Adaptive Lock

Madhav Iyengar <miyengar@andrew.cmu.edu>, Nathaniel Jeffries <njeffrie@andrew.cmu.edu>

ABSTRACT

Busy-wait synchronization, the spinlock, is the primitive at the core of all other synchronization constructs. The low latency access spinlocks give to small critical sections is ideal for several different use cases, from systems to High-Performance Computing applications. Unfortunately, the most ubiquitous spinlocks, built on atomic test-and-set operations, do not scale with increased core count and can be unboundedly unfair. The landmark paper by Mellor-Crummey and Scott addressed these issues with a local-spinning queue-based spinlock. However, simpler global spinning locks have better performance under low contention. We propose an adaptive spinlock based on a combination of the ticket and queue based spinlocks. Our lock dynamically transitions between global and local-spinning spinlock functionality based on contention. We show that our lock provides the lower latency of a simple ticket lock under low contention and the scaling of the queue-based lock under high contention. We also show that it can provide superior performance to the queue-based lock under cases of variable contention on a critical section, while running on the Intel Xeon Phi.

INTRODUCTION

Spinlocks are a fundamental building block for most types of synchronization used to lock critical sections today. They are lightweight and avoid the overhead of syscalls, making them ideal for low latency access to short critical sections. Spinlocks are ubiquitous in operating systems, where spinlocks offer a simple way to share critical data among threads operating on different cores. In addition, their low latency properties make them ideal for High-Performance Computing (HPC) applications, as it is important to minimize synchronization-related overheads.

The most common form of spinlocks is a test-and-set lock, involving on contending threads spinning on an atomic exchange operation. This type of implementation can be found in operating systems such as Linux[4], as well as in the synchronization constructs exposed by several APIs including pthreads and OpenMP[5]. However, test-and-set locks are poorly scaling, with coherence traffic (Remote Memory Accesses (RMA) in distributed memory systems) scaling superlinearly with contending processors. It has been shown that the poor scaling of test-and-set locks can cause system performance to suddenly drop as core counts increase[2]. For HPC applications, the additional synchronization traffic not only increases the cost of synchronization but can also impact the performance of portions of a parallelized algorithm not involved in the synchronization due to the additional network latency.

Attempts to address this include test-and-test-and-set (TTS) locks, which spin in an inner loop on reading the shared-memory location, and exponential back-off, which reactively decreases contention[1]. Unfortunately, both of these methods exacerbate the inherent unfairness of the test-and-set lock, making it more unlikely for threads waiting longer to gain access to a critical section. This can significantly impact system performance or the progress of an HPC application. It is clear that more scalable and more fair synchronization is necessary to allow systems and applications to scale to higher core counts while obtaining greater performance.

In this paper, we present Adaptive Lock, a new fair spinlock which adapts its functionality to the current contention level on a particular critical section. The goal of the Adaptive Lock is to achieve low latency under low contention (no more atomic operations than a test-and-set lock) while providing optimal scaling under high contention.

RELATED WORK

Anderson gave an analysis of several spinlock variants in his 1990 publication[3]. He proposed various optimizations, including spinning on reads, such as with the test-and-test-and-set locks, and adding various delays, such as static and exponential back-off, to reduce interconnect traffic. Additionally, he proposed a new, more scalable, lock which uses a global array of flags, indexed by unique tags obtained through a single atomic operation, to enable processors to spin on different locations.

Mellor-Crummey and Scott also proposed new fair and scalable spinlock variants in their landmark paper published in 1991[1]. They asserted that significant memory and interconnect network contention are not fundamental properties of busy-wait synchronization. They proposed new scalable spinlock and barrier implementations, analysing them against several state-of-the-art busy-wait mutual exclusion and barrier variants. We seek to expand on the ticket and queue-based locks proposed in that paper.

