Mutex Fairness Mitigation via Capacitors

Dave Dice
dave.dice@oracle.com
Oracle Labs
USA

Abstract

Classic test-and-and-set locks[6] are simple and exhibit low latency absent contention, and low handover latency under light contention, but may also admit long-term unfairness. Such unfairness can result in some threads starving, while other threads make progress resulting in a throughput disparity. This effect can persist over long periods. In turn, this can result in unpredictable performance, failure to adhere to service-level agreements (SLAs) and potential denial-of-service attacks [10, 38].

We present **Capacitors**, a novel technique that allows an existing unfair lock algorithm, such as test-and-set locks, to be "wrapped" by a simple and efficient synchronization construct that enforces fairness, while still preserving desirable properties of the existing unfair lock.

CCS Concepts: • **Software and its engineering** → *Multithreading*; *Mutual exclusion*; *Concurrency control*; *Process synchronization.*

Keywords: Synchronization; Locks; Mutual Exclusion; Mutex; Semaphores; Scalability; Cache-coherent Shared Memory

1 The Capacitor

The capacitor primitive exposes *arrive* and *depart* operations. The first B threads – where B is a tunable parameter – that call `arrive` will pass through the capacitor without blocking. After B have passed through `arrive`, the capacitor shifts to *closed* state where threads calling `arrive` will block in the capacitor. The capacitor remains closed until all B threads in the current *platoon* have called *depart*, in which case it re-opens to admit another platoon of B threads. If there are threads waiting in *arrive*, then the next platoon to be released must consist of the least recently arrived threads. Capacitors can be implemented by means of semaphores[23, 24, 32] coupled with one additional variable that tallies departures. We require that the admission order provided by the semaphore to be FIFO or first-in-first-enabled[1].

We then employ the capacitor to construct a composite lock as follows. Each existing lock, which we call an *interior lock* or *inner lock* is paired with a private capacitor instance. We say the capacitor "wraps" that lock. To acquire the composite lock, we invoke *arrive* on the capacitor, acquire the inner lock, execute the critical section, release the inner lock, and finally invoke *depart* on the capacitor.

Crucially, our new composite lock provides a strong *bounded bypass* bypass property, which confers both long-term fairness and avoids short-term starvation. Say thread $T1$ arrives to acquire the composite lock L. At most only $B-1$ other threads that arrived after $T1$ can acquire L before $T1$ manages to gain ownership of L, resulting in *B-1-bounded-bypass* property[1].

2 Implementation Details

```
struct CapacitorSimplistic {

  // Interior test-and-test-and-set lock body
  std::atomic<int> tts alignas(128) {0} ;

  // tunable "knob" ...
  static constexpr uint64_t BypassLimit = 10 ;

  // Capacitor ...
  std::atomic<int> Depart aligned(128) {0} ;
  std::atomic<uint64_t> SemaTicket {0} ;
  std::atomic<uint64_t> SemaGrant {BypassLimit} ;

  public :

  inline auto operator+(std::invocable auto && csfn) → void {
    // Arrive :
    auto tx = SemaTicket.fetch_add(1) ;
    while (SemaGrant.load(std::memory_order_acquire) <= tx) {
      Pause() ;
    }

    // Acquire interior TTS lock ...
    for (;;) {
      if (tts.exchange(1) == 0) break ;
      while (tts.load (std::memory_order_acquire) ≠ 0) {
        Pause() ;
      }
    }

    EnterCS :
    assert (tts.load() ≠ 0) ;
    csfn() ;
    assert (tts.load() ≠ 0) ;

    // Release interior TTS lock
    tts.store (0, std::memory_order_release) ;

    // Depart :
    auto v = Depart.fetch_add(1) + 1 ;
    assert (v >= 0 && v <= BypassLimit) ;
    // Invariant : Semaphore protects against v > BypassLimit
    // v == BypassLimit implies we are terminal thread in platoon
    // singleton -- no concurrency possible
    // Thus no need for atomic to update Depart
    // Can use simple store-release

    if (v == BypassLimit) {
      Depart.store (0, std::memory_order_release) ;
      SemaGrant.fetch_add (BypassLimit) ;
    }
  }
} ;
```

Listing 1. Capacitor - Simplistic

[1]bounded bypass is also referred to as *liveness*

In Listing-1 we show an implementation of the capacitor technique in modern C++. For brevity, this version expresses the critical section as a C++ lambda expression. Non-escaping lambdas are efficient and add no particular runtime overhead [2]. The `assert` statements in the listing express invariants and do not constitute checks against errant usage of the lock.

This particular implementation is not scalable as it utilizes a crude capacitor, but we use it to illustrate the operation of a capacitor wrapping a trivial test-and-set lock. Line-4 declares the test-and-set lock, `tts` while Lines 9-12 declare the associated capacitor. Lines-11 and -12 declare a simple non-scalable semaphore implemented as a variation on a classic ticket lock. The semaphore is initialized with B credits at Line-12. The `ByPassLimit` variable in the code corresponds to B. We also add a `Depart` variable which tallies the number of threads that have called the capacitor's *depart* primitive within the current platoon, allowing a means to determine when a platoon has completed and a new platoon should be enabled.

