ownership, exits the long-term waiting loop and switches to classic short-term waiting at lines 60–61. *T1* has promoted *T3* from long-term to short-term waiting in anticipation of the next unlock operation, to eventually be performed by *T2*.

(10) Thread *T1* now exits the `TWARelease` operator.

(11) Thread *T2* is the current owner, thread *T3* is waiting in short-term mode, and thread *T4* is waiting in long-term mode.

3 Related Work

Mellor-Crummey and Scott [16] proposed ticket locks with *proportional backoff*. Waiting threads compare the value of their ticket against the `grant` field. The difference reflects the number of intervening threads waiting. That value is then multiplied by some tunable constant, and the thread delays for that period before rechecking `grant`. The constant is platform- and load-dependent, and requires tuning. While this approach may decrease the futile polling rate on `grant`, it does not decrease the invalidation diameter. TWA and ticket locks with proportional backoff both make a distinction among waiting threads based on their relative position in the queue.

Partitioned Ticket Locks [9] augment each ticket lock with a constant-length private array of `grant` fields, allowing for *semi-local waiting*. Critically, the array is not shared between locks, and to avoid false sharing within the array, the memory footprint of each lock instance is significantly increased. Anderson's array-based queueing lock [1] is also based on ticket locks. It employs a waiting array for each lock instance, sized to ensure there is at least one array element for each potentially waiting thread, yielding a potentially large footprint. The maximum number of participating threads must be known in advance when initializing the array. Such dynamic sizing also makes static allocation of Anderson's locks more difficult than would be the case for a lock with a fixed size, such as TWA.

Various authors [2,12] have suggested switching adaptively between MCS and ticket locks depending on the contention level. While workable, this adds considerable algorithmic complexity, particularly for the changeover phase, and requires tuning. Lim [13] suggested a more general framework for switching locks at runtime.

4 Empirical Evaluation

Unless otherwise noted, all data was collected on an Oracle X5-2 system. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30 GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total. The system was running Ubuntu 18.04 with a stock Linux version 4.15 kernel, and all software was compiled using the provided GCC version 7.3 toolchain at optimization level "-O3". 64-bit C or C++ code was used for all experiments. Factory-provided system defaults were used in all cases, and Turbo mode [18] was left enabled. In all cases default free-range unbound threads were used. TWA is trivial to implement in C++ with `std::atomic<>` primitives.

We implemented all user-mode locks within LD_PRELOAD interposition libraries that expose the standard POSIX `pthread_mutex_t` programming interface. The framework was made available by Dice et al. [10]. This allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses locks. The C++ `std::mutex` construct maps directly to `pthread_mutex` primitives, so interposition works for both C and C++ code. All busy-wait loops used the Intel `PAUSE` instruction for polite waiting.

We use a 128 byte sector size on Intel processors for alignment to avoid false sharing. The unit of coherence is 64 bytes throughout the cache hierarchy, but 128 bytes is required because of the adjacent cache line prefetch facility where pairs of lines are automatically fetched together.

4.1 MutexBench

The MutexBench benchmark spawns T concurrent threads. Each thread loops as follows: acquire a central lock L; execute a critical section; release L; execute a non-critical section. At the end of a 10 s measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. We show the median of 5 independent runs in Fig. 2. The critical section advances a C++ `std::mt19937` pseudo-random generator (PRNG) 4 steps. The non-critical section uses that same PRNG to compute a value distributed uniformly in $[0, 200)$ and then advances the PRNG that many steps. To facilitate comparison of the algorithms, the X-axis is logarithmic and the Y-axis is offset to the minimum score.

As seen in the figure, ticket locks performs the best up to 6 threads, with TWA lagging slightly behind. As we further increase the thread count, however, ticket locks fail to scale. MCS provides stable asymptotic performance that surpasses ticket locks at 24 threads. TWA manages to always outperform MCS, freeing the developer from making a choice between MCS locks and ticket locks.

