Non-scalable locks are dangerous

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Abstract

Several operating systems rely on non-scalable spin locks for serialization. For example, the Linux kernel uses ticket spin locks, even though scalable locks have better theoretical properties. Using Linux on a 48-core machine, this paper shows that non-scalable locks can cause dramatic collapse in the performance of real workloads, even for very short critical sections. The nature and sudden onset of collapse are explained with a new Markov-based performance model. Replacing the offending non-scalable spin locks with scalable spin locks avoids the collapse and requires modest changes to source code.

1 Introduction

It is well known that non-scalable locks, such as simple spin locks, have poor performance when highly contended [1, 7, 9]. It is also the case that many systems nevertheless use non-scalable locks. However, we have run into multiple situations in which system throughput collapses suddenly due to non-scalable locks: for example, a system that performs well with 25 cores completely collapses with 30. Equally surprising, the offending critical sections are often tiny. This paper argues that non-scalable locks are dangerous. For concreteness, it focuses on locks in the Linux kernel.

One piece of the argument is that non-scalable locks can seriously degrade overall performance, and that the situations in which they may do so are likely to occur in real systems. We exhibit a number of situations in which the performance of plausible activities collapses dramatically with more than a few cores' worth of concurrency; the cause is a rapid growth in locking cost as the number of contending cores grows.

Another piece of the argument is that the onset of performance collapse can be sudden as cores are added. A system may have good measured performance with N cores, but far lower total performance with just a few more cores. The paper presents a predictive model of non-scalable lock performance that explains this phenomenon.

A third element of the argument is that critical sections which appear to the eye to be very short, perhaps only a few instructions, can nevertheless trigger performance collapses. The paper's model explains this phenomenon as well.

Naturally we argue that one should use scalable locks [1, 7, 9], particularly in operating system kernels where the workloads and levels of contention are hard to control. As a demonstration, we replaced Linux's spin locks with scalable MCS locks [9] and re-ran the software that caused performance collapse. For 3 of the 4 benchmarks the changes in the kernel were simple. For the 4th case the changes were more involved because the directory cache uses a complicated locking plan and the directory cache in general is complicated. The MCS lock improves scalability dramatically, because it avoids the performance collapse, as expected. We experimented with other scalable locks, including hierarchical ones [8], and observe that the improvements are negligible or small—the big win is going from non-scalable locks to scalable locks.

An objection to this approach is that non-scalable behavior should be fixed by modifying software to eliminate serialization bottlenecks, and that scalable locks merely defer the need for such modification. That observation is correct. However, in practice it is not possible to eliminate all potential points of contention in the kernel all at once. Even if a kernel is very scalable at some point in time, the same kernel is likely to have scaling bottlenecks on subsequent generations of hardware. One way to view scalable locks is as a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel.

The main contribution of this paper is amplifying the conclusion from previous work that non-scalable locks have risks: not only do they have poor performance, but

they can cause collapse of overall system performance. More specifically, this paper makes three contributions. First, we demonstrate that the poor performance of non-scalable locks can cause performance collapse for real workloads, even if the spin lock is protecting a very short critical section in the kernel. Second, we propose a single comprehensive model for the behavior of non-scalable spin locks that fully captures all regimes of operation, unlike previous models [6]. Third, we confirm on modern *x*86-based multicore processors that MCS locks can improve maximum scalability without decreasing performance, and conclude that the scaling and performance benefits of the different types of scalable locks is small.

The rest of the paper is organized as follows. Section 2 demonstrates that non-scalable locks can cause performance collapse for real workloads. Section 3 introduces a Markov-based model that explains why non-scalable locks can cause this collapse to happen, even for short critical sections. Section 4 evaluates several scalable locks on modern *x*86-based multicore processors to decide which scalable lock to use to replace the offending non-scalable locks. Section 5 reports on the results of using MCS locks to replace the non-scalable locks in Linux that caused performance collapse. Section 6 relates our findings and modeling to previous work. Section 7 summarizes our conclusions.

2 Are non-scalable locks a problem?

This section demonstrates that non-scalable spin locks cause performance collapse for some kernel-intensive workloads. We present performance results from four benchmarks demonstrating that critical sections that consume less than 1% of the CPU cycles on one core can cause performance to collapse on a 48-core *x*86 machine.

2.1 How non-scalable locks work

For concreteness we discuss the ticket lock used in the Linux kernel, but any type of non-scalable lock will exhibit the problems shown in this section. Figure 1 presents simplified C code from Linux. The ticket lock is the default lock since kernel version 2.6.25 (released in April 2008).

An acquiring core obtains a ticket and spins until its turn is up. The lock has two fields: the number of the ticket that is holding the lock (`current_ticket`) and the number of the next unused ticket (`next_ticket`). To obtain a ticket number, a core uses an atomic increment instruction on `next_ticket`. The core then spins until its ticket number is current. To release the lock, a core increments `current_ticket`, which causes the lock to be handed to the core that is waiting for the next ticket number.

```
struct spinlock_t {
  int current_ticket;
  int next_ticket;
}

void spin_lock(spinlock_t *lock)
{
  int t =
    atomic_fetch_and_inc(&lock->next_ticket);
  while (t != lock->current_ticket)
    ;   /* spin */
}

void spin_unlock(spinlock_t *lock)
{
  lock->current_ticket++;
}
```

Figure 1: Pseudocode for ticket locks in Linux.

