Survey of *Spin-based* Synchronization Mechanisms

Sandeep Kumar
School of Information and Technology
Indian Institute of Technology Delhi
New Delhi, India
sandeep.kumar@cse.iitd.ac.in

***Abstract*—Synchronization mechanisms such as *Locks*, are used extensively in providing data consistency guarantees in a shared and distributed environment. These locks have their own characteristics in terms of design and waiting policies and have their associated pros and cons. The performance of these locks is of utmost importance, as in a production setting they handle millions of concurrent access to shared data. Because of this the selection of lock for a particular use case needs to be done carefully. There are different variants which can be used along locks, like *Composite* and *Fast-path* properties. Composite reduces the latency at a single object in high contention whereas, Fast-path allows for quick lock acquire during low contention. We define high and low contention in section I**

We analyze different type of *spin-based* locks in terms of their performance in low to high contention environment and discuss their keys design points which differentiate them from one another, followed by how these designs aspects affects their performance. We primarily focus on the performance of TAS, CLH Composite , Composite and CLH Fast-path lock [1]. Apart from these we also take a look at TTAS Lock, CLH Lock [2], [3] and MCS Lock [4] . By experimentation we try to quantify the performance difference among these locks based on different metrics and justify this based on their design.

***Index Terms*—TAS Lock, TTAS Lock, MCS Lock, CLH Lock ,Composite Lock , Lock Performance, Fast Path Lock.**

I. Introduction

Synchronizations method have been used extensively since the dawn of shared memory among processes or threads. Shared memory is used for inter process communication to speed up a given task by creating a logical division of work. Recent trend, where a single core is reaching its peak performance and hardware manufactures are resorting to adding more cores to make the chip faster, has brought the problem of synchronization (cache coherency) among CPU cores. The advent of cloud computing and its popularity has moved the traditional computing from one's local machine to a data center which is shared by thousands of users and handles requests in order of millions. It is of utmost importance that data of one of the user should be protected from other. Hence, the design of these locks should consider performance and scalability in mind, as using a non-scalable locks create issues when the number of threads crosses a limit. [5].

Locks can be classified into *spin-based* or *sleep-based* category. In former, the threads waiting to acquire the lock just keeps spinning on the lock to check if it has become free or not. It will try to acquire the lock as soon as it becomes free. Here the the waiting thread wastes CPU cycles as during the *spinning* phase it is not doing any work. The most simple version of this is a **TAS** locks, where the threads keeps *testing* a atomic lock, as shown in the listing 1 in appendix. It may or may not succeed as some other thread might set the lock before and takes the lock. There is chance that a particular thread might starve, i.e. it might never get a lock in high contention situation. Hence, TAS is NOT a starvation free lock, but CLH and MCS lock which are *queue-spin* based locks are starvation free. More details on these locks can be found in the appendix A.

In *sleep based* locks threads waiting for a lock does not keep waiting for it to become free, instead it registers itself with the lock and goes to sleep. When the thread which has the lock is done with it, release the lock and awakens the threads waiting for the lock. Based on the scheduling algorithm, a new thread will get the lock and rest all will go back to sleep. This saves a lot of CPU cycles are other process gets more time to run, as waiting threads are sleeping and not scheduled.

It seems like the *sleep based* locks are ideal candidates as they save the resources. However, sleep based locks are slow when compared to the *spin based* locks as there are multiple context switches involved. Therefore, if we expect a lock to be released in a time less than two context switches time, we go for spin based locks otherwise we opt for sleep based locks. In this paper we focus on *spin based* locks which are ideal when the contention is low and the size of critical sections of thread is small.

We evaluate the performance of the lock in high and low contention. A high contention is a situation where more than one thread is trying to acquire the lock. The number of threads that can run at one time, depending on the number of cores in the system is the upper limit on the contention [6]. Similarly, a low contention time is defined when a thread get a chance to run alone, without anyone competing for the lock at that particular time.

These locks can be augmented with properties like *Composite* and *Fastpath*, which improves the performance of locks in certain conditions. Composite property of the locks is typically used in queue-based locks to reduce the contention at the *tail* node. We compare the performance of CLH and CLH Composite lock in section III-F to see the effect of this property and show that indeed composite properties help improves the performance in high contention scenarios. *Fast-path* property of locks specify a "fast-path" which the threads can take in case of low contention. We evaluate the performance of CLH and CLH Fast-path in section III-D and show that when the

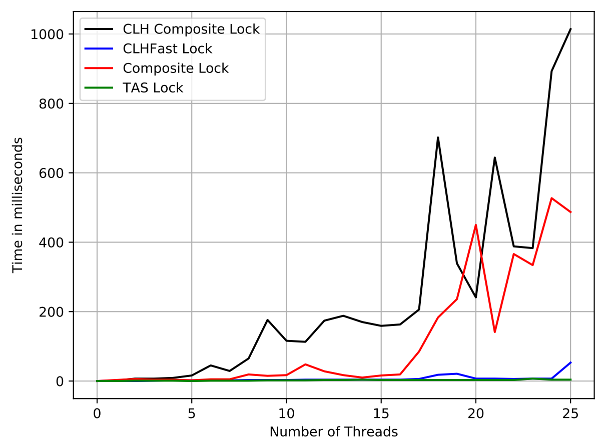


Fig. 1. Time to finish operation using Locks for 25 threads. This is dominated by lock acquire time as can be seen in figure 3. Each doing 10 operations. Average of 100 runs.

contention is low the performance improves by approximately 10% than the corresponding "non-fast-path" version.

