Alohomora: Unlocking the Secret of Locks!

Jatin Khare[1], Shreyas Ravishankar[1], and Shruti Dutta[1]

[1]Department of Electrical and Computer Engineering, University of Texas at Austin, Texas, USA.

***Abstract*— Extracting performance from present day's multi-core systems is a very challenging task. One of the major difficulties that arise are due to synchronization. Synchronization is used to guarantee correctness, but it also forces parts of the code to be serialized. In this report, we present a comprehensive study of various mutual exclusion techniques. Finally based on our observations, we also propose a new lock algorithm which incorporates best of all the discussed aspects.**

***Keywords*— Multi-threading, Consistency, Mutual Exclusion, Synchronization, Parallelism.**

I. Introduction

It is quite difficult to think how synchronization primitives scale, and this makes the inherently challenging task of extracting the maximum parallelism out of multi-core systems more difficult [1]–[3]. One has to make special efforts to guarantee that synchronization does not become the bottleneck and end up degrading system performance. Essentially, in a multiprocessor system we have to make sure that all the threads/processors work in a coordinated manner thus maintaining the accuracy. As more software developers need to be aware of the hardware intricacies to write better code, it is more important now than ever to understand the consequences of hardware implementations on the performance of synchronization primitives and mutual exclusion algorithms. It is important to understand the bottlenecks and what fails while scaling to a multiprocessor system. Factors such as hardware micro-architecture, cache-coherence protocol, and consistency model might affect the performance. Further, while scaling across multiple threads, size of critical sections and thread placements might affect the choice of spin-lock to be used.

II. Background and Recent Work

People have spent years analyzing the lock performances for given setups and many have proposed novel improvement ideas over the course of time. David *et al.* [4] conducted a detailed performance study on locks and concluded that scalability of primitives is mainly a property of the hardware. [5] presents a set of new benchmarks while evaluating them over various state-of-the art x86 architectures. Sanidhya *et al.* in [6] proposes a new family of locks that improves upon the existing locking mechanism implemented by Linux Kernel. Here we aim to not only perform our own experiments for scalability and performance of locks but also try to propose a new locking technique that consolidates various aspects.
In this report, we study different mutual exclusion constructs such as Compare-and-Swap (CAS), Test-and-Set (TAS), Test-Test-and-Set (TTAS), Fetch-and-Increment (FAI), Linked-Load/Store-Conditional (LL/SC) [3], Ticket Lock [7], [8], Anderson Lock [9], Mellor-Crummey and Scott (MCS) Lock [10], and Non-Uniform Memory Access (NUMA) aware lock algorithms [11]. These locks are combined with different waiting strategies such as busy-wait, scheduled yield, blocking wait, and pause [12]. Busy-wait is further optimized with active, exponential, and random back-off. Please refer to the Appendix I and II for the pseudo-codes and brief description about these locks and methods.

The report is structured as follows: In Section III we discuss the design of our benchmarks, followed by Methodology and Experiments in Section IV. Observations are presented and discussed in Section V. Section VI touches upon a few challenges we faced, and Section VII presents our new locking technique. Section VIII is the Conclusion.

III. Design of Benchmarks

We use Google Benchmarks to evaluate the latency[1] of various locks. We design our test-beds to analyze different aspects of synchronization primitives such as varying critical section size, different workloads (counter and stack), thread contention, number of lock acquisitions per thread, thread scheduling, and thread pinning strategies. We ran around 600 permutations of these parameters across various locks. For next few pages, we will be using the terms `LOOP_COUNT` and `CRITICAL_SECTION_SIZE` as per the implementation shown in Algorithm 1. The 'Case#s' in Table 1 will be used to refer to different cases throughout this paper.

Algorithm 1 Common Resource Description[2]

```
1: for $i \leftarrow 0$ : LOOP_COUNT do
2:     CRITICAL_SECTION
3: end for

4: function CRITICAL_SECTION
5:     for $i \leftarrow 0$ : CRITICAL_SECTION_SIZE do
6:         $counter \leftarrow counter + 1$
7:     end for
8: end function
```

IV. Methodology/Experiments Used

A. Comparing x86 vs ARM

Modern ISAs have several implementations for atomic instructions. The semantics for each of these atomic operations

[1]wall-clock time (real-time) and the time for which the threads make forward progress (CPU time)

[2]the total work done in all Cases is the same

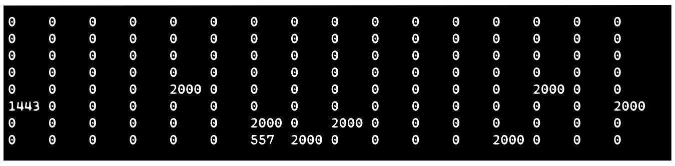
Case Number	`LOOP_COUNT`	`CRITICAL_SECTION_SIZE`	Brief Explanation
Case I	100000	1	Multiple lock acquires and releases by each thread.
Case II	1000	100	Trade-off between extremes
Case III	1	100000	Longer critical section, and single acquire and release by each thread.

Table 1. Description of Various Cases.

is slightly different from one another and some perform better than others under different circumstances. We analyze the latency numbers for Case I, Case II, and Case III on Apple M2 chip [ARM64], and AMD Ryzen7 [x86-64] (each having 8 cores) to understand the behavior of different locks and how their ISA and micro-architecture affect the performance of these locks. Further, using total latency and lock-acquisition latency, we also try to understand the effects of different cache coherency models and consistency patterns on synchronization primitives.