If many cores are waiting for a lock, they will all have the lock variables cached. An unlock will invalidate those cache entries. All of the cores will then read the cache line. In most architectures, the reads are serialized (either by a shared bus or at the cache line's home or directory node), and thus completing them all takes time proportional to the number of cores. The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process. Thus the cost of each lock handoff increases in proportion to the number of waiting cores. Each inter-core operation takes on the order of a hundred cycles, so a single release can take many thousands of cycles if dozens of cores are waiting. Simple test-and-set spin locks incur a similar $O(N)$ cost per release.

2.2 Benchmarks

We exercised spin locks in the Linux kernel with four benchmarks: FOPS, MEMPOP, PFIND, and EXIM. Two are microbenchmarks and two represent application workloads. None of the benchmarks involve disk I/O (the file-system cache is pre-warmed). We ran the benchmarks on a 48-core machine (eight 6-core 2.4 GHz AMD

Opteron chips) running Linux kernel 2.6.39 (released in May 2011).

FOPS creates a single file and starts one process on each core. Each thread repeatedly `open`s and `close`s the file.

MEMPOP creates one process per core. Each process repeatedly `mmap`s 64 kB of memory with the `MAP_POPULATE` flag, then `munmap`s the memory. `MAP_POPULATE` instructs the kernel to allocate pages and populate the process page table immediately, instead of doing so on demand when the process accesses the page.

PFIND searches for a file by executing several instances of the GNU find utility. PFIND takes a directory and filename as input, evenly divides the directories in the first level of input directory into per-core inputs, and executes one instance of find per core, passing in the input directories. Before we execute the PFIND, we create a balanced directory tree so that each instance of find searches the same number of directories.

EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message. We use the version of EXIM from MOSBENCH [3].

2.3 Results

Figure 2 shows the results for all benchmarks. One might expect total throughput to rise in proportion to the number of cores for a while, then level off to a flat line due to some serial section. Throughput does increase with more cores for a while, but instead of leveling off, the throughput *decreases* after some number of cores. The decreases are sudden; good performance with N cores is often followed by dramatically lower performance with one or two more cores.

FOPS. Figure 2(a) shows the total throughput of FOPS as a function of the number of cores concurrently running the benchmark. The performance peaks with two cores. With 48 cores, the total throughput is about 3% of the throughput on one core.

The performance collapse in Figure 2(a) is caused by a per-entry lock in the file system name/inode cache. The kernel acquires the lock when a file is closed in order to decrement a reference count and possibly perform cleanup actions. On average, the code protected by the lock executes in only 92 cycles.

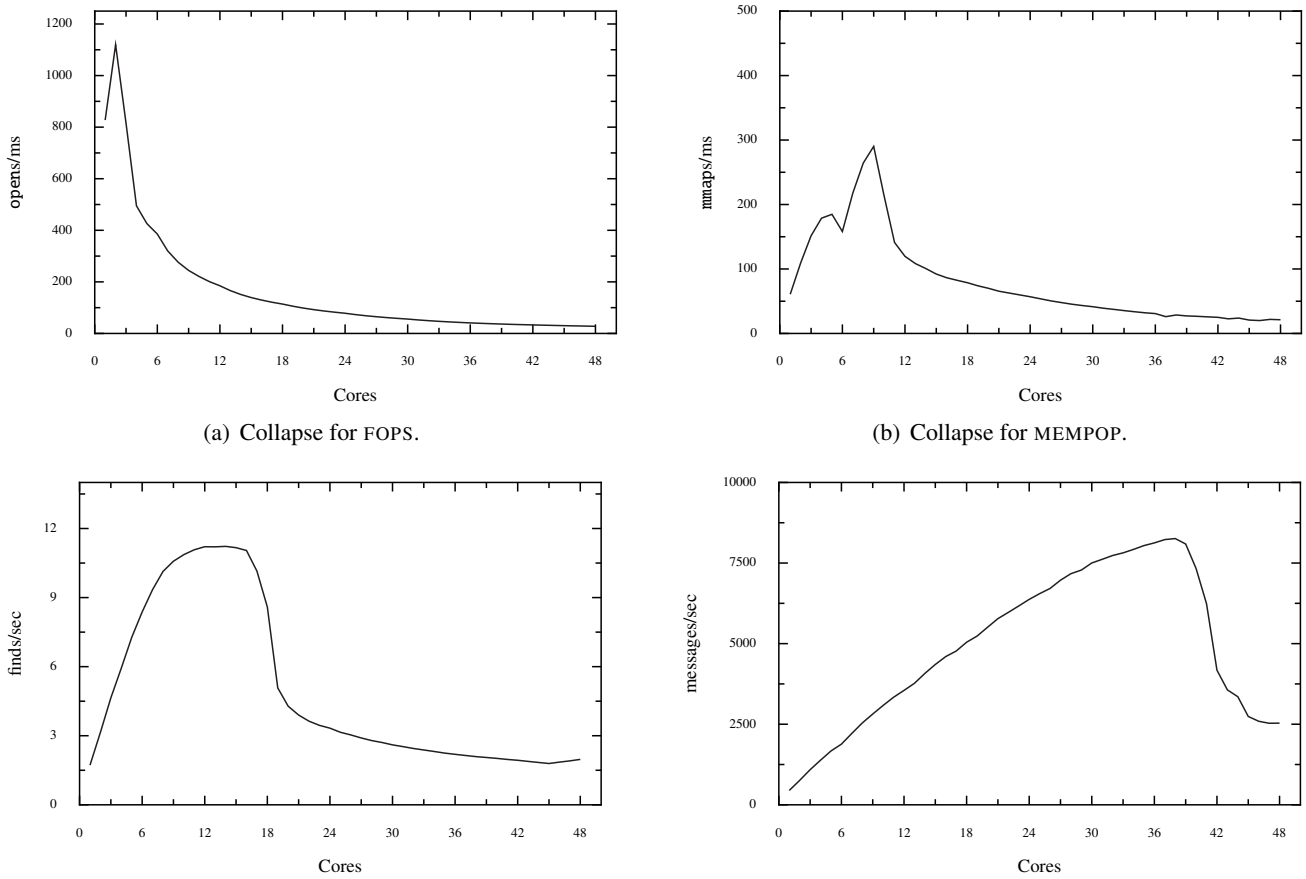
MEMPOP. Figure 2(b) shows the throughput of MEMPOP. Throughput peaks at nine cores, at which point it is 4.7× higher than with one core. Throughput decreases rapidly at first with more than nine cores, then more gradually. At 48 cores throughput is about 35% of the throughput achieved on one core. The performance collapse in Figure 2(b) is caused by a non-scalable lock that protects the data structure mapping physical pages to virtual memory regions.