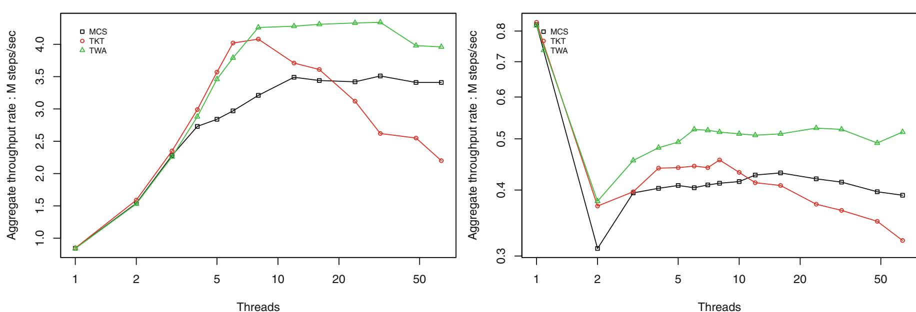

Fig. 2. MutexBench

Fig. 3. throw

4.2 throw

The “throw” benchmark launches T threads, each of which loop, executing the following line of C++ code:

```
try { throw 20 ;} catch (int e) {}.
```

Naively, this construct would be expected to scale linearly, but the C++ runtime implementation acquires mutexes that protect the list of dynamically loaded modules and their exception tables. The problem is long-standing and has proven difficult to fix given the concern that some applications might have come to depend on the serialization[1]. At the end of a 10 s measurement interval the benchmark reports the aggregate number of loops executed by all threads. Throw-catch operations are performed back-to-back with no intervening delay. In Fig. 3 we observe that performance drops significantly between 1 and 2 threads. There is little or no benefit from multiple threads, given that execution is largely serialized, but coherent communication costs are incurred. As we increase beyond two threads performance improves slightly, but never exceeds that observed at one thread. Beyond 2 threads, the shape of the graph recapitulates that seen in MutexBench.

4.3 libslock stress_latency

Figure 4 shows the performance of the “stress latency” benchmark from [7][2]. The benchmark spawns the specified number of threads, which all run concurrently during a 10 s measurement interval. Each thread iterates as follows: acquire a central lock; execute 200 loops of a delay loop; release the lock; execute 5000 iterations of the same delay loop. The benchmark reports the total number of iterations of the outer loop.

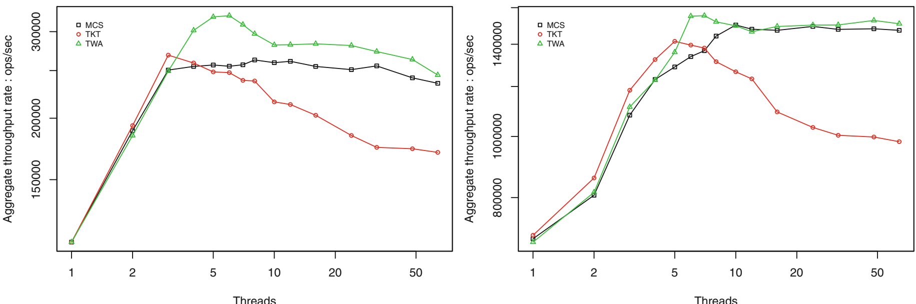

Fig. 4. libslock stress_latency

Fig. 5. LevelDB readrandom

[1] https://patchwork.ozlabs.org/patch/652301/.

[2] We use the following command line: ./stress_latency -l 1 -d 10000 -a 200 -n *threads* -w 1 -c 1 -p 5000.

4.4 LevelDB readrandom

In Fig. 5 we used the "readrandom" benchmark in LevelDB version 1.20 database[3] varying the number of threads and reporting throughput from the median of 5 runs of 50 s each. Each thread loops, generating random keys and then trying to read the associated value from the database. We first populated a database[4] and then collected data[5]. We made a slight modification to the `db_bench` benchmarking harness to allow runs with a fixed duration that reported aggregate throughput. Ticket locks exhibit a very slight advantage over MCS and TWA at low threads count after which ticket locks fade and TWA matches or exceeds the performance of MCS. LevelDB uses coarse-grained locking, protecting the database with a single central mutex: `DBImpl::Mutex`. Profiling indicates contention on that lock via `leveldb::DBImpl::Get()`.