Lines 17-21 acquire the semaphore, waiting if necessary. Once acquired, our thread is part of the current platoon, and then proceeds to acquire the *inner* lock at Lines 23-29. Having acquired the inner lock, our thread enters and executes the criticdal section at Line-33, and then releases the inner lock at Line-37. Finally, our thread executes the *depart* operation on the semaphore, at Lines 39-51. At Line-40 it first atomically increments the `Departed` field. When that value reaches B (`BypassLimit`) then we know the capacitor must be in closed state. As only one thread can be the final thread to cause the tally to reach `ByPassLimit`, and the capacitor is closed, not concurrency is possible at this point. Our thread then resets `Departed` to 0, and enables the next platoon at Line-50, potentially waking up to B threads, if there are that many waiting. i A more scalable implementation is shown in Listing-2, which uses an improved semaphore.

3 Related Work

Mutual exclusion remains an active research topic [2–4, 9, 11, 13–15, 21, 25–28, 28–31, 37, 40–42].

Simple test-and-set or polite test-and-test-and-set [41] locks are compact and exhibit excellent latency for uncontended operations, but fail to scale and may allow unfairness and even indefinite starvation. Ticket Locks are compact and FIFO, and also have excellent latency for uncontended operations, but they also fail to scale because of global spinning, although some variations attempt to overcome this obstacle, at the cost of increased space [15, 39]. For instance, Anderson's array-based queueing lock [5, 6] is based on Ticket Locks but provides local spinning. 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 lock. TWA[15] is a variation on ticket locks that reduces the incidence of global spinning.

Queue-based locks such as MCS[36] or CLH[9, 34] are FIFO and provide local spinning and are thus more scalable. MCS is used in the linux kernel for the low-level "qspinlock" construct [7, 8, 33]. Modern extensions of MCS edit the queue order to make the lock *NUMA-Aware*[14]. MCS readily allows editing and re-ordering of the queue of waiting threads, [11, 14, 35]. Additional approaches that use global spinning include HemLock[17, 19] and Reciprocating Locks[18]. Queue-based locks usually do not perform as well as simple test-and-set locks absent contention, or at low contention levels.

A number of lock algorithms attempt to provide *concurrency restriction* – a form of admission control – to reduce, over the short term, the number of threads circulation over a lock, in order to avoid scalability collapse [11, 13, 16]. Such algorithms focus on improving scalability, and do not directly address fairness. The capacitor construction also provides concurrency restriction, as well as fairness and progress guarantees.

4 Empirical Performance Results

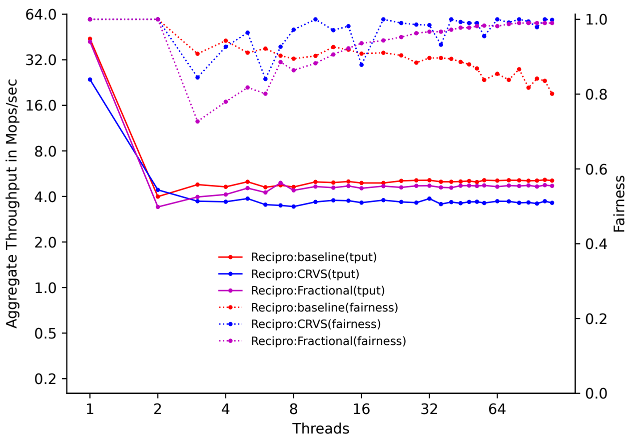
All data was collected on an ARMv8 Ampere Altra Max NeoVerse-N1 system with 128 processors on a single socket. The system was running Ubuntu 24.04. We compiled all code using the `-mno-outline-atomics -march=armv8.2-a+lse` flags in order to allow direct use of modern atomic `exchange`, `CAS` and `fetch-and-add` instructions instead of the legacy LL-SC (load-locked store-conditional) forms thereof. Factory-provided system defaults were used in all cases. In all cases default free-range unbound threads were used, with no pinning of threads to processors. All lock busy-wait loops used the ARMv8 `YIELD` instruction. In the listings, the `Pause` operation a single ARMv8 `YIELD` instruction.

4.1 MutexBench benchmark

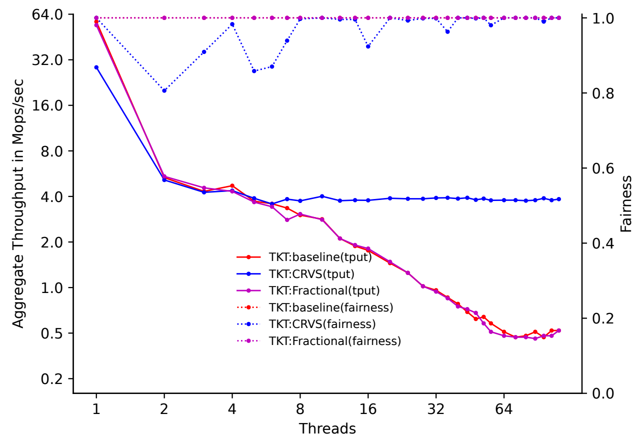
We used a simple locking microbenchmark, MutexBench, to examine the trade-off between capacitor overheads and fairness efficacy.

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. The critical section advances a shared global `std::mt19937` *Mersenne Twister* pseudo-random number generator (PRNG) one step, and the non-critical section is empty, subjecting the lock to extreme contention. (At just one thread, this configuration also constitutes a useful benchmark for uncontended latency).