PFIND. Figure 2(c) shows the throughput of PFIND, measured as the number of find processes that complete every second. The throughput peaks with 14 cores, then declines rapidly. The throughput with 48 cores is approximately equal to the throughput on one core. A non-scalable lock protecting the block buffer cache causes PFIND's performance collapse.

EXIM. Figure 2(d) shows EXIM's performance as a function of the number of cores. The performance collapse is caused by locks that protect the data structure mapping physical pages to virtual memory regions. The 3.0.0 kernel (released in Aug 2011) fixes this collapse by acquiring the locks involved in the bottlenecked operation together, and then running with a larger critical section.

Figure 3 shows measurements related to the most contended lock for each benchmark, taken on one core. The "Operation time" column indicates the total number of cycles required to complete one benchmark operation (opening a file, delivering a message, etc). The "Acquires per operation" column shows how many times the most contended lock was acquired per operation. The "Average critical section time" column shows how long the lock was held each time it was acquired. The "% of operation in critical section" reflects the ratio of total time per operation spent in the critical section to the total time for each operation.

The last column of Figure 3 helps explain the point in each graph at which collapse starts. For example, MEMPOP spends 7% of its time in the bottleneck critical section. Once 14 (i.e., $1.0/0.07$) cores are active, one would expect that critical section's lock to be held by some core at all times, and thus that cores would start to contend for the lock. In fact MEMPOP starts to collapse somewhat before that point, a phenomenon explained in the next section.



(a) Collapse for FOPS.

(b) Collapse for MEMPOP.

(c) Collapse for PFIND.

(d) Collapse for EXIM.

Figure 2: Sudden performance collapse with ticket locks.

Benchmark	Operation time (cycles)	Top lock instance name	Acquires per operation	Average critical section time (cycles)	% of operation in critical section
FOPS	503	`d_entry`	4	92	73%
MEMPOP	6852	`anon_vma`	4	121	7%
PFIND	2099 M	`address_space`	70 K	350	7%
EXIM	1156 K	`anon_vma`	58	165	0.8%

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

As an example of a critical section that causes non-scalability, Figure 4 shows the most contended critical sections for EXIM. They involve adding and deleting an element from a list, and consist of a handful of inlined instructions.

2.4 Questions

The four graphs have common features which raise some questions.

- Why does collapse start as early as it does? One would expect collapse to start when there is a significant chance that many cores need the same lock at the same time. Thus one might expect MEMPOP to start to see decline at around 14 cores (1.0/0.07). But the onset occurs earlier, at nine cores.
- Why does performance ultimately fall so far?
- Why does performance collapse so rapidly? One might expect a gradual decrease with added cores, since each new core should cause each release of the bottleneck lock to take a little more time. Instead, adding just a few more cores causes a sharp drop in total throughput. This is worrisome; it suggests that a system that has been tested to perform well with N cores might perform far worse with $N+2$ cores.

3 Model

This section presents a performance model for non-scalable locks that answers the questions raised in the previous section. It first describes the hardware cache coherence protocol at a high level, which is representative of a typical *x*86 system, and then builds on the basic properties of this protocol to construct a model for understanding performance of ticket spin locks. The model closely predicts the observed collapse behavior.

3.1 Hardware cache coherence

Our model assumes a directory-based cache coherence protocol. All directories are directly connected by an inter-directory network. The cache coherence protocol is a simplification of, but similar to, the implementation in AMD Opteron [4] and Intel Xeon CPUs.

```
static void
anon_vma_chain_link(
   struct anon_vma_chain *avc,
   struct anon_vma *anon_vma)
{
  spin_lock(&anon_vma->lock);
  list_add_tail(&avc->same_anon_vma,
                &anon_vma->head);
  spin_unlock(&anon_vma->lock);
}

static void
anon_vma_unlink(
   struct anon_vma_chain *avc,
   struct anon_vma *anon_vma)
{
  spin_lock(&anon_vma->lock);
  list_del(&avc->same_anon_vma);
  spin_unlock(&anon_vma->lock);
}
```

Figure 4: The most contended critical sections from EXIM. This compiler inlines the code for the list manipulations, each of which are less than 10 instructions.