B. *Impact of OS Scheduler and Thread Affinity*

We first try to understand the nature of OS scheduling on Texas Advanced Computing Center (TACC). Interestingly, N threads are not always bound to N fixed cores, and the OS will not always schedule N threads on N cores. Some cores might get reused (artificial over-subscription) or new cores might be used throughout the lifetime of the program. Fig. 1 shows the core usage (using rdtsc instruction) for CAS Case II implementation where 8 threads are run on 128 cores. Each value shows the number of times a thread used a particular core. Here 9 cores are involved for 8 threads, and the workload is not distributed equally.
We observe that one core is used 1443 times, and there is another core that is used 557 times. This is a clear example of thread migration. This randomization leads to stale cache data in the original core's cache, and unnecessary cache transfers across the cores involved. Hence we also examine the effect of pinning the threads on latency.



```
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0    0    2000 0    0    0    0    0    0    0    0    2000 0    0
1443 0    0    0    0    0    0    0    0    0    0    0    0    0    0    2000
0    0    0    0    0    0    2000 0    2000 0    0    0    0    0    0    0
0    0    0    0    0    0    557  2000 0    0    0    0    2000 0    0    0
```

Fig. 1. A 8 × 16 matrix representing the 128 cores in one node on TACC with each number showing the number of times a thread was run on a particular core (here 8 threads are running in parallel).

C. *Scaling*

We perform scaling experiments with two types of critical sections: concurrent counter and concurrent stack. Besides, to understand the effects of synchronization on a more generic workload, we study the performance of these locks on LevelDB [13](a fast key-value storage library by Google). The following aspects of locks are studied:

(i) Rate of Lock Acquisition: The `LOOP_COUNT` is varied from 1 (Case III) and 100000 (Case I).

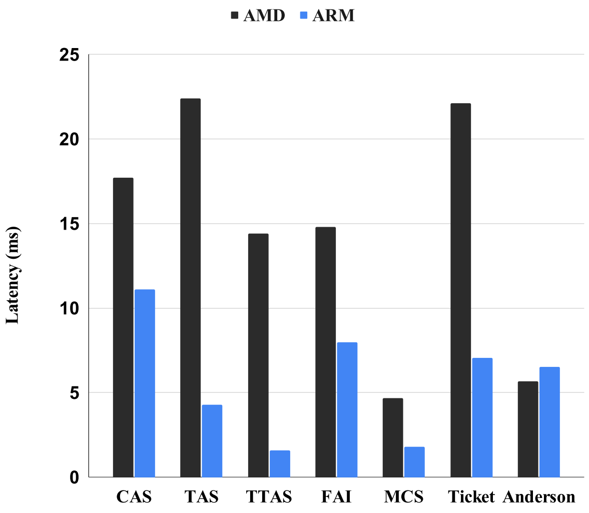


Fig. 2. Latency Plot for Case II for various locks for ARM Vs x86.

(ii) Size of the Critical Section: The size of the critical section (`CRITICAL_SECTION_SIZE`) is varied from 1 (Case I) and 100000 (Case III). Higher critical section sizes directly imply more work per lock acquisition.

(iii) Thread Contention: The number of threads are varied from 1 to 256 (with a factor of 2). Increasing the number of threads is at the heart of measuring lock scalability.
#threads [0, 63] $\rightarrow$ intra-socket
#threads = [64, 127] $\rightarrow$ inter-socket (movement of cache lines has a greater cost when moving inter-socket).

(iv) Waiting for the lock to get freed: Busy-wait not only wastes CPU cycles but trying to acquire the lock instantly after the first failure is not the most optimized method in terms of network traffic. Hence, we also implement different waiting methods [12] and back-off strategies that are described in the Appendix.

V. Observations and Results

A. *Comparing ARM vs x86*

Fig. 2 shows the latency plots for the tests that use concurrent counter as the critical section run for Case II. As evident from the figure, ARM cores perform better than x86 cores in most cases.

- In the case of CAS, TAS, and FAI, even though x86 has dedicated hardware instructions to implement them, we observe that LL/SC (the hardware primitive used in ARM cores) performs better. Primarily, LL/SC uses load-store architecture where reads (LL) and write (SC) can be implemented as separate instructions thus saving us the cost of atomic operations.

- The consistency model of ARM ISAs is quite weak as compared to the TSO of x86. The weak consistency model allows an LL/SC to reorder loads and even stores to other cache lines. ARM provides an architectural guarantee of eventual progress for LL/SC sequences of limited length.
- The ARM cores in M2 have huge private L_1 and L_2 caches. Each core has a 128 KB L_1 data cache and a 16 MB shared L_2 cache as compared to AMD having a 32 KB L_1 cache and a 4MB L_2 cache. The huge cache size on the M2 gives the processor some performance boost, even though caches help in improving the performance of a single thread (observed with concurrent stack workload as our memory footprint is bigger than the size of caches).
- It is important to note that MESIF does not allow sharing of the dirty cache line whereas MOESI does. Hence in cases of atomic RMW implementation, MESIF would perform worse than MOESI as the dirty data first needs to be written back and then the next core can start using the cache line from the main memory. E.g. in the case of TTAS, the O state is further helpful in sharing the dirty cache line (lock variable) with the cores performing the 'Test' part of the Test-TAS lock.

B. *Scaling Experiments*

i) Size of the critical section and rate of lock acquisitions:

- **Total lock performance**: The locks perform better with larger critical sections and smaller `LOOP_COUNT`. This contradicts the general notion that smaller critical sections lead to better performance. For example, in order to increment variable (`LOOP_COUNT`×`CRITICAL_SECTION_SIZE`) times, there can be multiple approaches where each thread acquires the lock `LOOP_COUNT` times just to increment the variable `CRITICAL_SECTION_SIZE` times after the lock acquisition. Our analysis shows that the performance is better when `CRITICAL_SECTION_SIZE` ≫ `LOOP_COUNT` across all locks. Fig. 3 shows that performance is bottle-necked by the rate of lock acquisition rather than the size of the critical section.
- **Worst-case wait time per acquisition**: Increasing size of the critical section does improve the total latency, but the worst-case lock acquisition time increases with the increase in critical section size. This can be seen in Fig. 4. Hence we can conclude that the number of times a lock has been acquired plays a more important role in determining the overall performance of the system.