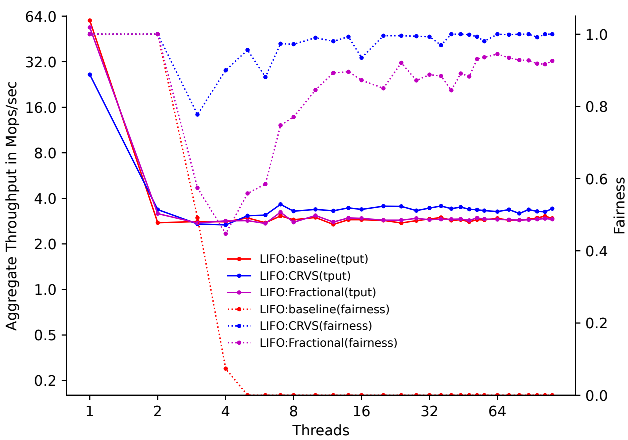
[2]Example usage : `CapacitorSimplistic L {}; int v = 5; L + [&]{ v += 2;} ;`



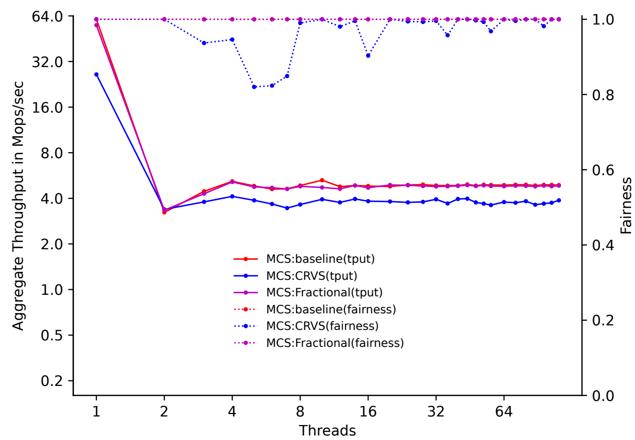
(a) Reciprocating Lock



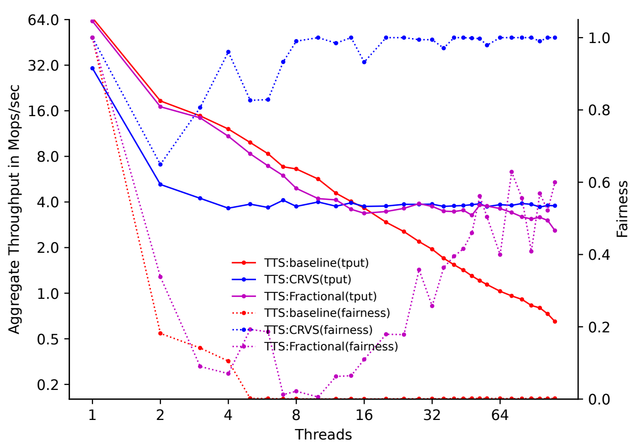
(b) Ticket Lock



(c) LIFO Lock



(d) MCS Lock



(e) TTS Lock

Figure 1. MutexBench

At the end of a 10 second measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. We also report a trivial measure long-term fairness, expressed as the number of iterations completed by the thread that made the least progress in the interval divided by the number of iterations completed by the thread that made the most progress. A value of 1 is ideally fair and values toward 0 less so.

Our implementation used the semaphore found in Listing-2 with B configured as 10.

We repeated each data point 11 times, with independent runs, and reported the median in Figure-1. Each individual graph represents the results for a specific lock algorithm, described below. Within each graph, the X-axis reflects the number of concurrently executing threads contending for the lock. The left Y-axis indicates aggregate throughput – the tally of all loops executed by all the threads in the measurement interval. The right Y-axis depicts fairness. Solid lines reflect throughput and dotted lines reflect fairness. Red lines are used for the baseline lock and blue for the baseline lock wrapped with the semaphore [3]. Magenta lines reflect a variant – labeled *Fractional* in the legend – where arriving threads run a Bernoulli trial, and with a probably of 99/100, skip the capacitor.

For clarity and to convey the maximum amount of information to allow a comparison of the algorithms, the throughput Y-axis and the X-axis are logarithmic.

We ran the benchmark under the following lock algorithms: **MCS** is classic MCS ; **Ticket** is a classic Ticket Lock ; **TTS** is a simple polite test-and-test-and-set lock with no backoff; **LIFO** is a simple lock that provides LIFO admission order; **Recipro** is the Reciprocating Lock algorithm, which can manifest long-term fairness of up to 2X.

Both `MCS` and `Ticket` are FIFO, and thus already provide ideal fairness. We include them in order to measure the overheads imposed by the capacitor. As noted, `LIFO` provides LIFO admission order and can be seen as maximally unfair. We included `LIFO` in order to show that the capacitor technique is effective at providing long-term fairness even in the face of an intentionally adversarial lock.

In general we find that the capacitor imposes a relatively modest throughput "tax" at low contention levels, while demonstrating high efficacy in terms of imposing long-term fairness.

the test-and-set lock is an interesting case as, beyond 9 threads, the capacitor improves both fairness and throughput.