3.1.1 The directory

Each core has a cache directory. The hardware (e.g., the BIOS) assigns evenly sized regions of DRAM to each directory. Each directory maintains an entry for each cache line in its local DRAM:

```
[ tag | state | core ID ]
```

The possible states are:

1. invalid (I) – the cache line is not cached;
2. shared (S) – the cache line is held in one or more caches and matches DRAM;
3. modified (M) – the cache line is held in one cache and does not match DRAM.

For modified cache lines the directory records the cache that holds the dirty cache line.

Figure 5 presents the directory state transitions for loads and stores. For example, when a core issues a load request for a cache line in the invalid state, the directory sets the cache line state to shared.

	I	S	M
Load	S	S	S
Store	M	M	M

Figure 5: Directory transitions for loads and stores.

	I	S	M
Load	–	–	DP
Store	–	BI	DI

Figure 6: Probe messages for loads and stores.

3.1.2 Network messages

When a core begins accessing an uncached cache line, it will send a load or store request to the cache line's home directory. Depending on the type of request and the state of the cache line in the home directory, the home directory may need to send probe messages to all directories that hold the cache line.

Figure 6 shows the probe messages a directory sends based on request type and state of the cache line. "BI" stands for broadcast invalidate. "DP" stands for direct probe. "DI" stands for direct invalidate. For example, when a source cache issues a load request for a modified cache line, the home directory sends a directed probe to the cache holding the modified cache line. That cache responds to the source cache with the contents of the modified cache line.

3.2 Performance model for ticket locks

To understand the collapse observed in ticket-based spin locks, we construct a model. One of the challenging aspects of constructing an accurate model of spin lock behavior is that there are two regimes of operation: when not contended, the spin lock can be acquired quickly, but when many cores try to acquire the lock at the same time, the time taken to transfer lock ownership increases significantly. Moreover, the exact point at which the behavior of the lock changes is dependent on the lock usage pattern, and the length of the critical section, among other parameters. Recent work [6] attempts to model this behavior by combining two models—one for contention and one for uncontended locks—into a single model, by simply taking the max of the two models' predictions. However, this fails to precisely model the point of collapse, and does not explain the phenomenon causing the collapse.

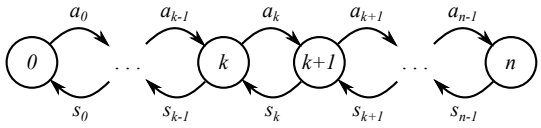


Figure 7: Markov model for a ticket spin lock for n cores. State i represents i cores holding or waiting for the lock. a_i is the arrival rate of new cores when there are already i cores contending for the lock. s_i is the service rate when $i+1$ cores are contending.

To build a precise model of ticket lock behavior, we build on queueing theory to model the ticket lock as a Markov chain. Different states in the chain represent different numbers of cores queued up waiting for a lock, as shown in Figure 7. There are $n+1$ states in our model, representing the fact that our system has a fixed number of cores (n).

Arrival and service rates between different states represent lock acquisition and lock release. These rates are different for each pair of states, modeling the non-scalable performance of the ticket lock, as well as the fact that our system is closed (only a finite number of cores exist). In particular, the arrival rate from k to $k+1$ waiters, a_k, should be proportional to the number of remaining cores that are not already waiting for the lock (i.e., $n-k$). Conversely, the service rate from $k+1$ to k, s_k, should be inversely proportional to k, reflecting the fact that transferring ownership of a ticket lock to the next core takes linear time in the number of waiters.

To compute the arrival rate, we define a to be the average time between consecutive lock acquisitions on a single core. The rate at which a single core will try to acquire the lock, in the absence of contention, is $1/a$. Thus, if k cores are already waiting for the lock, the arrival rate of new contenders is $a_k = (n-k)/a$, since we need not consider any cores that are already waiting for the lock.

To compute the service rate, we define two more parameters: s, the time spent in the serial section, and c, the time taken by the home directory to respond to a cache line request. In the cache coherence protocol, the home directory of a cache line responds to each cache line request in turn. Thus, if there are k requests from different cores to fetch the lock's cache line, the time until the winner (pre-determined by ticket numbers) receives the cache line will be on average $c \cdot k/2$. As a result, processing the serial section and transferring the lock to the next

holder when k cores are contending takes $s+ck/2$, and the service rate is $s_k = \frac{1}{s+ck/2}$.

Unfortunately, while this Markov model accurately represents the behavior of a ticket lock, it does not match any of the standard queueing theory that provides a simple formula for the behavior of the queueing model. In particular, the system is closed (unlike most open-system queueing models), and the service times vary with the size of the queue.

To compute a formula, we derive it from first principles. Let $P_0, \ldots, P_n$ be the steady-state probabilities of the lock being in states 0 through n respectively. Steady state means that the transition rates balance: $P_k \cdot a_k = P_{k+1} \cdot s_k$. From this, we derive that $P_k = P_0 \cdot \frac{n!}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)$. Since $\sum_{i=0}^{n} P_i = 1$, we get $P_0 = 1/\left(\sum_{i=0}^{n}\left(\frac{n!}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)\right)$, and thus:

$$P_k = \frac{\frac{1}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)}{\sum_{i=0}^{n}\left(\frac{1}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)} \qquad (1)$$

Given the steady-state probability for each number of cores contending for the lock, we can compute the average number of waiting (idle) cores as the expected value of that distribution, $w = \sum_{i=0}^{n} i \cdot P_i$. The speedup achieved in the presence of this lock and serial section can be computed as $n - w$, since on average that many cores are doing useful work, while w cores are spinning.