Ticket Lock:

The ticket lock comprises of two counters, ticket and turn. The lock operates on a simple concept, where each locking thread is fetches and increments the ticket variable, obtaining a unique ticket, and spins until the lock's turn variable reaches the value of the ticket. This guarantees fifo ordering of access to the critical section. Additionally, locking requires only a single atomic operation on lock, with the spin involving only reads, thus improving on the interconnect traffic of a test-and-set lock. We give the pseudo-code for the algorithm below (we do not indicate architecture and compiler specific barriers and flags for generality):

```
struct ticket_lock:
        u64 ticket;
        u64 turn;

lock(ticket_lock * lock):
        u64 ticket = atomic_fetch-and-add(&lock->ticket, 1);
        while (lock->turn != ticket);
```

```
unlock(ticket_lock * lock):
        lock->turn++;
```

Queue-based Lock:

The queue-based lock proposed by MCS involves an additional local variable. The lock itself is a pointer to the local variable of the most recent thread which is spinning on (or holding) the lock. On locking, a thread attempts to exchange a pointer to its local variable with the lock, obtaining either NULL if the lock is free or a pointer to its predecessor. If there is a predecessor, the thread notifies it and proceeds to spin on its local variable until the predecessor releases it, thus reducing interconnect traffic on locking to a single atomic operation and a write. On unlock, the thread attempts atomically to mark the lock as free, however, if there is a predecessor, it synchronizes with and releases it before returning. The pseudo-code is given below:

```
struct queue_node:
        queue_node * next;
        bool spin;

struct queue_lock:
        queue_node * tail;

lock(queue_lock * lock, queue_node * qnode):
        *qnode = {NULL, 0}; // initialize qnode

        // exchange with tail, if NULL, got lock
        queue_node * pred = atomic_exchange(&lock->tail, qnode);

        if (pred) // did not obtain lock, alert predecessor and spin
                qnode->spin = 1;
                pred->next = qnode;
                while (qnode->spin);

unlock(queue_lock * lock, queue_node * qnode):
        if (!qnode->next) // check for successor
                if (atomic_compare-and-swap(&lock->tail, qnode, NULL))
                        return; // no other locker, lock is unlocked

                while (!qnode->next); // wait for successor to notify
        qnode->next->spin = 0; // release successor
```

ADAPTIVE LOCK

Though the queue lock is very scalable, it performs less well under low contention, due to an additional atomic operation on the critical path. In this case, the ticket lock, though less scalable, does provide lower latency. This appears to be an opportunity to design an adaptive mechanism which can provide the benefits of both locks based on the current level of contention. There are some key considerations with regards to the nature of the solution. First, implementing adaptive behaviour cannot add significant latency. This is because the latencies involved are already very low, involving at most tens of instructions. Even a few additional arithmetic operations can have a large impact. More importantly, there can be no more atomic operations on the critical path of locking and unlocking than there are in the component lock implementations, as these involve expensive memory fences. The implementation must also take into account fairness. In order to create a new lock, there cannot be any aspect of reordering entry to the critical section, which also implies that the functionality of both locks cannot be introduced in serial, as there is no guarantee of ordering between. Finally, the implementation must have a reasonable method of estimating the current contention on the lock.

In order to address these considerations, our Adaptive Lock implementation works by adding ticket lock state to the queue lock. Essentially, the lock itself includes turn while the local node includes ticket, allowing for a ticket lock spin. On locking, a thread behaves similarly to the queue lock unless it does not gain immediate access. In this case, the thread obtains a ticket by obtaining and incrementing the ticket of its predecessor. By subtracting the lock turn variable from this new ticket, the thread obtains an estimate of the current contention. If the contention estimate is above a specified threshold, the thread spins as it would in the queue lock due to high contention, whereas if it is below, the thread spins on ticket not equaling turn, as with the ticket lock.