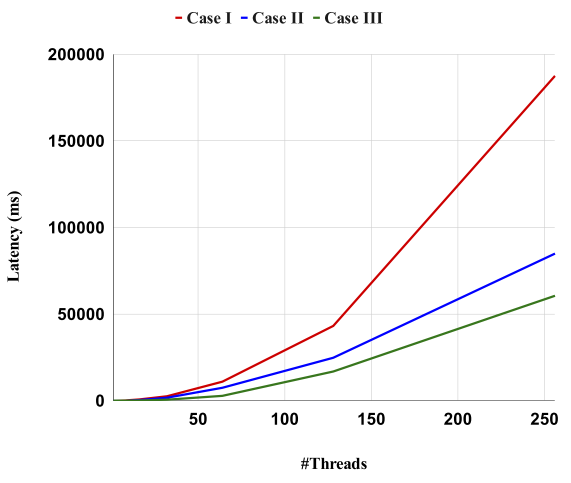


Fig. 3. CAS Latency for different Cases.

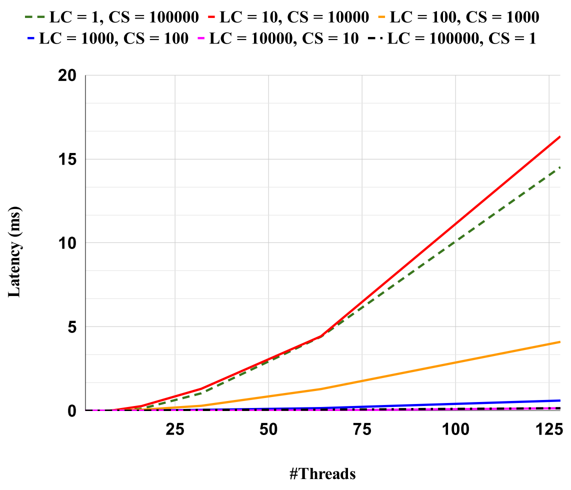
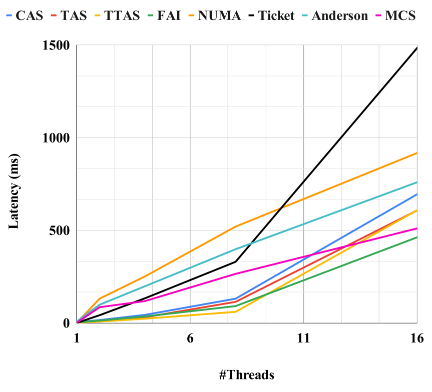


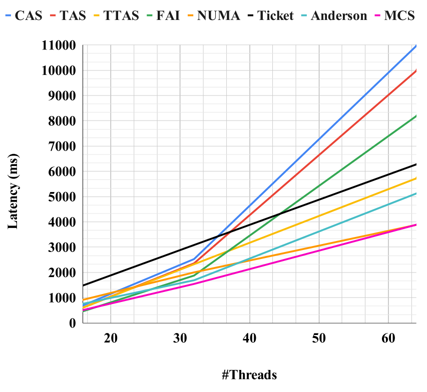
Fig. 4. Worst-case wait time analysis for different Cases of same workload.

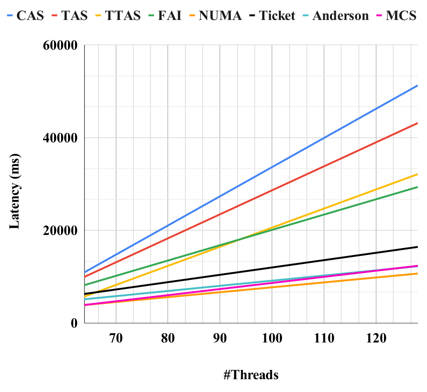
ii) Thread contention (comparison across different locks): Thread contention plays a critical role in deciding the performance of a multi-core system. In this section, we study the latency of different locks under high and low contention scenarios.

(a) Low thread contention (0-16 threads)

- When the thread count is small, i.e. contention is low, the instruction overhead of the complicated locks tends to dominate the overall performance of the system. From Fig. 5 (a), we one see that the best performing locks in a high contented scenario perform the worst in low contention scenario due to this extra overhead of storing values in array/queue and other complicated operations.
- Ticket lock under low thread contention (8-16 threads) performs the worst. As ticket lock favors fairness and there's a price for the same, which is more observable in the low thread count.







(a) (b) (c)

Fig. 5. (a) Latency plot across locks [0-16] threads, (b) Latency plot across locks [16-64] threads, (c) Latency plot across locks [64-128] threads.

Table 2. TAS Latency (in sec) results when 128 cores are oversubscribed

Threads	Busy-wait	Scheduled Yield	Active	Exponential	Random
256	225.6	60.612	82.403	56.720	55.472

(b) High thread contention

Intra-socket (16-64 threads, 64 cores)

- Locks such as CAS, TAS, and FAI perform worse as the thread contention increases. Complicated threads such as MCS, Anderson, and NUMA start performing better as shown in Fig. 5 (b) (they were designed to provide better results in a high contented scenario)
- CAS performs worse than TAS which is not intuitive given the invalidations generated in the 'always write' TAS case. We expect that maybe CAS also fetches the lock variable with write permission which can lead to unnecessary invalidations like TAS even when lock is not taken taken [5]. Moreover, CAS also has an extra overhead due to the compare operation.
- As expected, TTAS performs better than TAS and CAS due to the extra Test operation which avoids unnecessary test-and-sets.
- MCS performs best until 64 threads w.r.t NUMA as NUMA's real advantage is seen when we go inter-socket.