3.3 Validating the model

To validate our model, Figures 8 and 9 show the predicted and actual speedup of a microbenchmark with a single lock, which spends a fixed number of cycles inside of a serial section protected by the lock, and a fixed number of cycles outside of that serial section. Figure 8 shows the predicted and actual speedup when the serial section always takes 400 cycles to execute, but the non-serial section varies from 12.5k to 200k cycles. As we can see, the model closely matches the real hardware speedup for all configurations.

In Figure 9, we also present the predicted and actual speedup of the microbenchmark when the serial section is always 2% of the overall execution time (on one core),

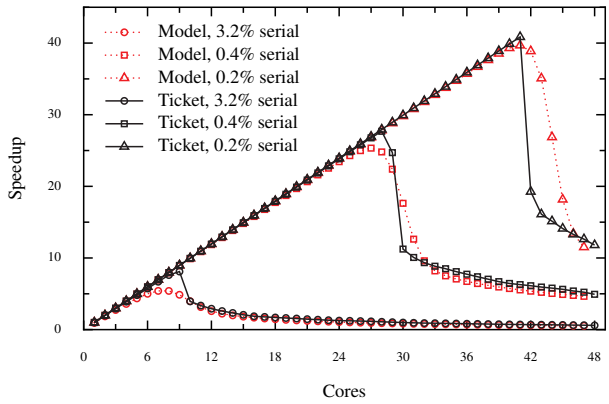


Figure 8: Predicted and actual performance of ticket spin locks with a 400-cycle serial section, for a microbenchmark where the serial section accounts for a range of fractions of the overall execution time on one core.

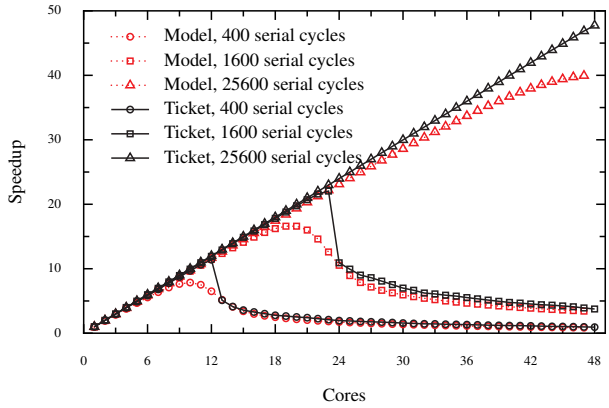


Figure 9: Predicted and actual performance for a microbenchmark where the critical section accounts for 2% of the execution time on one core, but with varying execution time for each invocation of the serial section.

but the time spent in the serial section varies from 400 to 25,600 cycles. Again, the model closely matches the measured speedup. This gives us confidence that our model accurately captures the relevant factors leading to the performance collapse of ticket locks.

One difference between the predicted and measured speedup is that the predicted collapse is slightly more gradual than the collapse observed on real hardware. This is because the ticket lock's performance is unstable near the collapse point, and the model predicts the average steady-state behavior. Our measured speedup reports the throughput for a relatively short-running microbenchmark, which has not had the time to "catch" the instability.

3.4 Implications of model results

The behavior predicted by our model has several important implications. First, the rapid collapse of ticket locks is an inherent property of their design, rather than a performance problem with our experimental hardware. Any cache-coherent system that matches our basic hardware model will experience similarly sharp performance degradation. The reason behind the rapid collapse can be understood by considering the transition rates in the Markov model from Figure 7. If a lock ever accumulates a large number of waiters (e.g., reaches state n in the Markov model), it will take a long time for the lock to go back down to a small number of waiters, because the service rate s_k rapidly decays as k grows, for short serial sections. Thus, once the lock enters a contended state, it becomes much more likely that more waiters arrive than that the current waiters will make progress in shrinking the lock's wait queue.

A more direct way to understand the collapse is that the time taken to transfer the lock from one core to another increases linearly with the number of contending cores. However, this time effectively increases the length of the serial section. Thus, as more cores are contending for the lock, the serial section grows, increasing the probability that yet another core will start contending for this lock.

The second implication is that the collapse of the ticket lock only occurs for short serial sections, as can be seen from Figure 9. This can be understood by considering how the service rate s_i decays for different lengths of the serial section. For a short serial section time s, $s_k = \frac{1}{s+ck/2}$ is strongly influenced by k, but for large s, s_k is largely unaffected by k. Another way to understand this result is that, with fewer acquire and release operations, the ticket lock's performance contributes less to the overall application throughput.

The third implication is that the collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl's law (for short serial sections). In particular, Figure 9 shows that a microbenchmark with a 2% serial section, which may be able to scale to 50 cores under Amdahl's law, is able to attain less than 10× scalability when the serial section is 400 cycles long.

4 Which scalable lock?

The literature has many proposals for scalable locks, which avoid the collapse that ticket locks exhibit. Which one should we use to replace contended ticket locks? This section evaluates several scalable locks on modern *x*86-based multicore processors.