The unlocking path is where our implementation attempts to surpass the queue lock. The objective is essentially to eliminate the atomic operation in unlock from the critical path in situations of low contention. Thus, an unlocking thread first increments the lock's turn. This immediately allows the next thread, if it is spinning on turn due to low contention, to proceed into the critical section before the unlocking thread reaches the atomic operation. Afterwards, the unlocking thread proceeds to unlock identically to the queue lock functionality. One issue here is that a thread given access to the critical section could attempt to unlock before the unlocking thread reaches the atomic operation. This is mitigated by forcing this thread to do a queue spin after incrementing turn in the unlock function. Pseudo-code for this algorithm is given below:

```
struct hybrid_qnode:
        hybrid_qnode * next;
        u64 ticket;
        bool spin;
        bool valid;

struct hybrid_lock:
        hybrid_qnode * tail;
        u64 turn;

lock(hybrid_lock * lock, hybrid_qnode * qnode):
        *qnode = {NULL, 0, 0, 0}; // initialize qnode

        // exchange with tail, if NULL, got lock
        hybrid_qnode * pred = atomic_exchange(&lock->tail, qnode);

        if (pred) // there is a previous locker
                // get ticket and notify predecessor
                qnode->spin = 1;
                while (!pred->valid); // wait for predecessor to have valid ticket
                qnode->ticket = pred->ticket + 1;
                qnode->valid = 1; // allow a successor to read ticket
                pred->next = qnode; // notify predecessor

                // choose ticket or queue lock behavior based on contention
                if (lock->turn - qnode->ticket > THRESHOLD)
                        while (qnode->spin); // high contention, qnode spin
                else
                        while (lock->turn != qnode->ticket); // low contention, ticket
spin
        else // no previous locker, got lock on xchg
                qnode->ticket = lock->turn;
                qnode->valid = 1; // allow successor to read ticket

unlock(hybrid_lock * lock, hybrid_qnode * qnode):
        lock->turn++; // immediately release successor spinning on turn
        while (qnode->spin); // sync with unlocking predecessor if skipped queue spin

        if (!qnode->next) // check for successor
                if (!atomic_compare-and-swap(&lock->tail, qnode, NULL))
                        return; // return if no successor, lock is unlocked

                while (!qnode->next); // wait for successor to notify
        qnode->next->spin = 0; // completely release successor
```

TESTING METHODOLOGY

We tested our hybrid lock on the Intel Xeon Phi with two key workloads. First, we wrote a tight loop with synchronization, where a fixed amount of work is done inside and outside of the critical section. Next, we formulated a micro-benchmark to represent a typical real-world style workload where contention varies as the program runs. To do this, we varied the amount of time spent outside of the critical section to tune contention, and raised and lowered it as the test ran. We ran these tests on a range of core counts, with a fixed number of loops locking and unlocking on each core. We ran the variable tests across various core counts, and tested the ranges of contention and core counts over which our adaptive lock outperformed all of the other lock types. We spread our threads across physical cores on the xeon phi, and for all of our tests we ran with one thread per core.

RESULTS

We ran tests on the first workload on varying core counts to control contention. The higher the core count, the higher the contention. The raw runtimes below show that the ticket lock outperforms the mcs queue lock on low core counts (and therefore low contention), but scales poorly. The mcs queue lock overcomes its higher overhead due to a longer unlocking critical path as the core count and contention increases. Observe that the mcs queue lock outperforms the omp critical construct across all core counts.