Inter-socket configuration (64-128 cores)

- NUMA seems to be performing the best at an inter-socket level as it optimizes inter-socket performance by prioritizing the intra-sockets threads first before going to the next socket as shown in Fig. 5 (c). This reduces the number of inter-socket NUMA coherence transactions.

iii) Back-off Strategies:

As per the Appendix II, when the cores are oversubscribed, busy wait performs worse as compared to giving up the core (using a yield operation). One can also see from Table 2 that exponential and random back-off perform better than active back-off, as in active back-off the CPU core is actively busy waiting and is not free to schedule the remaining threads. In passive back-off schemes, the CPU is available to the other threads if one of the threads is waiting on the resource to get free.

In general as seen from the Table 3, any wait-for-lock technique would perform better than busy wait (spin-lock) as it would avoid creating unnecessary coherence traffic and network contention, thus decreasing the overall time as shown in the table below. Out of all the wait-for-lock techniques, the 2 best strategies that work across a lot of locks and contention scenarios are **scheduled yield** and **exponential backoff**.

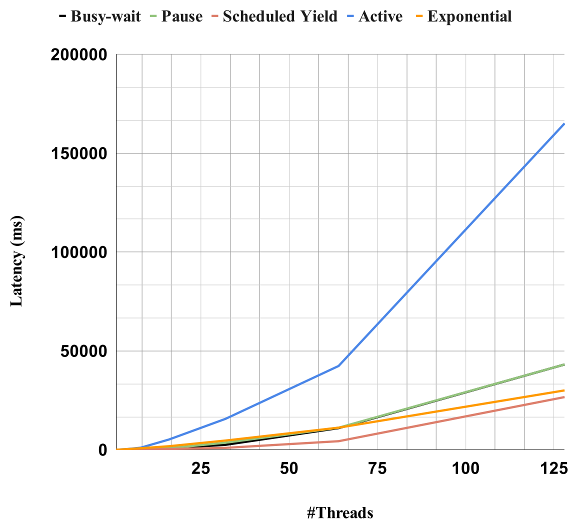


Fig. 6. CAS Back-off strategies analysis for Case I.

Table 3. TAS Latency (in sec) results for 32 threads

Threads	Busy-wait	Scheduled Yield	Active	Exponential	Random
32	2.360	2.069	1.475	1.348	0.939

For Case I: In this scenario shown in Fig. 6, each thread tries to acquire the lock higher number of times. Here scheduled yield works better than other back-off algorithms. As said earlier, the OS does not pin the threads to a core, and hence giving up the CPU allows other threads to continue their progress before they too start waiting.
One might wonder if the lock is owned by a single thread, then what is the use of yielding when other threads also have to wait for the lock? However, as we point out in the beginning, the OS often creates artificial over-subscription scenarios, where yielding the CPU would be a more optimal choice.

For Case III: Exponential back-off wins in Case III (Fig. 7), where each critical section size is more and each thread enters the critical section only once. This is because each thread does only a fixed amount of work, and does a lot of work per lock acquisition. So it makes sense for the other threads to back-off for longer periods of time (in power of 2), which is indeed possible using an exponential back-off-like strategy.

C. *Varying the Workload*

Across all the experiments, we see similar trends for the stack workload and hence has been omitted from here. We performed the experiments on Google's LevelDB across different locks to see if the performance changes on a generic workload. As shown in Fig. 8 and Fig. 9 below, we see **similar trends** in LevelDB performance as the results discussed above. As the number of threads are less, lock implementations with smaller overhead perform better. As contention increases, the more sophisticated locks start performing better. Though we ran the performance numbers on all the LevelDB benchmarks. Due to similar trends, we here present the results for the benchmark *readrandom*, where N concurrent reads happen randomly across the locations to have a fair evaluation.

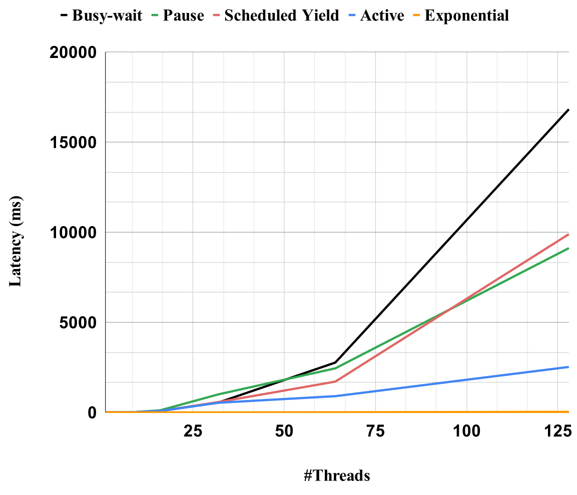


Fig. 7. CAS Back-off strategies analysis for Case III.

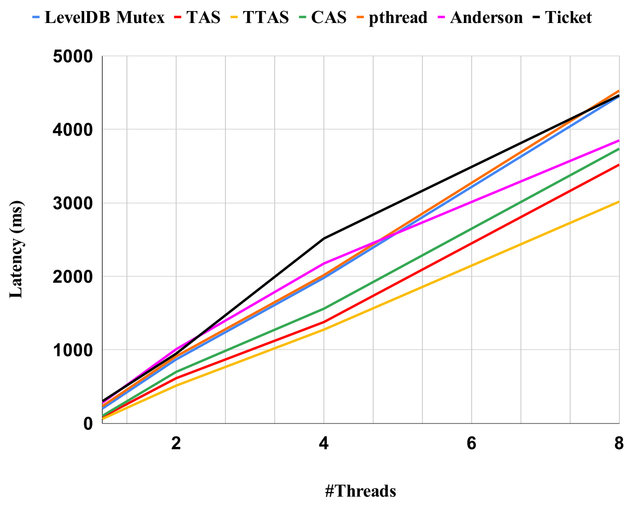


Fig. 8. LevelDB latency analysis across various locks for low thread contention.