4.1 Scalable locks

A common way of making the ticket lock more scalable is to adjust its implementation to use proportional backoff when the lock is contended. The challenge with this approach is what constant to choose to multiply the ticket number with. From our model we can conclude that choosing the constant well is important only for short critical section, because for large critical sections collapse does not occur. For our experiments below, we choose the best value by trying a range of values, and selecting the one that gives the best performance. This choice provides the best result that the proportional lock could achieve.

Another approach is to replace the ticket lock with a truly scalable lock. A scalable lock is one that generates a constant number of cache misses per acquisition and therefore avoids the collapse that non-scalable locks exhibit. All of these locks maintain a queue of waiters and each waiter spins on its own queue entry. The differences in these locks are how the queue is maintained and the changed necessary to the `lock` and `unlock` API.

MCS lock. The MCS lock [9] maintains an explicit queue of `qnode` structures. A core acquiring the lock adds itself with an atomic instruction to the end of the list of waiters by having the lock point to its `qnode`, and then sets the next pointer of the `qnode` of its predecessor to point to its `qnode`. If the core is not at the head of the queue, then it spins on its `qnode`. To avoid dynamically allocating memory on each `lock` invocation, the `qnode` is an argument to `lock` and `unlock`.

K42 lock. A potential downside of the MCS lock is that it involves an API change. The K42 lock [2] is a variant of the MCS lock that requires fewer API changes.

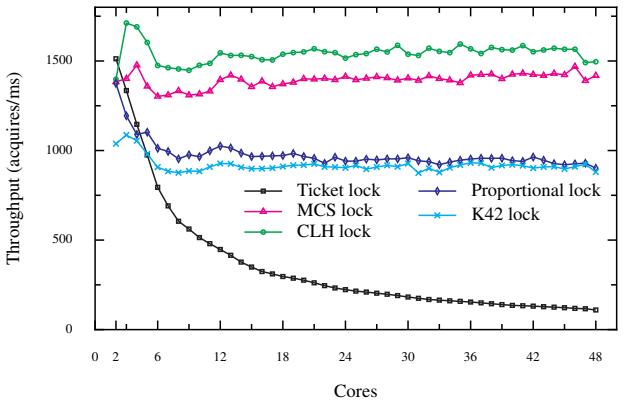


Figure 10: Throughput for cores acquiring and releasing a shared lock. Results start with two cores.

CLH lock. The CLH lock [5] is a variant of an MCS lock where the waiter spins on its predecessor `qnode`, which allows the queue of waiters to be implicit (i.e., the `qnode` next pointer and its updates are unnecessary).

HCLH lock. The HCLH lock [8] is a hierarchical variant of the CLH lock, intended for NUMA machines. The way we use it is to favor lock acquisitions from cores that share an L3 cache with the core that currently holds the lock, with the goal to reduce the cost of cache line transfers between remote cores.

4.2 Results

Figure 10 shows the performance of the ticket lock, proportional lock, MCS lock, K42 lock, and CLH lock on our 48-core AMD machine. The benchmark uses one shared lock. Each core loops, acquires the shared lock, updates 4 shared cache lines, and releases the lock. The time to update the 4 shared cache lines is similar between runs using different locks, and increases gradually from about 800 cycles on 2 cores to 1000 cycles in 48 cores. On our *x*86 multicore machine, the HCLH lock improves performance of the CLH lock by only 2%, and is not shown.

All scalable locks scale better than ticket lock on this benchmark because they avoid collapse. Using the CLH lock results in slightly higher throughput over the MCS lock, but not by much. The K42 lock achieves lower throughput than the MCS lock because it incurs an additional cache miss on acquire. These results indicate that for our *x*86 multicore machine, it does not matter much which scalable lock to choose.

Lock type	Single acquire	Single release	Shared acquire
MCS lock	25.6	27.4	53
CLH lock	28.8	3.9	517
Ticket lock	21.1	2.4	30
Proportional lock	22.0	2.8	30.2
K42 lock	47.0	23.8	74.9

Figure 11: Performance of acquiring and releasing an MCS lock, a CLH lock, and a ticket lock. Single acquire and release are measurements for one core. Shared acquire is the time for a core to acquire a lock recently released by another core. Numbers are in cycles.

We also ran the benchmarks on a multicore machine with Intel CPUs and measured performance trends similar to those shown in Figure 10.