5 Discussion

►Many platforms exhibit *topological favoritism* or, specifically, *NUMA favoritism*, where atomic operations that are either *near* (in the topology) the home node of the cache line or are near the cache that currently holds the line in modified state are more prone than *far* requesters to be successful and serviced in the near time. Perversely, test-and-set locks are *NUMA friendly* on such platforms in that they reduce *lock migration*[12, 14, 20, 21] by virtue of such architectural unfairness, offering improved performance compared to pure FIFO locks. Backoff mechanisms commonly used in test-and-set locks also exacerbate unfairness, as a waiting /thread that has backed off is less like to acquire the lock in unit time than a more recently arrived lock. Backoff also represents "dead time" and is not work-conserving, as it is possible, under varying load, that the last waiting thread is stalled in a long backoff period when it could otherwise acquire the lock and drive progress.

Wrapping such an unfair test-and-set lock with a capacitor acts to reduce unfairness arising from the factors mentioned above.

►A reasonable question to ask is why not just wrap the interior lock with a classic semaphore instead of the capacitor? Such a semaphore will indeed restrict concurrency on the interior lock, which may confer benefits. But it does not provide any form of bounded bypass guarantees. Lets say we configured the bypass limit B to be 10 and we have 20 threads circulating over the contended lock. One "unlucky" thread, T (or small set of threads) might be admitted by the semaphore, but then be continually bypassed by a stream later arriving threads, resulting in long-term unfairness and starvation. Using the capacitor, as the number of threads in a platoon decays down to just 1, our unlucky thread, assuming it was being interfered with and obstructed by other threads in that the platoon, will eventually become the sole remaining member of the platoon competing for the inner lock, and will then gain ownership and make progress [4].

► If we set the platoon size B to just one, then execution devolves to simple locking where all waiting is done via the semaphore. This degenerate configuration, while benign, is not particularly useful.

► As noted above, the semaphore activities in the capacitor impose a performance "tax" at low contention levels. The impact can be observed at low contention with very short critical sections and high arrival rates. In this operating regime the semaphore is usualy passive and threads do now tend to wait within the semaphore. That is, if there is no contention on the inner lock, then the capacitor provides no benefits. But the semaphore operations serve to increase the path length, adding latency. We now offer a number of techniques that can mitigate that "tax" or overhead :

❶ On arrival, in the `lock()` operation, threads run a simple randomized Bernoulli trial. The probability of success or failure, P, constitutes a tunable "knob". On success, we send control through the semaphore in the usual fashion for capacitors. On failure, however, we skip the semaphore and just divert control to directly acquire the inner lock. In that case, the subsequent corresponding `unlock()` also skips the capacitor *departure* phase. This approach reduces the "tax" overhead, and we note that we still impose long-term statistical admission fairness, although the short-term bypass

[3] CRVS is an acronym for Concurrency Restriction via Semaphore

[4] reflecting a *Leave no one behind* policy, in that all members of a platoon must depart before a new platoon is enabled via the semaphore.

bounds are no longer in place. The value of P reflects a trade-off between between the "tax" and the efficacy and time-frame of long-term fairness. We refer to this technique as *fractional bypass* and observe that it effective and acts to reduce long-term statistical admission fairness. Figure-1 shows the result of one form of fractional bypass. ❷ We can detect or otherwise intuit contention on the inner lock, and only activate the semaphore *arrival* and *departure* paths under contention, skipping the semaphore when there is no contention, and the capacitor construct does not serve to impose ordering or provide any value. That is, we avoid the semaphore when it is not needed and provides no benefit.

The relative tax (degradation compared to baseline) will be highest with very short critical sections and high arrival rates. Thus, to illustrate the operation of capacitor in the most challenging environment, we configure our benchmark in Figure-1 with a very short critical section and empty non-critical section.

► If scalability of the semaphore becomes a concern, we can easily apply the *multilane*[22] pattern to diffuse contention over a set of lanes, which each lane contains a semaphore. This preserves the bounded bypass property. We could also pick lanes at random, which abrogates the bounded bypass property, but still provides long-term statistical admission fairness.

►Threads blocked on the semaphore are engaged in long-term waiting while those waiting on the inner lock – having passed the semaphore – engage in short-term waiting. All threads within a platoon will be able to gain the inner lock before any threads waiting on the semaphore are able to acquire the inner lock. We also note that under contention, the number of threads in a platoon, currently executing the *arrive-depart* window, exhibits a sawtooth wave form. Initially B are allowed to pass the semaphore and compete for the inner lock. As those threads gain ownership and depart, the number of threads competing for the inner lock decays toward 0, at which point a new platoon is ultimately enabled.

►The "TWA" semaphore construct benefits from having wait-free constant-time arrival and release operations. Critically, it also provides FIFO FCFS admission order which is required to provide bounded bypass progress guarantees for the capacitor.

►The composite lock is subject to 2-stage waiting. Even if the semaphore has constant-time wait-free arrival and departure steps, and the inner lock similarly provides constant-time wait-free paths, we observe that the composite lock does *not* provide a wait-free arrival step because of the potential for 2-stage waiting, where a thread might need to wait twice in one acquisition episode, for the capacitor (the capacitor's semaphore), and then once again for the inner lock.