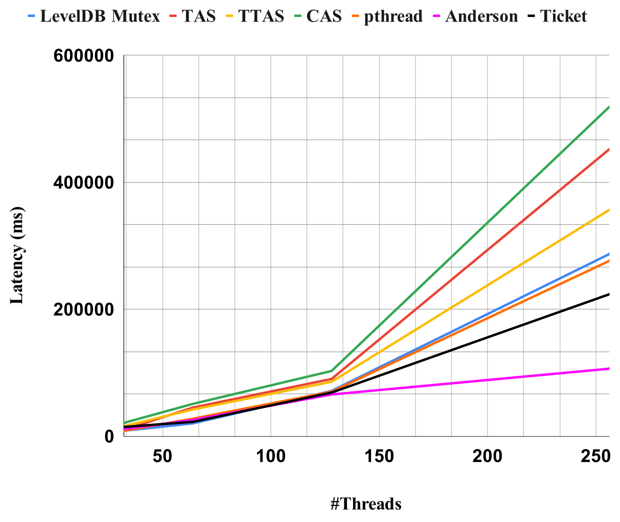


Fig. 9. LevelDB latency analysis across various locks for high thread contention.

D. *Thread Placement*

An experiment to analyze the latency variations for different thread placements was performed on TACC (Lonestar6). Though we tested these results across all locks and numbers of threads ranging from 2, 4, 8, 16, 32, 64, and 128, we present the results of the following configurations due to similar trends. (Fig. 10 shows the results of 3 runs for each thread pinning configuration)

Type of lock $\rightarrow$ CAS
Number of threads$\rightarrow$ 8
Number of counter variables $\rightarrow$ 2 (let's call them A and B)
Threads Classification: Half threads increment the counter variable A and the other half B.

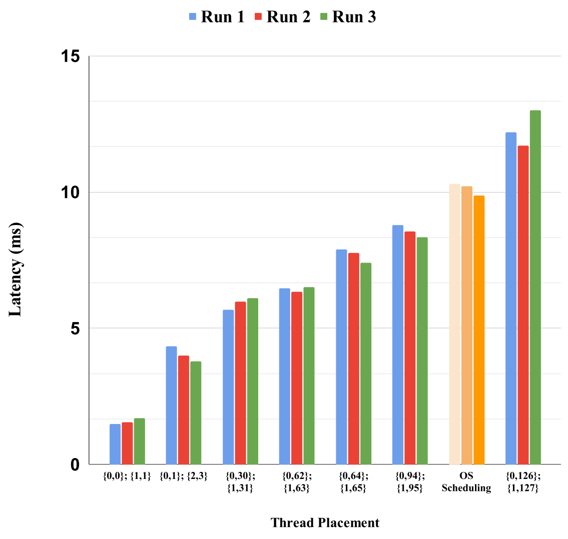


Fig. 10. Performance analysis of Thread placement.

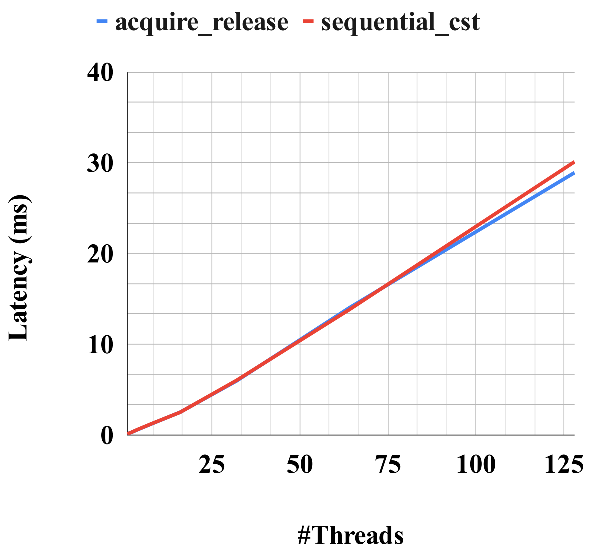


Fig. 11. Latency plot for memory ordering case II.

Thread Placement: $\{n_1, n_2\}$; $\{m_1, m_2\}$ where for $2N$ total threads working on A, N threads are placed in core n_1 and rest of the N threads are placed in core n_2. Similarly for the ones working on B are placed on m_1 and m_2.

- We find that for hyper-threading (HT) scenario where all the similar threads (working on A or B) are placed on the same core, the cache values (lock variable and the shared counters) are shared and hence incur the minimum latency. As we move these threads farther from one another, the latency keeps on increasing.
- Distance within socket also matters. Intra-socket latency is seen to increase when the threads are placed far from each other. Moreover, the OS scheduling latency remains at the higher end as the threads can be placed randomly.
- The OS scheduling is random in nature, and hence performs badly with respect to the cases when the threads are fixed to specific cores except when the threads are placed the farthest.
- HT in $\{0, 0\}$; $\{1, 1\}$ performs the best as there are no cache-to-cache transfers.

`perf c2c`: We use `perf c2c` for the thread placement case $\{0, 1\}$;$\{2, 3\}$ on Intel i5 with 4 physical cores. The stats show us that the cores 0 and 1 fight for exclusive access to the cache line for A and similarly 2 and 3 fight for B. The Hit_m[3] values for these cases wander around 30-45%. Upon running the test for the $\{0, 0\}$;$\{1, 1\}$ case, none of the cache lines are shared between cores, explaining the lower latency.