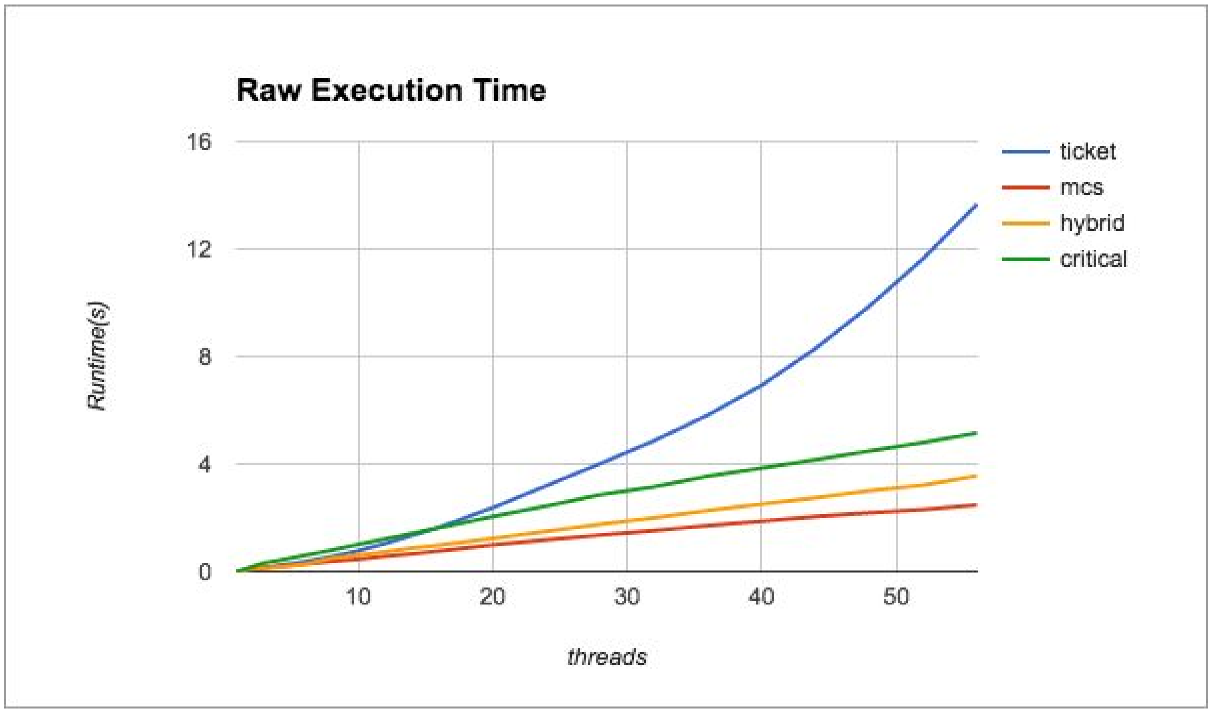

Fig 1. Raw runtime as number of cores (and total work) increases

Figure 2 shows the same results with the runtime normalized to the core count. With this representation, it is clear that the performance both the mcs queue lock and our adaptive lock scales linearly with respect to core count. It can also be seen that under low contention, the adaptive lock seems to follow the curve for the ticket lock, thus it manages to capture the behavior of both locks.

Figure 3 shows performance of our locks when compared to OMP Critical. With the exception of the ticket lock under high core counts and contention, all of the locks we implemented outperform OMP Critical. Under low contention and low core counts, the ticket lock is many times faster than OMP Critical, and under high contention with high core counts, the mcs queue lock is approximately twice as fast. Our hybrid lock maintains its superiority to OMP Critical throughout all core counts, but with additional overhead when compared to either the pure ticket or queue lock it falls short of the ticket lock under low contention, and falls short of the queue lock under high contention.