II. Experiments and Evaluation

All of these locks have different characteristics. Some are easy to implement, but face too much contention on a single variable, others are starvation free but involves complex lock acquire steps. So makes too many read calls to a remote object variable in NUMA. We use different metrics for the comparison of locks to make a fair comparison.

1) Average lock acquire time: The performance of lock is of utmost importance, hence we compare the average lock acquire time for different locks, under different scenarios. Specific details can be found in the sections III-B - III-F.
2) Write vs Read Calls (Cache Coherence): Cache Invalidation because of unnecessary writes to a shred variables. Any writes to shared variables by a thread forces other cores to invalidate their local cached copy (even if the same value was written) and then re-read the data.
3) Contention. To measure the contention, we use *miss-rate* heuristics as a measure of contention in the system. [7].
4) Remote NUMA calls: In case of NUMA architecture, calls to remote memory and caches are costly compared to calls made within the local socket. We use *perf c2c* tool to measure the number of remote calls made.

During experiments we saw that time taken during lock release is approximately similar for all the locks as it involves few deterministic steps. Hence, we focus on lock acquire time as the basis for lock comparison. Behavior of *throughput* of the locks is similar to lock acquire (figure 1), as the lock acquire is the takes the most time. Critical Section is a counter increment, and lock release is a few deterministic steps.

A. Experimental Setup

For our experiment evaluations we are using a machine with 48 cores configured with 2 NUMA nodes, each having 24 cores. It is configured with 96 GB of memory which makes sure that all the experiments are done in memory only and there is no swapping involved. It is running Centos, Linux kernel version 3.18. All the lock implementation is done in Java which is taken from [8]. To measure the time taken to acquire the locks we rely on the tools provided by Java to measure times before and after a function call. To measure the remote socket calls (memory or cache) and highest contended cache line, we use the tool *perf c2c* [9]. Python is used to plot the graphs using *matplotlib*. In some of the plots the x-axis index starts from 0. In the interest of time, we add a zero to every data at index 0 so the actual result starts from x=1. Complete Code can be found here: https://github.com/sandeep007734/Lock-Comparison

```
void run() {
  barrier.await();
  for (int i = 0; i < RUNS; i++) {
    lock_acuire()
    counter = counter + 1;
    lock_release();
  }
}
```

Listing 1. "Critical Section of the threads."

The critical section protected by these locks is small and involves incrementing a shared variable as shown in the code listing 1. The value of this variable is used to check the correctness of lock implementation. Threads try to acquire the lock a fixed number (varies per experiment) of time to do operation. We use barrier to ensure that after creation all of the threads starts from the same point at same time. As these locks have different characteristic, to test their performance different settings were used. For example, to study the effect of fast-path we keep the number of threads ¡ 10, as its effect can be seen during low contention only. We test for maximum of 200 and 500 threads, as to see the effect of contention, thread count greater than number of cores in the machine is sufficient [6].

B. OS Jitter

Measuring the performance of "performance critical components" on a general purpose Operating system is affected by jitter in OS because of system calls, context switches, interrupts and other processes running [10]. There are different ways to handle this. First is using a light weight kernel [11] where kernel operations are limited. However, they limit the capabilities of the operating system and can be used for specific applications only. Advent of multiprocessor and multi core chips allow a way to handle this by dedicating a core to handle OS related activities which affects the rest of the operation in the minimal way [12]. Also, Non determinism present in the scheduler makes the time to acquire a lock vary by a factor 10-50.

To limit the effect of OS jitter, we average out the reading of a particular metric recorded over multiple runs. Running the lock in a continuous loop allows the system to reach a stead state [13].

III. Results

A. TAS vs CLH Composite vs Composite vs CLH Fast-Path Lock

We compare the performance of TAS, CLH Composite, Composite and CLH Fast-Path lock for different number of threads. The performance comparison for them can be seen in the figure 3.

The performance of TAS lock remains somewhat the same throughout the experiment. This is because of its simple implementation which does not add much over-head going from 25 to 200 threads.

Composite locks have an *waiting* array defined, where threads wait before trying to acquire the locks. The idea is that having a waiting array, the access at the *tail* for queue based locks is reduced in case of high contention. The size of the array is a configurable parameter, which can be changed as per the use case. We experimented with different array size as can be seen in the figure 2. If we use a small size array, the thread will compete for a position in the array and even though the system is capable of handling more threads, threads equal to the size of array will only get to compete for lock and this will impact the lock acquire time. If we use a large value for array size then most of the threads will get to compete for the lock, essentially defeating the purpose of using a Composite lock. Hence there has to be a balance between the array size and the contention.