E. *Memory Consistency*

We ran our tests with multiple different implementations that utilize different memory ordering knobs provided by the C++ atomic library. For reference, we are demonstrating the ticket lock results which were experimented with relaxed and acquire-release, and sequential consistency models (Refer Fig. 11). These were our observations-

- For all the Cases (I, II, III): The change in wall clock time with and without these relaxed models was not significant. This might be because even though barriers incur 100s of clock cycles in case of sequential consistency, our programs were of millisecond order of execution. So a difference of a few 100s nanoseconds does not contribute to a huge difference in the run-time of the programs.
- Also we must understand that these knobs only affect memory orders that are possible within the lock and unlock a portion of the code. The rest of the program is reordered using the given processor's consistency model (AMD x86 uses TSO). This further strengthens our reasoning for the negligible changes observed.

F. *Understanding the Effects of Coherency on Locks*

Working with synchronization on muti-core systems involves a deep understanding of the effect of coherency on locks [5], [14]. Based on our tests and a comprehensive literature review, we hereby provide our qualitative description of the same.

Most atomic instructions work in a read-modify-write (RMW) manner. We assume that an atomic first performs a load that is transformed into a read with the intent to modify (RWITM) hence sending invalidations to other copies. The store operation is performed after that. The interesting thing to note here is that in often, the invalidations are sent as part of the atomic read-modify-write only for the lock acquisition to fail, which is unnecessary.

[3]Misses for values that are modified in others' cache.

In simple terms, the latency of an atomic operation can be written as:

$$L = T(A) + T(CC) \tag{1}$$

where A is the atomic operation, and CC is the coherence state of the cache line. $T(CC)$ can be defined as the latency of taking the read ownership of the given cache line and invalidating other cache copies. $T(A)$ is the latency for locking the cache line, executing A, and writing the value [5].

A general notion is that atomic RMW instructions (CAS/TAS/FAI) are more expensive than normal loads.
But this might not always be the case. In general, remote accesses consume a lot more time, and thus the cost to perform a normal remote operation falls close to a remote atomic [14]. Here is the reason why:

- **Local access (L_1 hit):** Consider the case where the lock variable is present in M/E state in the private cache of the core that is trying to perform an atomic instruction. This is the case where the latency of the read and atomic RMW will be comparable.
  On the other hand, if the lock variable is in the shared state, the atomic RMW operation will cause invalidates, this increases the gap between the read and the atomic instruction latency.
- **Remote access:** If the cache line is in a remote core in M/E state, we expect the cache-to-cache transfer latency to dominate the entire latency, hence the difference between read and atomic RMW will not be significant.
  Whereas if the cache line is in the S state but on a remote core, the invalidations and cache-to-cache transfers come into picture forming a significant portion of the total latency of atomic RMW instruction.

VI. Challenges in Testing and. Future Scope

The time measurement at the instruction level can be accurately measured by using **rdstc** instruction [15]. But using rdtsc instruction was challenging because:
1) We need to ensure serialization of rdtsc instruction and make sure no memory reordering takes place.
2) ARM core does not have this (or similar) instruction.
Hence, we decided to use the `std::chrono` and Google benchmark libraries in C++ to measure time which gave us a good estimate of lock acquisition latency and lock performance.

Future Scope:

- Given the time constraint, even though we have analysed several lock constructs, many more exist. In future, this study can be expanded (semaphores, condition variables, barriers, etc.) to make the understanding of mutual exclusion stronger.
- Expanding the LevelDB experiments further to other popular and real-world databases such as MangoDB would provide an insight to different ways every database tackles the problems of shared memory.
- Due to difficulties in getting access to and using an ARM based computer-cluster, the ARM vs x86 experiments were ultimately performed on our PCs. But it would be interesting to perform these experiments on more cores and sockets to compare between the two architectures.

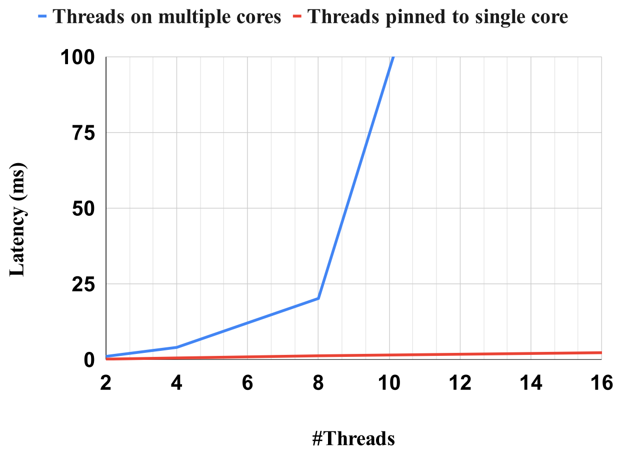


Fig. 12. Latency plot for Single core Vs Multi core thread placement.

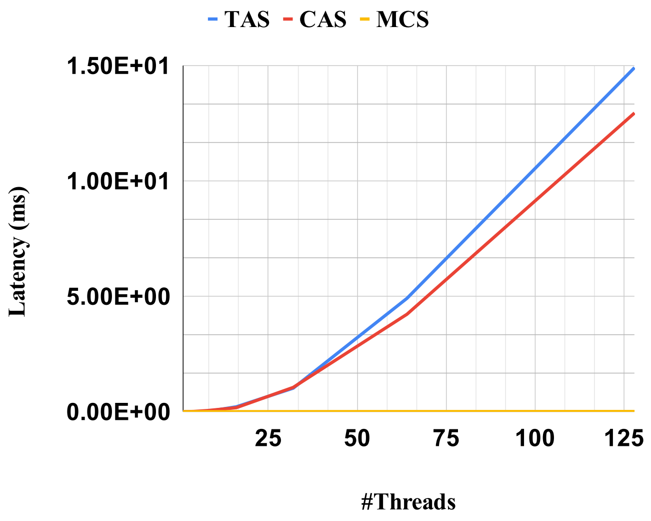


Fig. 13. Lock Acquisition Latency across CAS, TAS, MCS.