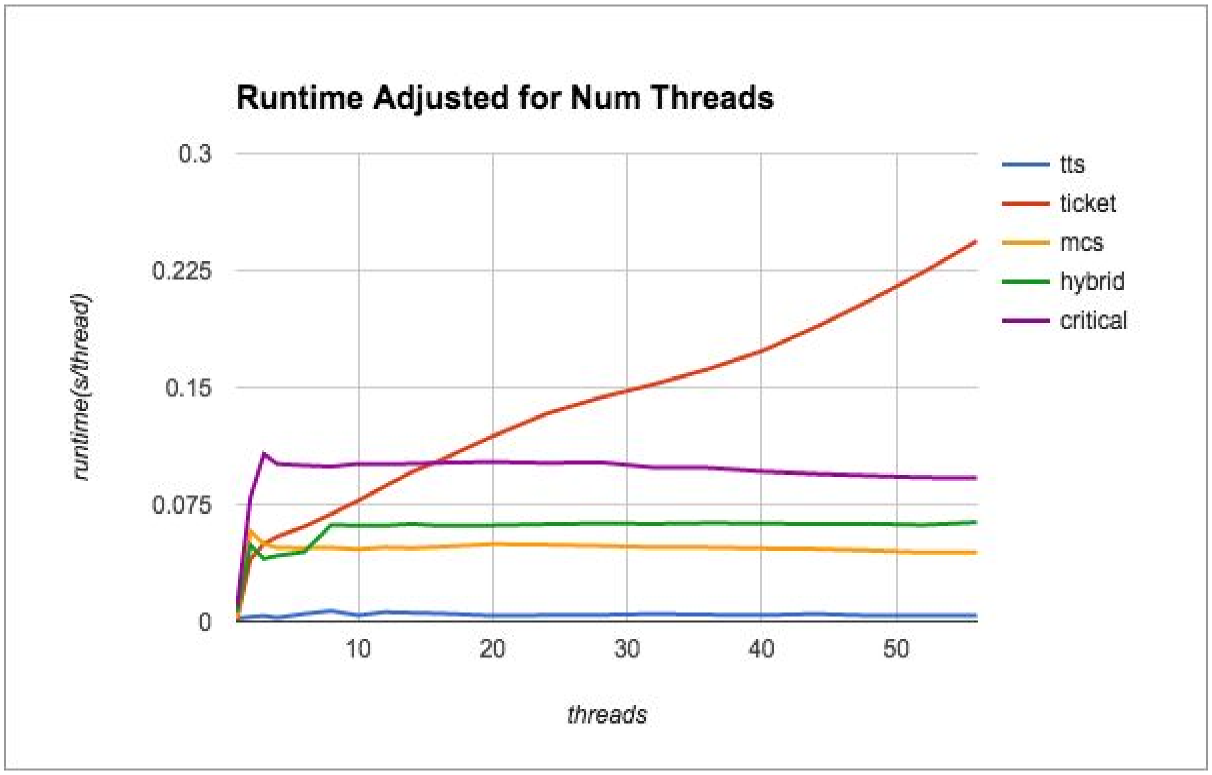

Fig 2. Runtime normalized to core count to demonstrate scaling behavior

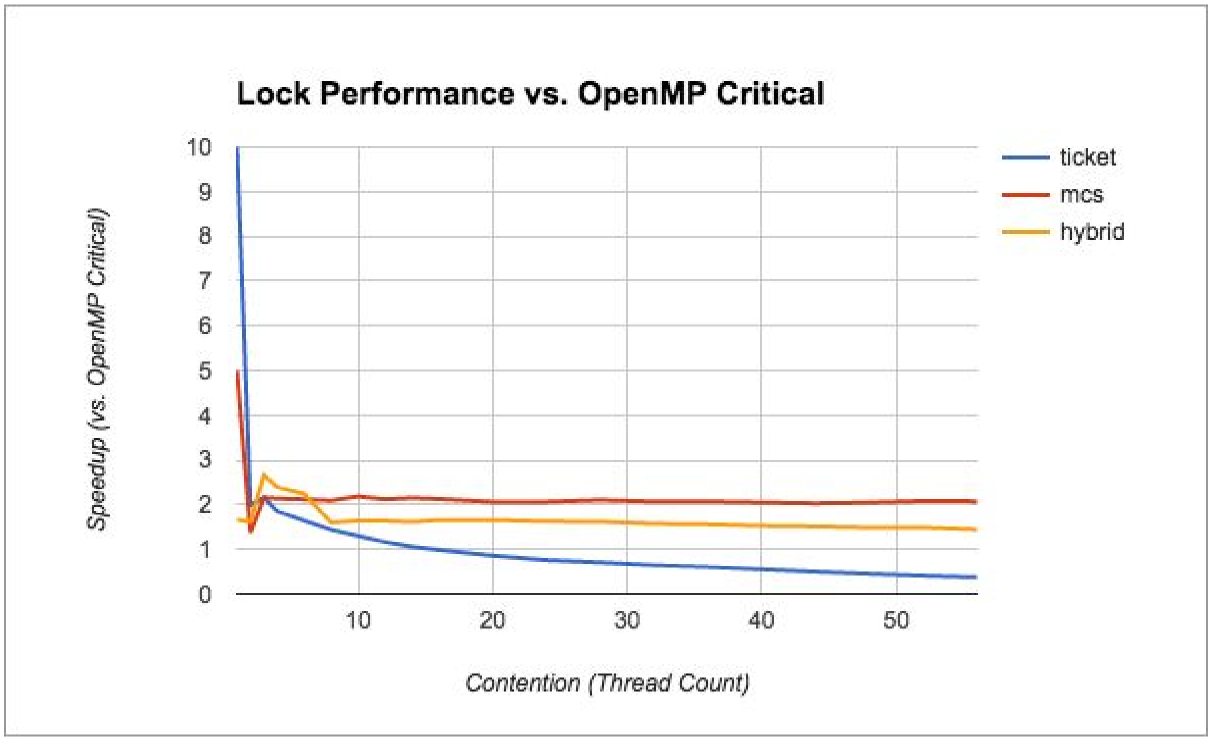

Fig 3. Performance relative to OMP Critical (note the improvement our hybrid lock brings over OMP Critical over all core counts)

We measured fairness by counting the number of times the shared lock is acquired before the last locker grabs the lock for the first time. This is a good estimation of how badly each lock starves lockers during a typical run of our first workload. Fairness is important because unfair locks can starve a thread and hinder overall progress of a program. Many workloads perform better with fair locks, because starvation of a particular thread can prevent proper parallel operation. The test and test and set lock is, as expected, extremely unfair. We observed that the OMP Critical construct is surprising unfair as well, whereas the ticket, mcs queue, and hybrid locks all serve lockers in FIFO order, maintaining perfectly fair operation. The differences we observed are likely an artifact of the different lock and unlock critical path lengths for each lock.