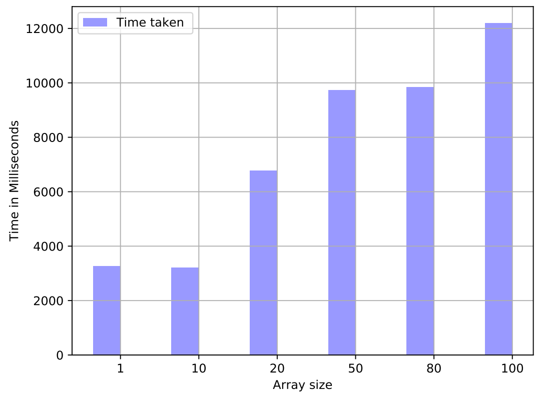


Fig. 2. Performace of Composite Lock on varying the array size. 100 threads doing 10 operations each. Average of 100 runs.

Comparing variants of CLH, for 25 threads, the performance of Composite and CLH Composite is approximately 5x-7x slower than the CLH Fast-path. This is because of extra overhead associated with composite forms of locks, where threads first contend for place in *waiting* array. In CLH-Fastpath, some of the threads can take *fastpath* to reduce the lock acquiring time. However, as the number of thread increases, the performance of CLH-Fastpath lock is order or magnitude worse than its Composite counterparts because of the increased contention at the *tail* . Also, when 200 threads are contending for lock, it is unlikely that any of them takes the fast-path.

We compare the performance of CLH with CLH Composite and CLH-Fastpath in section III-F and III-D respectively to get more understanding on the property of Composite and Fast-Path properties.

B. TAS vs TTAS

TAS *(Test and Set)* and TTAS *Test- Test and Set* are most simple synchronization mechanism compared to other locks. However, they are not starvation free and there is severe contention at a single memory object when many threads are trying to acquire the lock.

The number of writes done in case of TAS is 20x more than TTAS (figure III-B), as each thread while waiting, writes the value *true or 1* on the lock and if the old value was *false* it gets the lock. Because of these writes, in case of multiprocessor, the cache is invalidated at other cores and the value has to be fetched from the main memory again. This increases traffic on the interconnect of memory to cores. TTAS on the other hand, before writing a *true* on the lock checks whether it is false or not. It writes only if the value is false. It might happen that some other thread writes runs away with the lock (sets the value to 1 before this thread gets a chance), between the checking and the writing. Checking the value of the lock before writing it, reduces the number of cache invalidation.

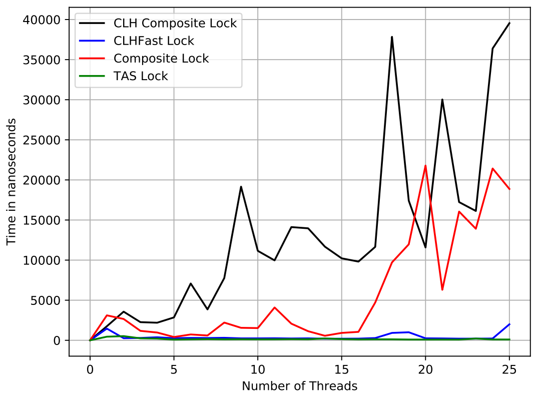
As seen in the figure 4, where bar chart represents the number of write calls, and line graph measures the contention on the ”highest” contended cache line as reported by the tool *perf*. We can see that with increase in number of threads from 1 to 200, write calls increases to 2217 for TAS and to 122 for TTAS and the contention increases by a factor of 7 for TAS and remains same for TTAS. This result in approximately 5x performance overhead on lock acquire time as seen in the figure 5.

C. CLH vs MCS

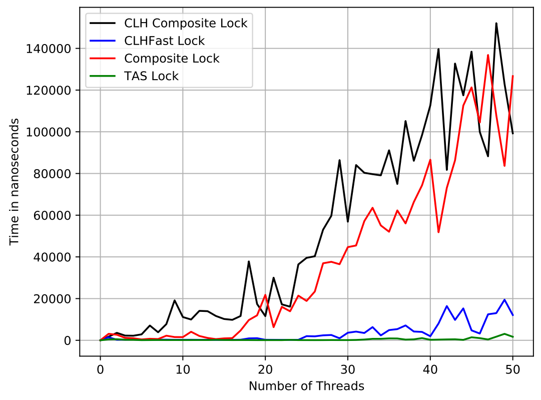
CLH and MCS lock are both queue based spin lock, which guarantees freedom from starvation. Altough similar in nature, as mentioned in the appendix A, the difference is that CLH Lock wait on a remote field, which belongs to its predecessor in the queue and MCS waits on a local field, which is modified by its predecessor when the lock is released. Using the *perf-c2c* [9] tool we measure the number of remote calls (calls to another socket) made to during the a program execution by varying the number of threads. The results are sown in the figure III-C. We can see that number of remote calls made by the program while using CLH lock is higher compared to that made when using MCS lock by a factor of 2 for 500 threads. These remote costs are costly compared to calls made to local socket [14], because of which the average lock acquisition time increases by a factor of 4 for 25 threads, and by order magnitude for 200 threads as seen in the figure 6.