►Say we have T threads circulating, competing for a composite lock, and $T < B$. In this case threads can recirculate and re-arrive before the semaphore credits for a platoon episode are depleted and the capacitor closes. That is, in this circumstance, a thread may be able to enter and leave a given platoon "episode" multiple times. Some "fast" threads might bypass and overtake other threads in the same platoon. Such overtaking is bounded, however. Say $B = 8$ and we also have 2 active circulating thread that have passed the semaphore. One of the 2 threads, $T1$, is "fast" and the other, $T2$ is"slow" in that it passes the capacitor's `arrive` operator but then stalls before being able to acquire the inner lock. Our fast thread $T1$ can then *lap* or overtake slow $T2$ another 6 times before the capacitor becomes depleted and closes, and the fast thread $T1$ then needs to wait in the `arrive` step. Eventually the slow thread $T2$ will pass the inner lock, call *depart*, and the capacitor will re-open. In the very worst case and under sustained and equal offered load, a given thread can make at most $B - 1$ times more progress (admissions to the critical section) than another given thread. As such, our approach imposes a useful bound on long-term unfairness and also avoids short-term starvation.

While all cases of bypass are intra-platoon, when $T < B$, if a thread departs and then comes back through the still open semaphore, it can rejoin the same platoon and re-compete for the inner lock. In practice, when $T < B$, we observe that the semaphore tends to say passive and threads rarely block there.

When $T >= B$, and assuming sustained arrival rates and offered load, we find our approach provides *1-bounded-bypass*, as any "fast" thread will depart, re-arrive, and then block, joining the next platoon, and inhibiting bypass.

►We observe that *any* unfair lock can be wrapped with a capacitor. The composition technique is agnostic as to the properties or design of the inner lock. We note that if the inner lock happens to be NUMA-aware, then the wrapped lock will also be NUMA-aware.

►If the inner lock uses *global spinning*, where multiple waiting threads busy-wait on the same memory location, then lock inner lock will usually suffer from scalability collapse at high contention – with many waiting threads and a high arrival rate. Ticket locks and test-and-set are classic examples of locks that use global spinning. When wrapped with a capacitor, however, such locks may become scalable at high contention levels, as we have restricted the degree of global spinning. As such, wrapping test-and-set locks can provide both improved fairness *and* better throughput.

We observed that *fractional bypass* does *NOT* improve throughput for locks that use global spinning, as bypassing the semaphore allows large numbers of threads to concurrently engage in local spinning.

As expected, locks that avoid global spinning, such as MCS, do not gain any performance benefit from wrapping. Similarly, as MCS is already FIFO, so wrapping provides not fairness benefits.

►In the case of Reciprocating Locks, wrapping mitigates the long-term unfairness arising from the reciprocating admission schedule.

6 Conclusion

The Capacitor mechanism provides strong bypass and long-term fairness guarantees. It also acts to reduced performance variance.

References

[1] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Trans. Program. Lang. Syst.*, 1994. doi:10.1145/177492.177731.

[2] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *SIGPLAN Notices OOPSLA 1999*, 1999. doi:10.1145/320385.320402.

[3] Vitaly Aksenov, Dan Alistarh, and Petr Kuznetsov. Performance prediction for coarse-grained locking. *CoRR*, abs/1904.11323, 2019. URL: http://arxiv.org/abs/1904.11323, arXiv:1904.11323.

[4] Vitaly Aksenov, Daniil Bolotov, and Petr Kuznetsov. Peformance prediction for coarse-grained locking: MCS case. *CoRR*, abs/2110.05545, 2021. URL: https://arxiv.org/abs/2110.05545, arXiv:2110.05545.

[5] J.H. Anderson, Y.J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 2003. URL: https://doi.org/10.1007/s00446-003-0088-6.

[6] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1990. doi:10.1109/71.80120.

[7] Jonathan Corbet. MCS locks and qspinlocks. https://lwn.net/Articles/590243, March 11, 2014, 2014. Accessed: 2018-09-12.

[8] Jonathan Corbet. Mcs locks and qspinlocks, 2014. URL: https://lwn.net/Articles/590243/.

[9] Travis Craig. Building FIFO and priority-queueing spin locks from atomic swap, 1993.

[10] NIST Information Technology Laboratory National Vulnerability Database. Cve-2021-32036, 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-32036.

[11] Dave Dice. Malthusian locks. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017. URL: http://doi.acm.org/10.1145/3064176.3064203.

[12] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. *CoRR*, abs/1810.05600, 2018. URL: http://arxiv.org/abs/1810.05600.

[13] Dave Dice and Alex Kogan. Avoiding scalability collapse by restricting concurrency. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, Lecture Notes in Computer Science. Springer, 2019. doi:10.1007/978-3-030-29400-7_26.

[14] Dave Dice and Alex Kogan. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, 2019. doi:10.1145/3302424.3303984.

[15] Dave Dice and Alex Kogan. TWA - ticket locks augmented with a waiting array. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings.* Springer, 2019. doi:10.1007/978-3-030-29400-7_24.

[16] Dave Dice and Alex Kogan. Fissile locks. In *Networked Systems (NETYS 2020)*, 2021. URL: https://doi.org/10.1007/978-3-030-67087-0_13.

[17] Dave Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. *CoRR*, abs/2102.03863, 2021. URL: https://arxiv.org/abs/2102.03863.

[18] Dave Dice and Alex Kogan. Reciprocating locks, 2025. URL: https://arxiv.org/abs/2501.02380, arXiv:2501.02380.

[19] David Dice and Alex Kogan. Hemlock : Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2021. URL: https://doi.org/10.1145/3409964.3461805.