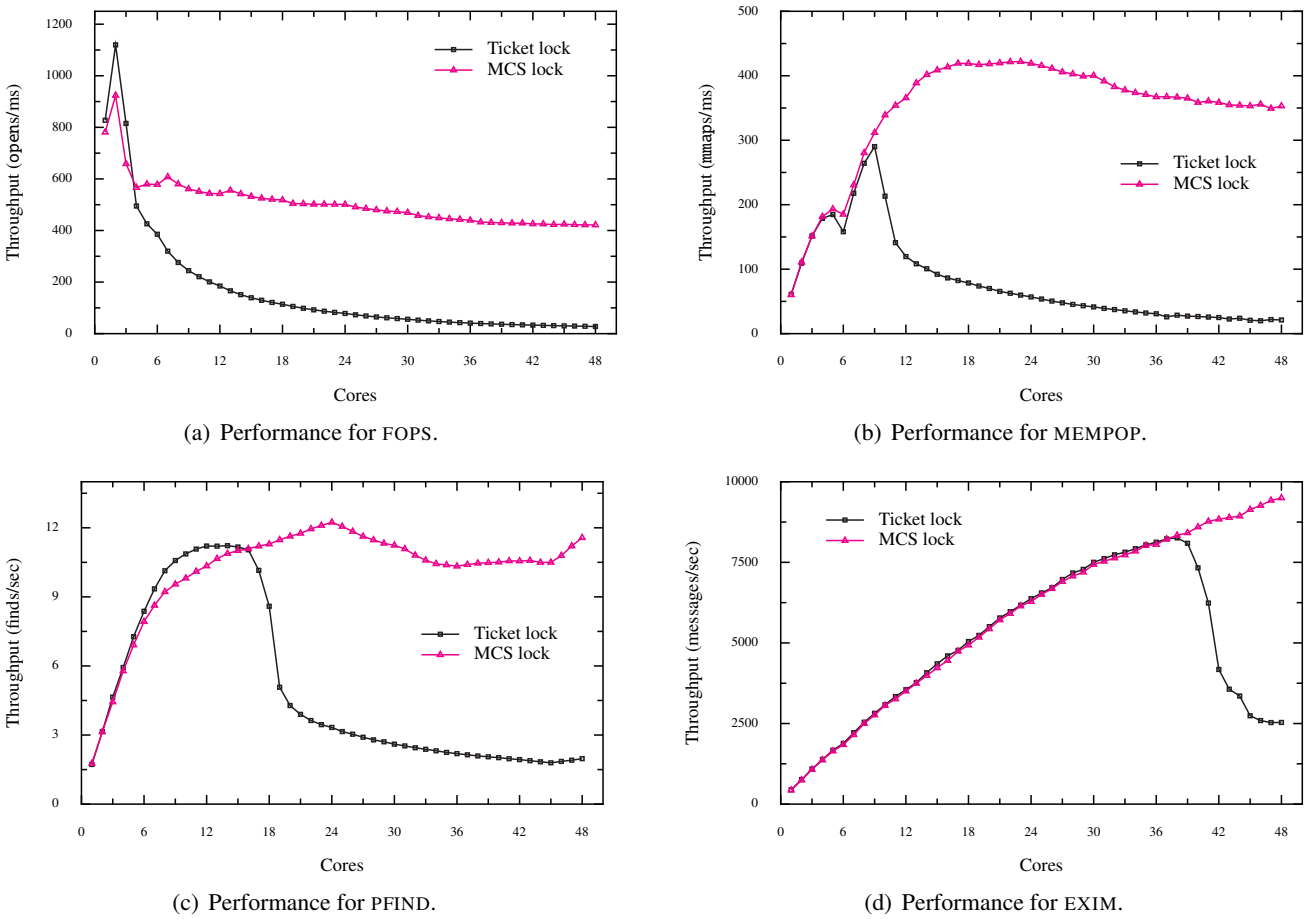
Another concern about different locks is the cost of `lock` and `unlock`. Figure 11 shows the cost for each lock in the uncontended and contended case. All locks are relatively inexpensive to acquire on a single core with no sharing. MCS lock and K42 lock are more expensive to release on a single core, because, unlike the other locks, they use atomic instructions to release the lock. Acquiring a shared but uncontended lock is under 100 cycles for all locks, except the CLH lock. Acquiring the CLH lock is expensive due to the overhead introduced by the `qnode` recycling scheme for multiple cores.

5 Using MCS locks in Linux

Based on the result of the previous section, we replaced the offending ticket locks with MCS locks. We first describe the kernel changes to use MCS locks, and then measure the resulting scalability for the 4 benchmarks from Section 2.

5.1 Using MCS Locks

We replaced the three ticket spin locks that limited benchmark performance with MCS locks. We modified about 1,000 lines of the Linux kernel (700 lines for `d_entry`, 150 lines for `anon_vma`, and 150 lines for `address_space`).



(a) Performance for FOPS.

(b) Performance for MEMPOP.

(c) Performance for PFIND.

(d) Performance for EXIM.

Figure 12: Performance of benchmarks using ticket locks and MCS locks.

As noted earlier, MCS locks have a different API than the Linux ticket spin lock implementation. When acquiring an MCS lock, a core must pass a `qnode` variable into `mcs_lock`, and when releasing that lock the core must pass the same `qnode` variable to `mcs_unlock`. For each lock a core holds, the core must use a unique `qnode`, but it is acceptable to use the same `qnode` for locks held at different times.

Many of our kernel modifications are straightforward. We allocate an MCS `qnode` on the stack, replace `spin_lock` and `spin_unlock` with `mcs_lock` and `mcs_unlock`, and pass the `qnode` to the MCS acquire and release functions.

In some cases, the Linux kernel acquires a lock in one function and releases it in another. For this situation, we stack-allocate a `qnode` on the call frame that is an ancestor of both the call frame that calls `mcs_lock` and the one that calls `mcs_release`. This pattern is common in the directory cache, and partially explains why we made so many modifications for the `d_entry` lock.

Another pattern, which we encountered only in the directory cache code that implements moving directory entries, is changing the value of lock variables. When the kernel moves a `d_entry` between two directories, it acquires the lock of the `d_entry->d_parent` (which is also a `d_entry`) and the target directory `d_entry`, and then sets the value `d_entry->d_parent` to be the target `d_entry`. With MCS, we must make sure to unlock `d_entry->d_parent` with the `qnode` originally used to lock the target `d_entry`, instead the `qnode` original used to lock `d_entry->d_parent`.