D. CLH and CLH Fastpath Lock

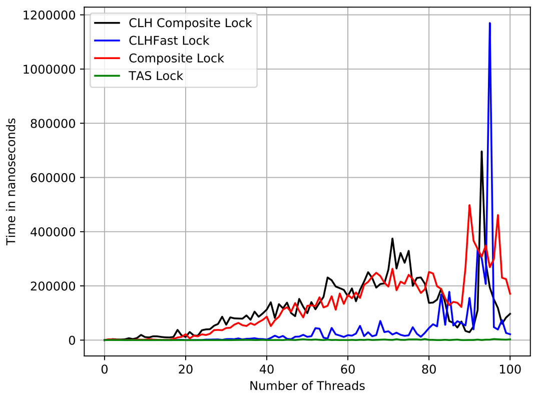
The main idea behind a fastpath lock is to acquire the lock quickly when the contention is low. We compare the performance of locks speed in environment, where the contention is



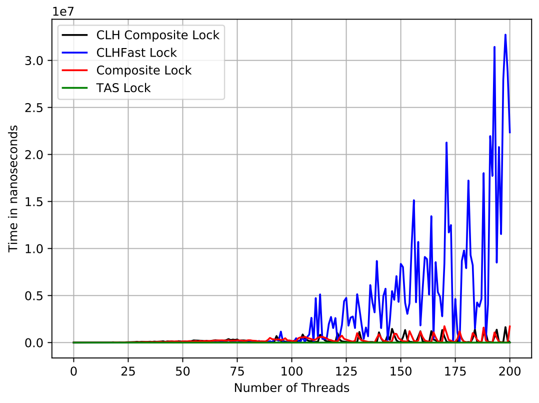
(a) Performance for 25 threads.



(b) Performance for 50 threads.



(c) Performance for 100 threads.



(d) Performance for 200 threads.

Fig. 3. Average lock acquire time: Performance comparison of TAS, CLH Composite, Composte and CLH Fast-Path lock for different number of threads. Each thread doing 10 operations. Average of 100 runs.

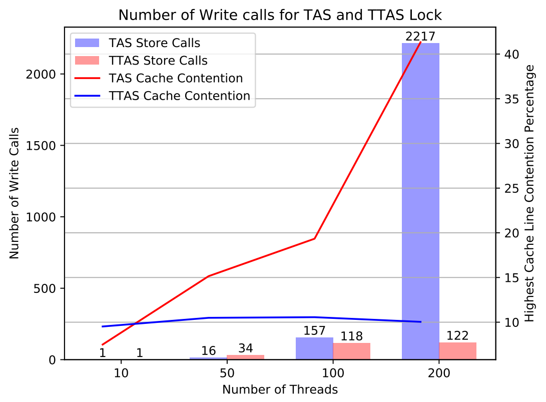


Fig. 4. Increase in Write calls with increase in the number of threads between TAS and TTAS Lock. Bar chart showing number of write calls made and the line chart showing the highest contention cache line.

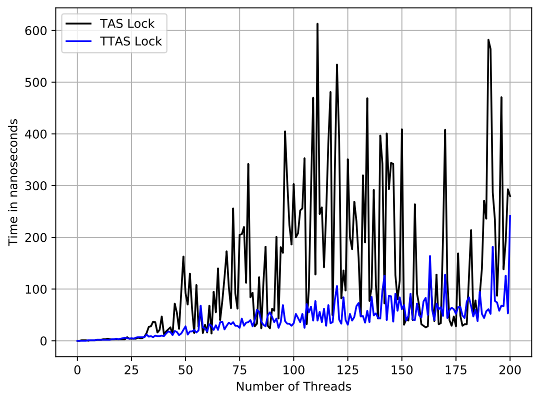
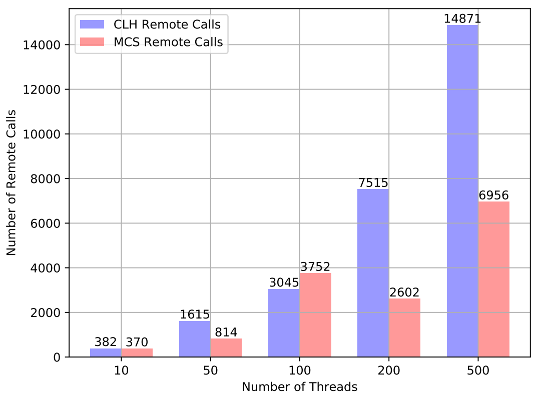
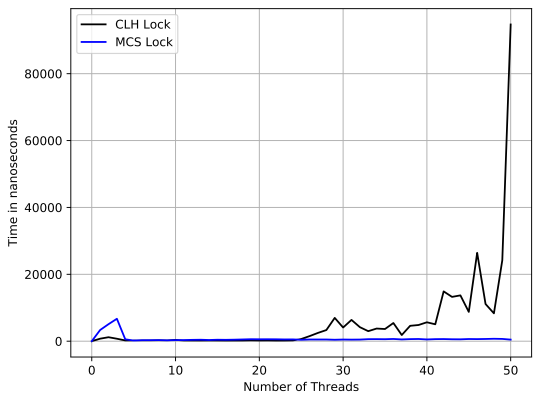


Fig. 5. Average lock acquire time: TAS Lock as compared to TTAS Lock for 200 threads each doing 10 operations. Average of 100 runs.