[20] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12. Association for Computing Machinery, 2012. doi:10.1145/2145816.2145848.

[21] David Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 2015. URL: http://doi.acm.org/10.1145/2686884, doi:10.1145/2686884.

[22] David Dice and Oleksandr Otenko. Brief announcement: multilane - a concurrent blocking multiset. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11. Association for Computing Machinery, 2011. doi:10.1145/1989493.1989545.

[23] Edsger W. Dijkstra. Over de sequentialiteit van procesbeschrijvingen. (EWD-35) E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription), undated, 1962 or 1963.

[24] Allen B. Downey. *The Little Book of Semaphores.* Green Tea Press, 2016. URL: https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf.

[25] Rotem Dvir and Gadi Taubenfeld. Mutual Exclusion Algorithms with Constant RMR Complexity and Wait-Free Exit Code. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8652, doi:10.4230/LIPIcs.OPODIS.2017.17.

[26] Stijn Eyerman and Lieven Eeckhout. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10. ACM, 2010. URL: http://doi.acm.org/10.1145/1815961.1816011, doi:10.1145/1815961.1816011.

[27] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–unlock: Is that all? a pragmatic analysis of locking in software systems. *ACM Trans. Comput. Syst.*, 2019. doi:10.1145/3301501.

[28] Wim H. Hesselink and Peter A. Buhr. MCSH, a Lock with the Standard Interface. *ACM Trans. Parallel Comput.*, 2023. URL: https://doi.org/10.1145/3584696.

[29] Wim H. Hesselink, Peter A. Buhr, and Colby A. Parsons. First-come-first-served as a separate principle. *ACM Trans. Parallel Comput.*, 2024. doi:10.1145/3669989.

[30] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Towards an ideal queue lock. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020. Association for Computing Machinery, 2020. URL: https://doi.org/10.1145/3369740.3369784.

[31] Siddhartha Visveswara Jayanti. Simple, fast, scalable, and reliable multiprocessor algorithms. Massachusetts Institute of Technology, 2023. URL: https://dspace.mit.edu/handle/1721.1/150219.

[32] Jouni Leppäjärvi. A pragmatic, historically oriented survey on the universality of synchronization primitives, 2008. URL: http://www.enseignement.polytechnique.fr/informatique/INF431/X09-2010-2011/AmphiTHC/SynchronizationPrimitives.pdf.

[33] Waiman Long. qspinlock: Introducing a 4-byte queue spinlock implementation. https://lwn.net/Articles/561775, July 31, 2013, 2013. Accessed: 2018-09-19.

[34] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, 1994. URL: https://doi.org/10.1007/11823285_84, doi:10.1109/IPPS.1994.288305.

[35] Evangelos P. Markatos and Thomas J. LeBlanc. Multiprocessor synchronization primitives with priorities. In *8th IEEE Workshop on Real-Time Operating Systems and Software*. IEEE, 1991. doi:10.1016/S1474-6670(17)51259-8.

[36] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 1991. URL: http://doi.acm.org/10.1145/103727.103729.

[37] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91. ACM, 1991. URL: http://doi.acm.org/10.1145/109625.109637.

[38] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020. doi:10.1145/3342195.3387521.

[39] Pedro Ramalhete. Ticket lock - array of waiting nodes (awn), 2015. URL: http://concurrencyfreaks.blogspot.com/2015/01/ticket-lock-array-of-waiting-nodes-awn.html.

[40] Pedro Ramalhete and Andreia Correia. Tidex: A mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016. URL: http://doi.acm.org/10.1145/2851141.2851171.

[41] Michael L. Scott and Trevor Brown. *Shared-Memory Synchronization, Second Edition*. Springer, 2024. doi:10.1007/978-3-031-38684-8.