4.5 RocksDB readwhilewriting

We next present results in Fig. 6 from the RocksDB[6] version 5.14.2 database running the "readwhitewriting" benchmark which has one fixed writer thread and a variable number of readers. The benchmark is similar to the form found in LevelDB, above, but the underlying database allows more concurrency and avoids the use of a single central lock. We intentionally use a command-line configured to stress the locks that protect the sharded LRU cache, causing contention in `LRUShard::lookup()`[7].

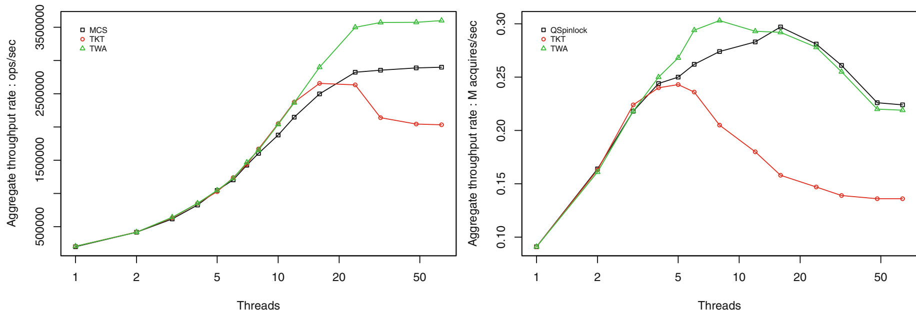

Fig. 6. RocksDB readwhilewriting

Fig. 7. LockTorture

[3] leveldb.org.

[4] db_bench —threads=1 —benchmarks=fillseq —db=/tmp/db/.

[5] db_bench —threads=*threads* —benchmarks=readrandom —use_existing_db=1 —db=/tmp/db/ —duration=50.

[6] rocksdb.org.

[7] db_bench —duration=200 —threads=*threads* —benchmarks=readwhilewriting —compression_type=none —mmap_read=1 —mmap_write=1 —cache_size=100000 —cache_numshardbits=0 —sync=0 —verify_checksum=0.

4.6 Linux Kernel `locktorture`

We ported TWA into the Linux kernel environment and evaluated its performance with the `locktorture` benchmark[8]. `Locktorture` is distributed as a part of the Linux kernel and is implemented as a loadable kernel module. The benchmark spawns a specified number of threads, each of which loops, contending for a central lock. We used locktorture to compare TWA, classic ticket locks, and the default kernel qspinlock.

The Linux *qspinlock* construct [4,5,14] is a compact 32-bit lock, even on 64-bit architectures. The low-order bits of the lock word constititue a simple test-and-set lock while the upper bits encode the tail of an MCS chain. In order to fit into a 32-bit work – a critical requirement – the chain is formed by logical CPU identifiers instead of traditional MCS queue node pointers. The result is a hybrid of MCS and test-and-set[9]. We note that qspinlocks replaced classic ticket locks as the kernel's primary low-level spin lock mechanism in 2014, and ticket locks replaced test-and-set locks, which are unfair and allow unbounded bypass, in 2008 [6].

The average critical section duration used by `locktorture` is a function of the number of concurrent threads. In order to use the benchmark to measure and report scalability, we augmented it to parameterize the critical and non-critical section durations, which are expressed as steps of the thread-local pseudo-random number generator provided in the `locktorture` infrastructure. We used 20 steps for the critical section. Each execution of the non-critical section computes a uniformly random distributed number in $[0 - 400)$ and then steps the local random number generator that many iterations. At the end of a run (lasting 30 s in our case), the total number of lock operations performed by all threads is reported. We report the median of 7 such runs in Fig. 7.

As we can see in Fig. 7, classic ticket locks perform well at low conconcurrency but fade as the number of threads increases. TWA performs the same or slightly better than qspinlock, although TWA is far simpler[10].