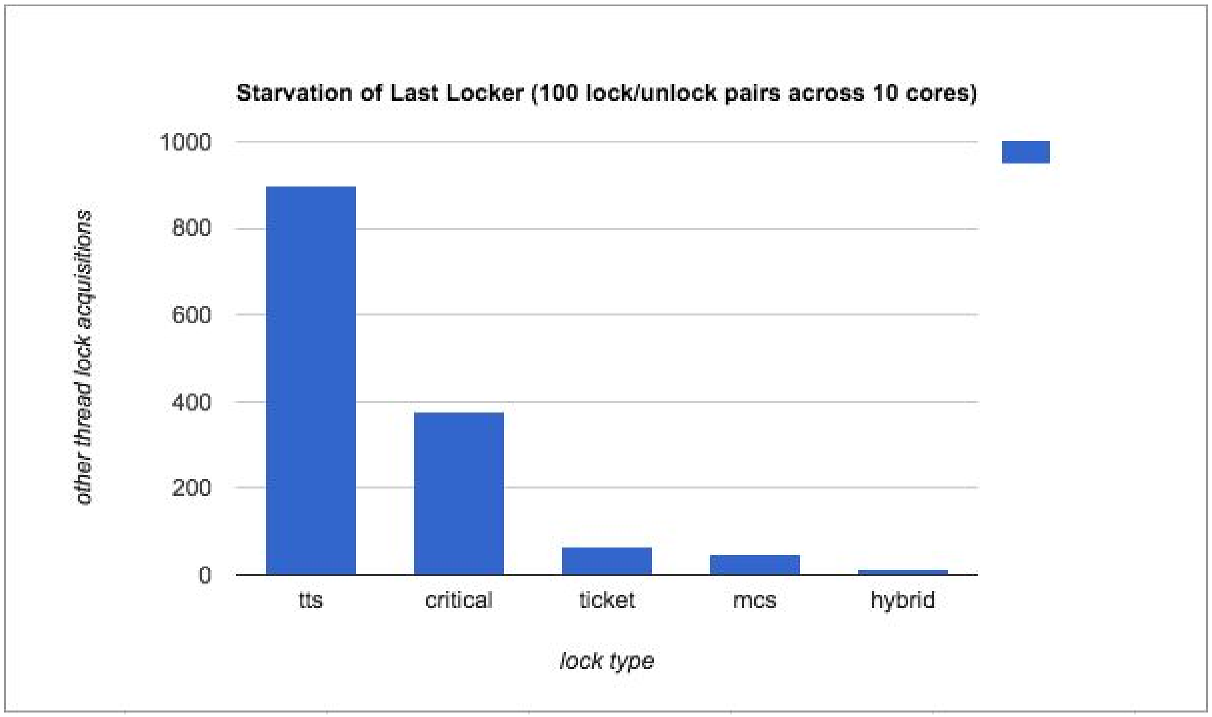

Fig 4. Approximate starvation caused by each lock

Our final microbenchmark was to test our locks under varying workloads. This approximates the varying workloads similar to those that can be found in most real world programs. It also makes a case for using our hybrid lock. As shown in figure 5, with 4-5 threads and a delay of 800 to 1500 cycles outside of the loop, and 10 cycles within the loop, the hybrid lock is king. When we consider the broader range of thread counts seen in figure 6 or different workloads, however, we see that the mcs queue lock almost always performs better for high contention.

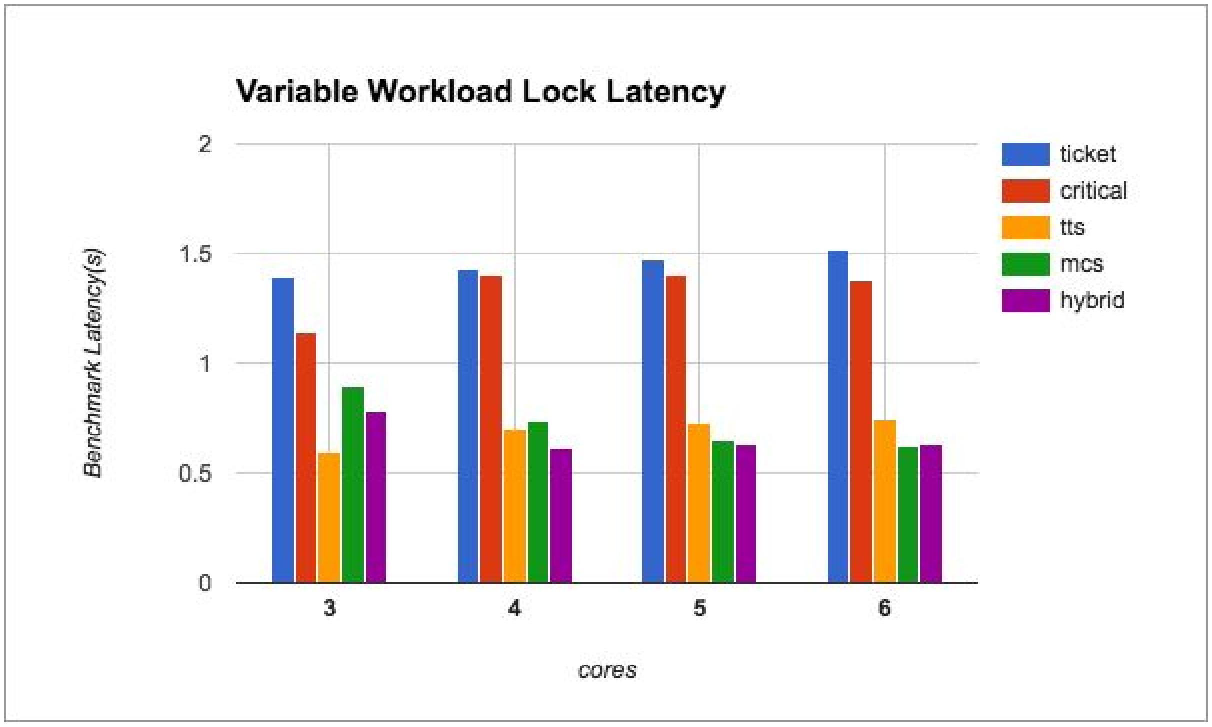

Fig 5. Core range where our hybrid lock outperforms all other locking types on a variable workload