VII. New lock proposal

If the workload has a critical section is the major chunk of the code, then do propose to do following:

- Since we know that the critical sections are serialized anyway, we might as well reduce the coherence traffic involved in obtaining locks and updating the variables. This is apparent by the Fig. 12 which shows pinning N threads on one single core gives us much better performance as it avoids the cache coherence overhead.

- We could create a new **barrier*** before entering the lock function, so that all threads perform previous tasks (and wait for other threads to reach the barrier*). Note that the barrier* would be a special kind of instruction performing 2 functionalities:
  i) Pick a core/small subset of cores $\{core_S\}$ and pre-fetch the lock variable and shared variables of their critical section. Make sure these pre-fetched lines do not get evicted).
  iii) Then dynamically change thread affinity so that all threads map to $\{core_S\}$ set of cores. Once the critical section of the code is performed by the threads, the threads can be allowed to move on to the rest of the non-critical code. This is done by un-setting the thread affinity `sched_setaffinity()` and then simply relying on the OS scheduler to now map the threads back to all the core.
  The reason why we insist on a high critical section size is to amortize the latency of the barrier. So this solution might not be the best for smaller critical sections.

We haven't formally implemented these ideas due to a shortage of time, although it will be interesting to see how feasible to implement and effective this idea is.

VIII. Conclusion

Following are some takeaways from this project:

- The rate at which the lock is acquired plays a more important role than the size of critical section in determining the overall performance of the lock.
- The way in which the threads are placed on the available cores plays a primordial role in multi-threaded system performance.
- If there are multiple lock variables, we can distribute them over various cache lines to avoid coherency traffic.
- Putting the system to sleep while waiting for the lock might be better for the code despite the additional system call.
- Use a simpler lock (CAS/TAS/FAI) when the contention is less, and we move to MCS lock if the contention is more. Further, if we detect data crossing the node, we should have a NUMA-aware lock. A similar implementation is done in the Linux kernel with the concept of fast and slow paths based on contention. The intention for this is to reduce the lock acquisition overhead when contention is less. This is also apparent from our results as seen in Fig.13.
- The latency of CAS and TAS are similar when we consider #threads < 16, so we can use a simpler CAS when #threads < 16 and use MCS when #threads > 16.
- We hope this study would help to understand the various dimensions of lock performance related to hardware and software aspects of the synchronization primitives.

Acknowledgment

We would like to for providing us the environment thank Derek, Alex, Rui Ma, and all our classmates thinking and pushed us to our extremes. who helped us develop our critical

{maybe multi-threaded printing of this poorly protected statement was a wrong idea :P}

References

[1] M. L. Scott, "Shared-memory synchronization." [Online]. Available: https://www.amazon.com

[2] T. Downs, "A concurrency cost hierarchy," Jul 2020. [Online]. Available: https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html

[3] M. L. Scott, *Synchronization*. Boston, MA: Springer US, 2011, pp. 1989–1996. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4-252

[4] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 33–48. [Online]. Available: https://doi.org/10.1145/2517349.2522714

[5] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," Oct 2020. [Online]. Available: https://arxiv.org/abs/2010.09852v1

[6] S. Kashyap, I. Calciu, X. Cheng, C. Min, and T. Kim, "Scalable and practical locking with shuffling," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 586–599. [Online]. Available: https://doi.org/10.1145/3341301.3359629

[7] D. P. Reed and R. K. Kanodia, "Synchronization with eventcounts and sequencers (extended abstract)," in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, ser. SOSP '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 91. [Online]. Available: https://doi.org/10.1145/800214.806550

[8] L. Lamport, "A new solution of dijkstra's concurrent programming problem," in *CACM*, 1974.

[9] T. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[10] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, p. 21–65, feb 1991. [Online]. Available: https://doi.org/10.1145/103727.103729

[11] D. Dice and A. Kogan, "Compact numa-aware locks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[12] Nick, "Coffeebeforearch - the github page," Oct 2020. [Online]. Available: https://coffeebeforearch.github.io/

[13] Google, "Google/leveldb: Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values." [Online]. Available: https://github.com/google/leveldb

[14] P. Tsigas and Y. Zhang, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," in *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 320–321. [Online]. Available: https://doi.org/10.1145/378420.378810

[15] S. Pai, Y. Patel, A. Raina, M. Verma, and V. Goel, "Everything you still don't know about synchronization and how it scales," Dec 2016.

Appendix I

A. Basics of Synchronization Primitives

Compare and Swap (CAS): The memory location is atomically swapped with a new value only when its old value

Algorithm 2 Compare-and-Swap

```
1: function CAS($*location, compare\_value, swap\_value$)
2:     if $*location == compare\_value$ then
3:         $old\_value \leftarrow *location$
4:         $*location \leftarrow swap\_value$
5:     end if
6:     return $old\_value$
7: end function
```

matches with the provided expected value. It has a small lock acquisition latency at low contention, and lower memory footprint. But CAS lacks fairness and incurs a lot of cache coherence traffic.

Test and Set (TAS) and Test-Test and Set (TTAS): In TAS, the memory location is set to 1 and the old value is returned back. This return value tells us whether the value was finally set by our TAS operation or it was already set. TAS also has small lock acquisition latency at low contention and lacks fairness.
The TTAS only sets the value to one when the location has a 0 stored in it. Else it returns 0 stating the TAS test failed. This helps us address the cache coherence traffic issue in TAS.