[42] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. Be my guest: Mcs lock now welcomes guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2851141.2851160.

```
// Semaphores implemented with ticket locks or TWA
//
// Properties
// @ Wait-free constant-time Put/Take fast-path operations absent the need to park-unpark
// @ fast-path Benaphore-style operations
// @ FIFO-FCFS admission order
// @ Compact
// @ semi-local waiting via TWA-style waiting array
// @ Can use STP waiting policy
//    *  Direct futex waiting on the sequence counter in the waiting array
//    *  Implement the usual concurrent pop-stack chains via buckets in the waiting array
//       Push and Detach-and-wake-all primitives
//
// CONSIDER:
// @ Could also implement via "multilane" idiom
//    Use array of underlying simple inferior sub semaphores
//    Full construction can be scalable even if the underlying primitives
//    semaphores are not.
//    Requiries per-lock unshared array of sub semaphores

struct TWASemaphore {

  // Design considerations for the TWAHash() function -- ticket-aware hash function
  // The hash function maps Lock address and Ticket value pairs to indices -- ``buckets'' --
  // in the long-term waiting array.
  //
  // @ Desiderata for the hash function
  //   Efficient, low-latency and enables instruction-level parallelism ILP to
  //   the extent possible.
  //   We do NOT generally require a full-strength arbitrary hash function
  //   with the usual avalanche property.
  //   Instead, one specialized to the problem is better.
  // @ Index-aware in the sense that we understand that ticket values
  //   from a given lock or semaphore increase by 1, which we can use to our advantage.
  ///  Ideally, as the ticket advances, the index computed by
  //   TWAHash(L,ticket) will "walk" through the full gamut of possible bucket
  //   indices before repeating.
  // @ To reduce coherence traffic and false sharing, adjacent (or numerically proximal) ticket
  //   values should map to indices in the table that reside on different cache lines.
  //   A set of waiting threads will have adjacent ticket values but we want them
  //   well-dispersed over the buckets to avoid false sharing or even true collisions.
  // @ Hash collisions induce extra coherence traffic _and also
  //   induce futile wakeups in the buckets, as our wakeup operator
  //   activates _all the elements of a bucket and presumes that collisions are rare.
  //   We expect in general, however, that only one thread waits on a
  //   given bucket at a given time.
  // @ Cache-friendly and TLB-friendly to the bucket array
  // @ Standing in tension to the above, we are also concerned about inter-lock
  //   parallel ticket entrainment, where two different locks or semaphores might move in unison,
  //   inadvertently generate streams of ticket values that continue to collide in the array.
  //   Simple supplementary hashes can mitigate this issue.
  //   Multiplying the "L" lock address component by the "Ticket" component would
  //   be ideal but also tends thwart some of our other desiderata, enumerated above.
  //   Broadly, however, the other concerns dominate and have precedence over this
  //   particular design aspect.
  // @ Be aware of automatic stride-based hardware prefetch on some CPUs.
  // @ The multiplication constants in the TWAHash() function should be coprime
  //   with each other and the length of the array, and also sufficiently
  //   large that adjacent ticket values map to indices that map to
  //   different cache lines.
  //   This gives us the optimal "tickets marching through the array" behavior that we
  //   prefer and desire.
  //   The choice of constants is also convolved with the size of the bucket elements.
  // @ An interesting variation is to precondition the incoming ticket value : Ticket >>= 1.
  //   This groups threads with numerically adjacent ticket values into pairs.
  //   This increases futile wakeups but also yields a pipelined early wakeup effect
  //   that can put "near" futile successors a chance to wake early and warmup.
  // @ Another profitable variation divides the ticket value into distinct bit fields.
  //   The upper bits directly identify and select a logical "sub-page" in the waiting array.
  //   The lower bits undergo a hash operation.
  //   A given stream of sequential incrementing ticket values will "orbit" within a
  //   sub-page, accessing all elements, and then shift to the next sub-page.
  //   This approach minimizes page transitions for a given stream of ticket values.
  //   In turn, that access pattern can act to reduce TLB pressure.
  //   This is somewhat related to "Morton" or Z-order access patterns.
  // @ Gray codes also show promise

  [[gnu::pure]] uint32_t TWAHash (uint32_t Ticket) {
    return uint32_t(uintptr_t(this)) +(Ticket * 17) ;
  }

  // Changes to the UpdateSequence field indicate that long-term waiters
  // on that particular bucket must should re-check the Grant values to
```

```
// see if the waiting condition has been satisfied.

struct WaitBucket {
   std::atomic<uint64_t> UpdateSequence {0} ;
} ;

// TableSize is a tunable parameter ...
// A sensible policy is to set TableSize to be proportional to the number
// of logical CPUs in the system.
// We force alignment to reduce the number of TLBs underlying the table in
// order to reduce TLB misses.
// Ideally, we would place the table on large pages.
// We note that an extremely large table might sound appealing, as that
// design will reduce hash collisions and false wakeups, but a large
// table will also result in cache pollution as the ticket values
// march over the table.

[[gnu::pure]] static WaitBucket * IndexToBucket (uintptr_t A) {
   constexpr int TableSize = 2048 ;
   static_assert (TableSize > 0 && (TableSize & (TableSize-1)) == 0) ;
   static WaitBucket WaitSlots [TableSize] __attribute__ ((aligned(4096))) ;
   const int ix = A & (TableSize-1) ;
   return &WaitSlots [ix] ;
}

// LongTermThreshold tunable parameter ...
// Threshold for near-global vs far-local polite waiting
static constexpr int LongTermThreshold = 1 ;

TWASemaphore () {
   Grant.store (BBLimit) ;
}

TWASemaphore (int ITicket, int IGrant) {
   assert (IGrant >0 && BBLimit > 0) ;
   Grant.store (BBLimit) ;
}

public :
// Consider: alignas(128) here to reduce false sharing
std::atomic<uint64_t> Ticket {0} ;
std::atomic<uint64_t> Grant {0} ;

void SemaTake () {
   const auto tx = Ticket.fetch_add(1) ;
   int64_t dx = Grant.load() - tx ;

   // Fast-path uncontended return
   if (dx > 0) return ;