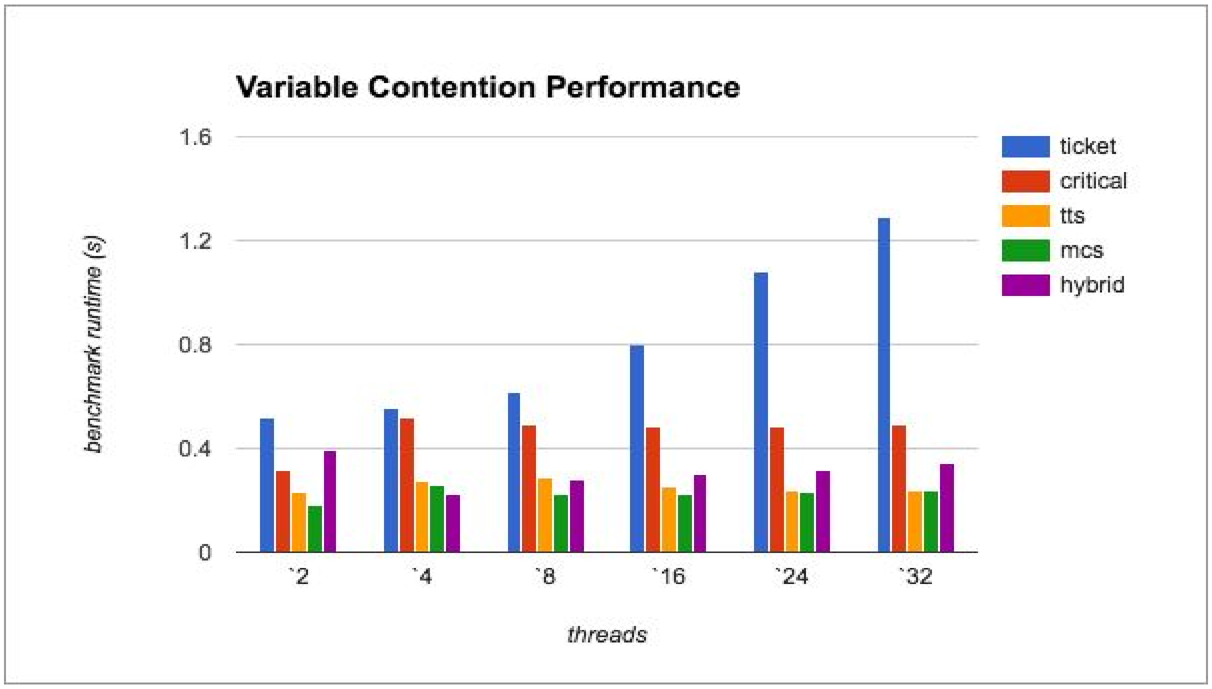

Fig 6. Broader range of core counts, demonstrating that mcs queue locks perform better under most circumstances

CONCLUSION

We determined that under most circumstances where scalability is a concern, the mcs queue lock is the best choice. When contention and core count is low, ticket locks perform better than mcs queue locks. If fairness is not a concern, we found that the test and test and set lock with exponential backoff can be the best choice due to its superior throughput. Our hybrid lock performs better than other types of locks under certain workloads with moderate and variable contention.

WORK DONE BY EACH STUDENT

Equal work was done by both students.

REFERENCES

[1] John M. Mellor-Crummey and Michael L. Scott; et al. (February 1991). "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". ACM TOCS.

[2] Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." Proceedings of the Linux Symposium. 2012. http://pdos.csail.mit.edu/papers/linux:lock.pdf

[3] Thomas Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems, vol. 1, num. 1,* January 1990, pages 6 - 16.

[4] Linus Torvalds (2016). Linux (4.x) [Operating system]. Retrieved from https://github.com/torvalds/linux

[5] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.