Algorithm 3 Test-and-Set

```
1: function TAS($*location$)
2:     $old\_value \leftarrow *location$
3:     $*location \leftarrow 1$
4:     return $old\_value$
5: end function
```

Fetch and Increment (FAI): Here 1 is added to the memory location and the old value is returned back.

Algorithm 4 Fetch-and-Increment

```
1: function FAI($*location$)
2:     $old\_value \leftarrow *location$
3:     $*location \leftarrow old\_value + 1$
4:     return $old\_value$
5: end function
```

Linked-Load (LL)/Store-Conditional (SC):
LL: First we load a value from the `addr` and link this read to a register (LR).
SC: Store to the `addr` is performed only when LR shows no one else stored to this location in between. This address many issues from previously discussed locks but still lacks fairness.

Ticket Lock: Assign a ticket number to each requester. When `my_ticket` equals to `now_serving`, the lock is acquired. This addresses the fairness issue but still causes a lot of cache coherence traffic.

Algorithm 5 Ticket Lock

```
1: function LOCK($my\_number, serving\_number$)
2:     $my\_number \leftarrow FAI(ticket\_variable)$
3:     while $my\_number \neq serving\_number$ do ;
4:     end while
5: end function
6: function UNLOCK($my\_number, serving\_number$)
7:     $serving\_number \leftarrow serving\_number + 1$
8: end function
```

Anderson Lock: To reduce the cache-coherence traffic, we can assign distinct wait variables to each requester (ensure that each of these variables are cache-aligned so that there is no false sharing). But here we statically allocate the space in this wait array which increases the memory footprint.

Algorithm 6 Anderson Array Lock

```
1: function LOCK
2: ▷ Size of the array = Maximum number of threads = N.
3:     $my\_index \leftarrow FAI(global\_index)\%N$
4:     while $ARRAY[my\_number] \neq 1$ do ;
5:     end while
6: end function
7: function UNLOCK
8:     $ARRAY[(my\_number + 1)\%N] \leftarrow 1$
9: end function
```

MCS Lock: Dynamically allocating nodes to the requester ensures less memory footprint. But inter-socket thread scheduling can lead to cache-to-cache data transfer which can increase the latency. This issue is address in NUMA aware locks.

NUMA-aware Lock: Here we prioritize the threads from our own socket first. But in long run to maintain fairness, one must also randomly entertain other-node requests (use a time-out).

Appendix II

There are multiple ways a thread can wait for the lock to be released. Here we discuss a few of them:

Busy-wait: This is the vanilla approach where the thread tries to acquire the lock repeatedly in a while loop. Over-subscription will lead to worse performance in general, as it will use up the CPU and not let the other threads run on the same CPU. So we expect over-subscription with this type of spinning will lead to long lock acquisition times for the threads.

Scheduled Yield: When the thread fails to acquire the lock, it performs a system call and relinquishes the CPU. The thread is removed from the running queue and is put at the end of the queue only to be run when it is next scheduled by the OS. This gives other waiting threads a chance to run.

Algorithm 7 MCS Lock

```
1: function LOCK($tail, my\_node$)
2: ▷ node has two members, pointer ‘next’ to another node, and a bool called ‘wait’.
3:     $(my\_node \rightarrow next) \leftarrow NULL$
4:     $previous\_node \leftarrow CAS(tail, tail, my\_node)$
5:     if $previous\_node \neq NULL$ then
6:         $(my\_node \rightarrow wait) \leftarrow 1$
7:         $(previous\_node \rightarrow next) \leftarrow my\_node$
8:         while $(my\_node \rightarrow wait) \neq 0$ do ;
9:         end while
10:    end if
11: end function
12: function UNLOCK($tail, my\_node$)
13: ▷ node has two members, pointer ‘next’ to another node, and a bool called ‘wait’.
14:    if $(my\_node \rightarrow next) == NULL$ then
15:        if $CAS(L, I, NULL) == I$ then return
16:            while $(my\_node \rightarrow next) == NULL$ do ;
17:            end while
18:            $(my\_node \rightarrow next \rightarrow wait) \leftarrow 0$
19:        end if
20:
```

Blocking Wait: To some degree this is similar to scheduled yield, where the thread performs a system call and goes to sleep. But here, the thread only runs again when it is woken up by someone. As CPU time counts the duration when the thread is active, this leads to a lower CPU time (but it still counts in wall clock time). The signaling mechanism is implemented using a condition variable where the owner thread has two notification choices:
notify_one(): For unfair locks like CAS, TAS, FAI, etc. owner thread can signal any one of the waiters. Although, in the case of fair locks, we need to use different condition variables to notify the correct waiter.
notify_all(): Wakes up all the waiters, leading to a race condition where there can be only one winner.

Active Back-off: Here the failed thread waits for a fixed amount of time (we use a bounded for-loop) before retrying for the lock. This reduces the network traffic every thread tries to acquire the lock at a lower rate. But this technique introduces several instructions in the pipeline, which will lead to increased CPU time and power consumption.

Passive Back-off: Intel (`pause`) and ARM (`yield`) are used to hint to the CPU that the thread is in a spin loop. Knowing this, the CPU lowers the power consumption and allows some of the resources to be reused by other threads (when HT).

Exponential Back-off: This is a perfect example of ‘learning from mistakes’. The wait time exponentially increases at every successive failure. This is achieved by using 2N number of pause instructions every time the thread tries to acquire the lock. We stop increasing the back-off time after a certain number of back-offs to give it a fair chance to acquire the lock in the future.

Random back-off: Here the thread backs off using random (4 to 1024) pause instructions before it retries.