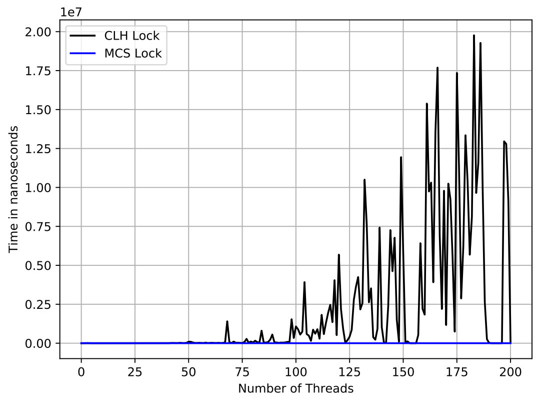
low in the beginning and then gradually increase in number. The results are shown in the figure 7 where the number of threads are increases from 1 to 10. Initially we see the fast-path lock working better than the CLH lock by 10%, however, the performance becomes similar during high contention as both lock follows the same code path. In fact fast-path is doing little bit of extra work of checking every time whether a ”fastpath” lock acquisition is possible or not.



(a) Comparison of Remote calls (including cache and memory) made by program when using CLH Lock and when using MCS Lock for varying number of threads.

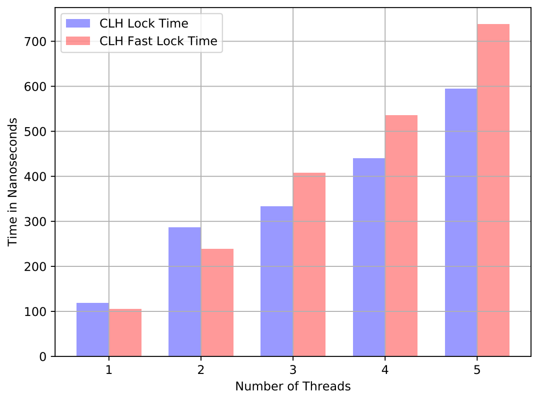


(b) Performance Comparison of CLH Lock and MCS for Lock for upto 50 threads.

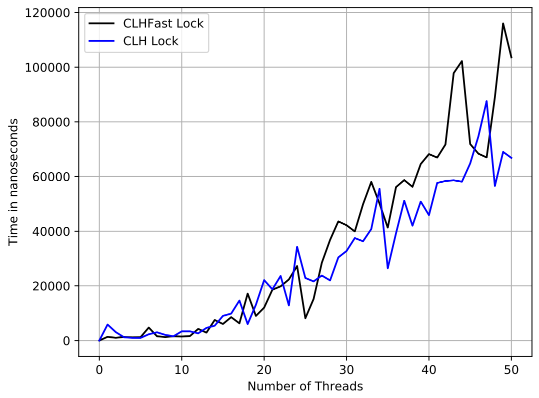


(c) Performance Comparison of CLH Lock and MCS for Lock for 200 threads. Performacne of CLH keeps worsening going forward.

Fig. 6. Comparison of Performance of CLH and MCS Lock.



(a) Performance comparison of CLH and CLH Fastpath Lock. Time to acquire locks with changing contention. CLH Fastpath lock performs better when contention is low.



(b) Performance comparison of CLH and CLH Fastpath Lock. Performance remains same with increase in contention as the lock is seldom free for the thread to take fastpath.

Fig. 7. CLH VS CLH Fastpath Lock

E. Composite and Composite Fastpath Lock

In case of Composite and Composite Fast-path we seee similar patterns as seen in between CLH and CLH-Fastpath as seen in the figure 8. When the contention is low, in fast-path the lock acquire time is 4x faster (fig. 8 part (a)). However, as the contention increases, performance of both remains the same, (fig. 8 part (b)) as it is unlikely that any thread gets to take the fast-path.

F. CLH and Composite CLH Lock

Composite locks are used to reduce the contention on the lock itself, as the thread first fight for a place in array which can acquire lock. To see the effect of using composite locks we use the tool *perf c2c* which, for a program, gives total hits on every cache line used. For MCS and CLH, during high contention, the cache line storing the *tail* value will have the highest contention and therefore the highest amount of hit percentage. So we compare this value in both of the locks to see how much the hit percentage goes down by using composite lock.

IV. Conclusion

In this paper we have explored different variations of Spin based locks. We have seen simple locks such as TAS and TTAS which do not guarantee fairness is faster compared to queue-based spin locks such as MCS, CLH and Composite Locks which guarantees starvation freedom. We saw the TAS Lock causes cache invalidation because of the lock write at every wait loop, which is reduced in TTAS lock as can be seen in the figure 4. This is because in TTAS the write is only done after checking the current value of the lock.

Similarly when comparing MCS and CLH lock, in a NUMA machine with 2 NUMA nodes, we observe (figure 6) that CLH performs order of magnitude worse than MCS because of remote calls made by it to check the state of its predecessor. We also suspect that performance of CLH can be because of unoptimized implementation and can be improved upon. This part needs further exploration.

Finally we explore the performance when using Composite and Fast-pat properties. In composite, we saw that, when the contention is high, a waiting queue helps in the performance of the locks as the thread first competes for a position in the waiting array and after that competes for lock acquirement. Fast-path helps when the contention is low, and the thread can skip the overhead associated with queue-based locks and take a ”fast-path” in code for lock acquisition.