   // slow-path contention -- need to wait
   const auto s = IndexToBucket(TWAHash(tx)) ;
   auto mx = s->UpdateSequence.load() ;

   for (;;) {
      dx = Grant.load() - tx ;
      if (dx > 0) {
         return ;
      }

      // Short-term waiting -- global waiting on S->Grant
      // Our thread is near the head of the logical queue of waiting threads.
      // It is "proximal" or "near" to acquisition.
      // As such we switch from semi-local waiting in the WaitArray to
      // This strategy yields 2 modes or phases of waiting : near and far
      // TODO-FIXME: extract short-term waiting into a distinct dedicated modal loop.
      // Bail out of LT into ST when we become "near"
      // This gives the branch predictors better leverage
      if ((dx + LongTermThreshold) > 0) {
         Pause() ;
         continue ;
      }

      // Long-term distal waiting -- semi-local waiting via the WaitingArray.
      // We use indirect waiting on s->UpdateSequence "proxy" indicator
      // our thread is "far" from the head of the logical queue
      const auto vx = mx ;
      while ((mx = s->UpdateSequence.load()) == vx) {
         Pause() ;
      }
   }
}

// Optional optimizations ...
// Try to avoid unnecessary long-term notification operations in the common
```

```
    // uncontended case where there are no waiters.
    // Our approach automatically confers the same "fast path" benefits conferred by the
    // well-known "benaphore" optimization :
    // https://www.haiku-os.org/legacy-docs/benewsletter/Issue1-26.html#Engineering1-26

    void SemaPost () {
        auto g = Grant.fetch_add(1) ;
        g += LongTermThreshold ;

        // Optional optimization ...
        // In the case where there are no long-term waiting threads on this specific semaphore,
        // we would prefer to avoid otherwise unnecessary accesses to the long-term waiting array.
        // Absent such an optimization, back-to-back sequences of SemaPost() operations would tend to
        // "march" through the long-term waiting array, increasing cache pressure, with futile accesses.
        //
        // To this end, we fetch and examine the value of S->Ticket.
        // If the difference between that observed ticket value and the value we just wrote into
        // grant is less than the LongTermThreshold value then we can safely intuit that there
        // are no long-term waiters that need notification, and can skip the long-term notification step.
        // Note that this approach is conservative and racy.
        // If some other thread calls SemaPost() in the window between the fetch_add(), above, and
        // the fetch of S->Ticket, then we might still perform unnecessary notification, but
        // that is benign in terms of correctness.
        // That is, the fast-path return does not filter out all unnecessary long-term notifications.
        //
        // This optimization constitutes a "bet".
        // The downside is that we incur more coherence traffic as all SemaPost() operations access
        // both Ticket and Grant fields. Having said that, the cache line underlying Ticket and Grant is
        // most likely still in local "M"-state (MESI "modified") at the time of the load() and has
        // not yet been arbitrated away by concurrent accesses during the fetch_add() - load() window.
        // The economic "profitability" model for this optimization is thus platform-specific.
        // The access to Ticket is wasted and futile if we still ultimately need to poke
        // long-duration waiters spinning in the bucket.
        //
        // If we fail to take the fast-path return, then access to Ticket was futile
        // and we have incurred a penalty in terms of path length and coherence traffic.
        // So we have a tension and trade-off.
        // The fast-path attempt helps under light load but may incur a performance penalty
        // under contention.
        //
        // Using the fast-path shortcut return optimization means that the
        // SemaPost() operator must access both Ticket and Grant fields.
        // This precludes or obviates any coherence traffic benefit we might see from
        // sequestering Ticket and Grant fields on distinct cache lines.
        int64_t dx = g - Ticket.load() ;    // surplus - excess
        if (dx >= 0) {
            return ;      // common fast-path return : short-cut
        }

        // Poke successor-of-successor from long-term mode into short-term mode
        // The immediate successor has already been enabled to enter the critical section.
        // Our approach yields some useful pipeline parallelism as we now act
        // to shift the successor's successor from long-term waiting to short-term waiting mode.
        const auto s = IndexToBucket (TWAHash(g)) ;
        s->UpdateSequence.fetch_add(1) ;
    }

    void SemaPostConservative () {
        auto g = Grant.fetch_add(1) ;
        g += LongTermThreshold ;
        const auto s = IndexToBucket (TWAHash(g)) ;
        s->UpdateSequence.fetch_add(1) ;
    }

    // Simplistic non-optimized form -- incremental and naive
    void SemaPostMulti (int N) {
        while (--N >= 0) {
            SemaPost () ;
        }
    }
} ;
```

```
struct CapacitorTWA {

    // tunable "knob" ...
    static constexpr uint64_t BypassLimit = 10 ;

    // test-and-test-and-set interior lock
    std::atomic<int> tts alignas(128) {0} ;

    // Capacitor ...
    std::atomic<int> Depart alignas(128) {0} ;
    TWASemaphore CRSema {0, BypassLimit} ;
```

```
public :

  inline auto operator+(std::invocable auto && csfn) → void {
     // Arrive :
     CRSema.SemaTake() ;

     // Acquire interior TTS lock ...
     for (;;) {
        if (tts.exchange(1) == 0) break ;
        while (tts.load (std::memory_order_acquire) ≠ 0) {
           Pause() ;
      }
     }

     EnterCS :
     assert (tts.load() ≠ 0) ;
     csfn() ;
     assert (tts.load() ≠ 0) ;

     // Release interior inner lock ...
     tts.store (0, std::memory_order_release) ;

     // Depart :
     auto v = Depart.fetch_add(1) + 1 ;
     assert (v >= 0 && v <= BypassLimit) ;
     // Invariant : Semaphore protects against v > BypassLimit
     // v == BypassLimit implies We are terminal thread in cohort
     // singleton -- no concurrency possible
     // Thus no need for atomic to update Depart
     // Can use simple store-release

     if (v == BypassLimit) {
        auto k = Depart.exchange (0) ;
        assert (k == BypassLimit) ;
        CRSema.SemaPostMulti(BypassLimit) ;
     }
  }
} ;
```

Listing 2. Capacitor - TWA Semaphore