5.2 Results

The graphs in Figure 12 show the benchmark results from replacing contended ticket locks with MCS locks. For a

large number of cores, using MCS improves performance by at least 3.5× and in one case by more than 16×.

Figure 12(a) shows the performance of FOPS with MCS locks. Going from one to two cores, performance with both ticket locks and MCS locks increases. For more than two cores, performance with the ticket spin lock decreases continuously. Performance with MCS locks initially also decreases from two to four cores, then remains relatively stable. The reason for this decrease in performance is that the time spent executing the critical section increases from 450 cycles on two cores to 852 cycles on four cores. The critical section is executed multiple times per-operation and modifies shared data, which incurs costly cache misses. As more cores are added, it is less likely that a core will have been the last core to execute the critical section, and therefore it will incur more cache misses, which will increase the length of the critical section.

Figure 12(b) shows the performance of MEMPOP with MCS locks. Performance with MCS and ticket locks increases up to 9 cores, at which point the performance with ticket locks collapses and continuously decreases as more cores are used. The length of the critical section is relatively short. It increases from 141 cycles on one core to about 350 cycles on 10 cores. MCS avoids the dramatic collapse from the short critical section and increases maximum performance by 16.6×.

Figure 12(c) shows the performance of PFIND with MCS. The `address_space` ticket spin lock performs well up to 15 cores, then causes a performance drop that continues until 48 cores. The serial section updates some shared data which increases the time spent in the critical section from 350 cycles on one core to about 1100 cycles on more than 44 cores. MCS provides a small increase in maximum performance, but not as much as with MEMPOP since the critical section is much longer.

Figure 12(d) shows the performance of EXIM with MCS. Ticket spin locks perform well up to 39 cores, then cause a performance collapse. The critical section is relatively short (165 cycles on one core), so MCS improves maximum performance by 3.7× on 48 cores.

The results show that using scalable locks is not that much work (e.g., less work than using RCU in the kernel) and avoids the performance collapse that results from non-scalable spin locks. The latter benefit is important because institutions often use the same kernels for several years, even as they upgrade to hardware with more cores and the likelihood of performance cliffs increases.

6 Related Work

The literature on scalable and non-scalable locks is vast, many practitioners are well familiar with the issues, and it is well known that non-scalable locks behave poorly under contention. The main contribution that this paper adds is the observation that non-scalable locks can cause system performance to collapse, as well as a model that nails down why the performance collapse is so dramatic, even for short critical sections.

Anderson [1] observes that the behavior of spin locks can be "degenerative". The MCS paper shows that acquisition of a test-and-set lock increases linearly with processors on the BBN Butterfly and that on the Symmetry this cost manifests itself even with a small number of processors [9]. Many researchers and practitioners have written microbenchmarks that show that non-scalable spin locks exhibit poor performance under contention on modern hardware. This paper shows that non-scalable locks can cause the collapse of system performance under plausible workloads; that is, the locking costs for short critical sections can be very large on the scale of kernel execution time.

The Linux scaling study reports on the performance collapse that ticket locks introduce on the EXIM benchmark, but doesn't explain the collapse [3]. This paper shows the collapse and the suddenness with several workloads, and provides a model that explains the acuteness.

Eyerman and Eeckhout [6] provide closed formulas to reason about the speedup of parallel applications that involve critical sections, pointing out that there is regime in which applications achieve better speedup than Amdahl's law predicts. Unfortunately, their model makes a distinction between the contended and uncontended regime and proposes formula for each regime. In addition, the formulas do not model the insides of locks; instead, they assume scalable locks. This paper contributes a comprehensive model that accurately predicts performance across the whole spectrum from uncontended to contended, and applies it to modeling the inside of locks.

7 Conclusion

This paper raises another warning about non-scalable locks. Although it is well understood that non-scalable

spin locks have poor performance, it is less well appreciated that their poor performance can have dramatic effect on overall system performance. This paper shows that non-scalable locks can cause system performance to collapse under real workloads and that the collapse is sudden, and presents a Markov model that explains the sudden collapse. We conclude that using non-scalable locks is dangerous because a system that has been tested with N cores may experience performance collapse at a few more cores—or, put differently, if one upgrades a machine in hopes of achieving higher performance, one might run the risk of ending up with a performance collapse. Scalable locking is a stop-gap solution that avoids collapse, but achieving higher performance with additional cores can require new designs using lock-free data structures.

Acknowledgments

We thank Abdul Kabbani for helping us analyze our Markov model. This work was supported by Quanta Computer, by Google, and by NSF awards 0834415 and 0915164.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.

[2] M. A. Auslander, D. J. Edelsohn, O. Y. Krieger, B. S. Rosenburg, and R. W. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 10/128,745, 2003.

[3] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*, Vancouver, Canada, October 2010.

[4] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16–29, March–April 2010.

[5] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, University of Washington, Department of Computer Science and Engineering, 1993.

[6] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 262–370, Saint-Malo, France, June 2010.

[7] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufman, 2008.

[8] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *Proceedings of the European Conference on Parallel Computing*, pages 801–810, Dresden, Germany, August–September 2006.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.