We see that all of these locks have different properties and care should be taken before using them for a particular workload. Most of the language abstract away these details behind an interface to ease the process of development. However, selecting a wrong lock can lead to scalability and performance issues.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[2] T. Craig, “Building fifo and priority-queuing spin locks from atomic swap,” tech. rep., 1993.

[3] P. S. Magnusson, A. Landin, and E. Hagersten, “Queue locks on cache coherent multiprocessors,” in *Proceedings of the 8th International Symposium on Parallel Processing*, (Washington, DC, USA), pp. 165–171, IEEE Computer Society, 1994.

[4] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, Feb. 1991.

[5] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, “Non-scalable locks are dangerous,” *Ottawa Linux Symposium (OLS)*, pp. 1–12, 2012.

[6] V. J. Marathe, M. Moir, and N. Shavit, “Composite abortable locks,” in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 1–10, April 2006.

[7] S. Blagodurov, A. Fedorova, S. Zhuravlev, and A. Kamali, “A case for numa-aware contention management on multicore systems,” in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 557–558, Sept 2010.

[8] “Art of Multiprocessor Book Website.” https://booksite.elsevier.com/9780123705914/?ISBN=9780123973375. [Online; accessed 27-10-2017].

[9] J. mario, “C2c - false sharing detection in linux perf.” https://joemario.github.io/blog/2016/09/01/c2c-blog/, 2016. [Online; accessed 27-10-2017].

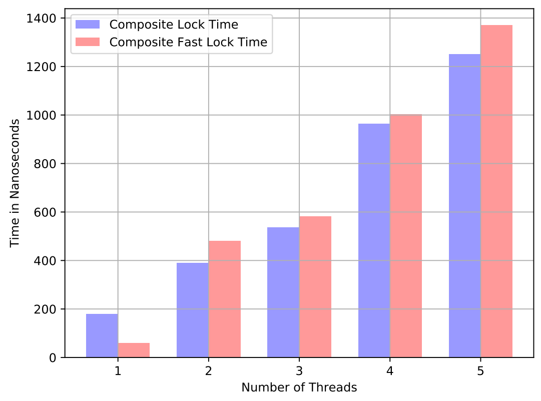
[10] P. De, R. Kothari, and V. Mann, “Identifying sources of operating system jitter through fine-grained kernel instrumentation,” in *2007 IEEE International Conference on Cluster Computing*, pp. 331–340, Sept 2007.

[11] J. Laros, C. A Segura, and N. Dauchy, “A minimal linux environment for high performance computing systems,” 04 2006.

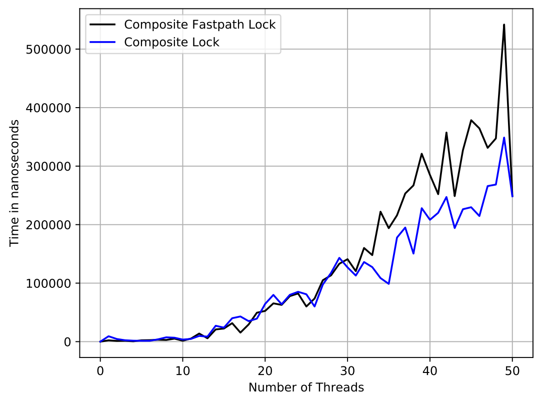
[12] P. D. V. Mann and U. Mittaly, “Handling os jitter on multicore multithreaded systems,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–12, May 2009.

[13] J. F. Barhorst and R. W. Seelye, “Ada run-time system contention measurement,” in *Proceedings of the Conference on TRI-ADA ’90*, TRI-Ada ’90, (New York, NY, USA), pp. 334–338, ACM, 1990.

[14] T. Brown, A. Kogan, Y. Lev, and V. Luchangco, “Investigating the performance of hardware transactions on a multi-socket machine,” in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’16, (New York, NY, USA), pp. 121–132, ACM, 2016.

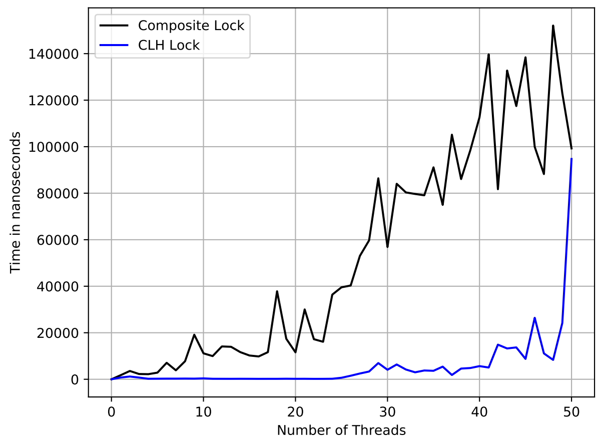


(a) Performance comparison of Composite and Composite Fastpath Lock. Performance remains same with increase in contention as the lock is seldom free for the thread to take fastpath.