5 Conclusion

TWA is a straightforward extension to classic ticket locks, providing the best performance properties of ticket locks and MCS locks. Like ticket locks, it is simple, compact, and has a fixed memory footprint. The key benefit conferred by TWA arises from improved transfer of ownership (handover) in the unlock path, by reducing the number of threads spinning on the `grant` field at any given time. Even though TWA increases the overall path length in the unlock operation, adding an atomic fetch-and-increment operation compared to the classic ticket lock, it decreases the effective critical path duration for contended handover.

[8] https://www.kernel.org/doc/Documentation/locking/locktorture.txt.

[9] https://github.com/torvalds/linux/blob/master/kernel/locking/qspinlock.c.

[10] An extended version of this paper is available at https://arxiv.org/abs/1810.01573, where we apply various complexity measures to compare ticket locks, qspinlock, and TWA.

References

1. Anderson, T.E.: The performance of spin lock alternatives for shared-money multiprocessors. IEEE Trans. Parallel Distrib. Syst. (1990). https://doi.org/10.1109/71.80120
2. Antić, J., Chatzopoulos, G., Guerraoui, R., Trigonakis, V.: Locking made easy. In: Proceedings of the 17th International Middleware Conference, Middleware 2016. ACM (2016). http://doi.acm.org/10.1145/2988336.2988357
3. Boyd-Wickizer, S., Kaashoek, M.F., Morris, R., Zeldovich, N.: Non-scalable locks are dangerous. In: Ottawa Linux Symposium (OLS) (2012). https://www.kernel.org/doc/ols/2012/ols2012-zeldovich.pdf
4. Corbet, J.: Cramming more into struct page, 28 August 2013. https://lwn.net/Articles/565097. Accessed 01 Oct 2018
5. Corbet, J.: MCS locks and qspinlocks, 11 March 2014. https://lwn.net/Articles/590243. Accessed 12 Sept 2018
6. Corbet, J.: Ticket spinlocks, 6 February 2008. https://lwn.net/Articles/267968. Accessed 12 Sept 2018
7. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: SOSP (2013). http://doi.acm.org/10.1145/2517349.2522714
8. Dice, D.: Malthusian locks. In: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017 (2017). http://doi.acm.org/10.1145/3064176.3064203
9. Dice, D.: Brief announcement: a partitioned ticket lock. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011 (2011). http://doi.acm.org/10.1145/1989493.1989543
10. Dice, D., Marathe, V.J., Shavit, N.: Lock cohorting: a general technique for designing NUMA locks. ACM Trans. Parallel Comput. (2015). http://doi.acm.org/10.1145/2686884
11. Eyerman, S., Eeckhout, L.: Modeling critical sections in Amdahl's law and its implications for multicore design. In: ISCA. ACM (2010). http://doi.acm.org/10.1145/1815961.1816011
12. Ha, P.H., Papatriantafilou, M., Tsigas, P.: Reactive spin-locks: a self-tuning approach. In: 8th International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN 2005 (2005). https://doi.org/10.1109/ISPAN.2005.73
13. Lim, B.H., Agarwal, A.: Reactive synchronization algorithms for multiprocessors. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI. ACM (1994). http://doi.acm.org/10.1145/195473.195490
14. Long, W.: qspinlock: introducing a 4-byte queue spinlock implementation, 31 July 2013. https://lwn.net/Articles/561775. Accessed 19 Sept 2018
15. Marsaglia, G.: Xorshift RNGs. J. Stat. Softw. (2003). https://doi.org/10.18637/jss.v008.i14. https://www.jstatsoft.org/v008/i14
16. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. (1991). http://doi.acm.org/10.1145/103727.103729
17. Reed, D.P., Kanodia, R.K.: Synchronization with eventcounts and sequencers. Commun. ACM (1979). http://doi.acm.org/10.1145/359060.359076
18. Verner, U., Mendelson, A., Schuster, A.: Extending Amdahl's law for multicores with turbo boost. IEEE Comput. Arch. Lett. (2017). https://doi.org/10.1109/LCA.2015.2512982