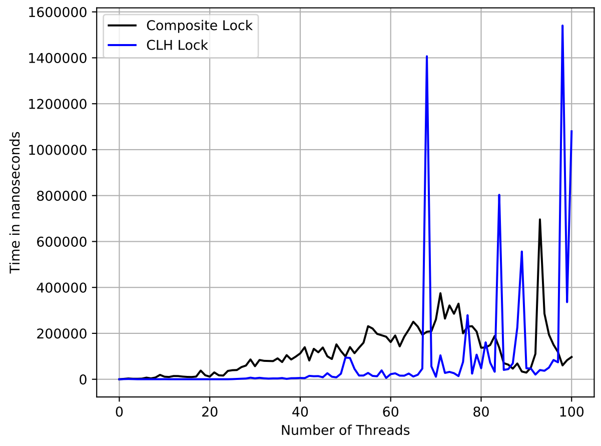


(b) Performance comparison of Composite and Composite Fastpath Lock. Performance remains same with increase in contention as the lock is seldom free for the thread to take fastpath.

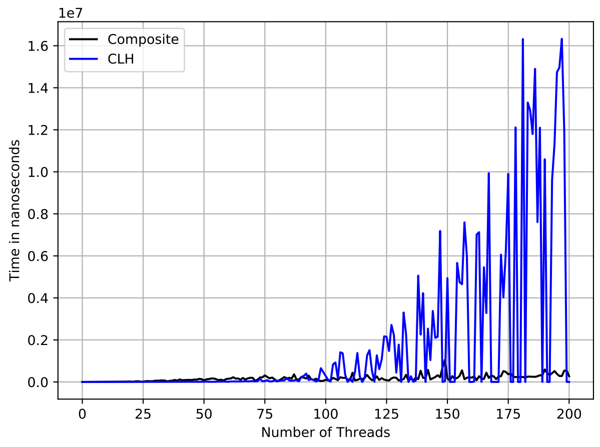
Fig. 8. Composite VS Composite Fastpath Lock



(a) CLH vs CLH Composite. CLH Lock performs better in low contention because of extra overhead associated with CLH Composite Lock.



(b) As the contention increase the performance of CLH and CLH Composite Locks become similar.



(c) In high contention, the Composite CLH locks performs much better the CLH lock because on the upper bound on the number of threads that can try to get the lock at a given time.

Fig. 9. Performance Comparison of CLH and CLH Composite Locks.

Appendix

We briefly explain the locks, their design, how they work, how they are different and their use cases.

A. TAS Lock

TAS or *Test and Set* is one of the most simple and easy to implement locks. It just uses a single boolean atomic variable. if the boolean value is set, this means the lock is acquired by some thread, and the new thread uses get_and_set instruction to set the value to true and get its old value. It waits till it is getting true (lock is acquired). The pseudo code for TAS lock is given in algorithm 1.

Algorithm 1 TAS Lock
```
1: procedure LOCK(*lock)
2:     while $test\_and\_set(lock) == 1$ do
3:         repeat;
4: procedure UNLOCK(*lock)
5:     $lock = 0$;
```

As shown in the code above, while locking, if multiple entities tries to set the *lock* to 1, only one of them will succeed and the rest will see 1 and keeps on waiting.

Some of the features of TAS Lock:

1) **Simple and Easy to Implement**: It is one of the most easiest to implement locks, provided that hardware support is present.
2) **Limited Use case**: This is used when we expect the waiting time to be very less and low contention for a lock.
3) **Not fair, Threads can starve**: It is not fair in its current format as once the lock is available, any thread can get it and there is no concept of queue.
4) **Cache Coherency**: As all of the threads wait on a single atomic variable, it creates an issue of cache coherency in case of multiprocessor architecture, where the local cached value of variable is invalidated at each $test\_and\_set$ instruction. A more refined approach is TTAS Lock where the number of writes to the atomic variable is very less.

B. TTAS Lock

TTAS lock as shown in the algorithm 2 is much similar to the TAS Lock with one crucial difference. TAS lock writes to the *lock* value at each iteration and if it finds the old value to be 0 it gets the lock. However in case of TTAS lock, it first checks the current value of *lock* in line number 3 and when it finds it to be 0 it tries to acquire the lock by setting the value to 1. Now, it might happen that some other threads get the lock in between, in the case it just tries again from the beginning line 6.

Reading the value of lock before writing it avoid lots of unnecessary cache invalidation (because of writes) and hence it performs better than TAS lock. However, it needs support for *test* instruction from the hardware.

We do an extensive comparison of TAS and TTAS lock in section III-B.

Algorithm 2 TTAS Lock
```
1: procedure LOCK(*lock)
2:     retry:
3:     while $test(lock) == 1$ do
4:         repeat;
5:     if $test\_and\_set(lock) == 1$ then
6:         goto retry
7: procedure UNLOCK(*lock)
8:     $lock = 0$;
```

C. Composite Lock

In a queue based spin lock, which provides first come first serve, fast lock release and low contention, suffers when it comes to recycling abandoned nodes because of time-out. The other way it to use a an exponential back-off algorithm, which provides a simple way for thread to exit but is not scalable.

The key insight used in *composite lock* is that if a thread is way behind in the queue, then it should not have to worry about the node abandoning. The threads closer to the head of the queue should be the one to perform the hand-off of the lock.

Algorithm 3 Composite Lock: Overview
```
1: procedure LOCK
2:     Qnode node = acquireNode();
3:     Qnode pred = joinQnode();
4:     wait(pred, node);
5: procedure UNLOCK
6:     Qnode acqnode = myNode.get();
7:     acqNode.state = RELEASED;
8:     myNode.set(null);
```

D. CLH Lock

CLH[2], [3] lock is a queue based spin lock which provides a fair starvation free synchronisation mechanism. As seen in algorithm 7. Every thread which wants to acquire the lock joins the waiting queue using *genAndSet* operation on the tail of the list, and then it waits for its predecessor to finish it operations by spinning on a field in its predecessor node. It get the lock as soon as its predecessor is done.

Unlocking is simply setting its own locked state as *false* and the successor will take notice and get the lock. After releasing the lock, for all future purpose it will use its predecessor's node and not its own (line 11). This is done to avert a problem known as *ABA* problem. This is the case where a thread which just released the lock, tries to acquire the lock again and gets it without going through the queue because its node is already in the beginning of the queue.

A problem with CLH lock is that it is waiting on a field which belongs to its predecessor's node. This means

Algorithm 4 Composite Lock: AcquireNode

```
function ACQUIRENODE
    while true do
        node = waiting[random()];
        if node.state.CAS(FREE, WAITING) then
            return node;
        ⟨ctail, stamp⟩ = tail.get();
        if node.sate==(ABORTED or RELEASED) then
            if node==ctail then
                QNode myPred = null;
                if node.state == ABORTED then
                    mypred = node.pred;
                if tail.CAS(ctail, myPred, stamp, stamp+1) then
                    node.state = WAITING;
                    return node;
        if timeout() then
            throw Exception();
```

Algorithm 5 Composite Lock: JoinQNode

```
function JOINQNODE(node)
    do
        ⟨tail, stamp⟩ = tail.get();
        if timeout() then
            node.state = FREE;
            throw exception();
    while (!tail.CAS(ctail, node, stamp, stamp+1))
    return ctail;                ▷ Predecessor of the new tail
```

Algorithm 6 Composite Lock: wait

```
function WAIT(pred, node)
    if pred == null then
        myNode.set(node);
        return ;
    State ps = pred.staet();
    while $ps \neq RELEASED$ do
        if ps==ABORTED then
            QNode temp = pred;
            pred = pred.pred;
            temp.state = FREE;
        if timeout() then
            node.pred = pred;
            node.state = ABORTED
            throw exception()
        ps = pred.state;
    pred.state = FREE;
    myNode.set(node);
    return
```

making remote calls to read a value. In case of NUMA this might create a problem as its predecessor thread might be scheduled on a different NUMA node which will degrade its performance.

Algorithm 7 CLH Lock

```
procedure LOCK
    Qnode qnode = myNode.get();
    qnode.locked = true;
    QNode pred = tail.getAndSet(qnode);
    myPred.set(pred)
    while pred.locked do
        repeat
procedure UNLOCK
    Qnode qnode = myNode.get();
    qnode.locked = false;
    myNode.set(myPred.get());
```

E. MCS Lock

MCS like CLH lock is a queue based spin lock, which guarantees fairness and hence is starvation free. Just like CLH Lock, a thread which want to get the lock attach itself to the queue at the end of the queue and waits for it predecessor to finish. The key difference from CLH lock is that here instead of waiting on the remote field of the predecessor's node it waits on its own local field. It is the job of the predecessor to change it while releasing the lock.

This subtle changes removes a lots of remote calls and and improves the performance of the lock as it is spinning on a local variable. We compare the performance of MCS lock with CLH lock in section III-C.

Algorithm 8 MCS Lock

```
procedure LOCK
    Qnode qnode = myNode.get();
    QNode pred = tail.getAndSet(qnode);
    if $pred \neq NULL$ then
        qnode.locked = true;
        myPred.next = qnode;
        while qnode.locked do ▷ Differs from CLH Lock
            repeat
procedure UNLOCK
    Qnode qnode = myNode.get();
    if qnode.next == NULL then
        if tail.compareAndSet(qnode, NULL) then
            return;
        while qnode.next == NULL do          ▷ small wait
            repeat;
    qnode.next.locked = false;
    qnode.next = null;
```

F. Fast Path Lock

The basic idea in a fast-path lock is that it provides a way to acquire the lock in low contention by taking a "fast-path" in the lock. It uses the highest bit in the tail of the queue based lock to indicate whether a node has acquired fast-path or not, as threads coming for this need to account for their presence. A thread taking the fast-path for lock acquire also uses a fast-path for unlock operation.

Algorithm 9 Fast Path Lock

```
 1: function FastPathLock
 2:     FP = $1 \ll 30$
 3:     $\langle anode, stamp \rangle = tail.get()$;
 4:     if qnode != null then
 5:         return false
 6:     if $stamp$ & $FP$ $! = 0$ then
 7:         return false
 8:     newstamp =$(stamp + 1)|FP$;
 9:     return tail.CAS(qnode, null, stamp, newStamp);
10: function Lock
11:     if FastPathLock() then
12:         return true
13:     if SlowLock() then ▷ The usual way to getting a lock
14:         while $tail.getStamp()$ & $FP$ $! = 0$ do
15:             repeat
16:         return true
17:     return false
```