Overview of Kernel locking improvements

Developed by the HP Linux Kernel Performance Team:

1. Waiman Long
2. Jason Low
3. Davidlohr Bueso
4. Scott Norton
5. Aswin Chandramouleeswaran

Why was this effort undertaken?

1. Lock contention is the primary reason for cache contention.

Test Case

a. Threads were bound to specific cores.
b. Memory was allocated from a specific node.
c. The threads did a tight loop with spin_lock/unlock 1,000,000times after they were synchronized.

Observation

1. Contention degraded performance slowly as more cores were involved.

# Cores		**Performance degradation** from the previous test
2 cores	:	0%
3 cores	:	38%
4 cores	:	30%
....14 cores	:	7%

2. There is **a huge drop in performance** when they add a single core from a **second socket.**

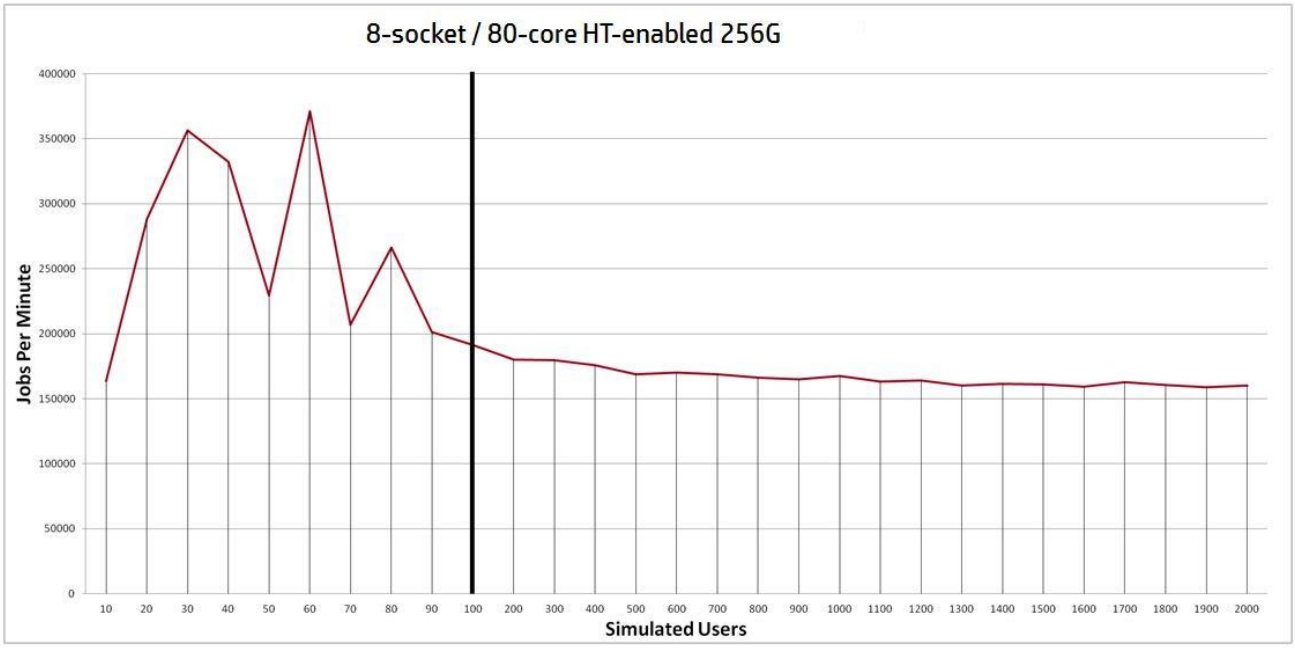
This was due to the need to go through the **QPI layer to handle the cache-to-cache traffic** to handle the cache line contention.

Why is this an important issue?

a. Applications today have been tuned to run on bigger systems. They cannot be expected to scale based on only the number of CPUs if lock contention has problems with scaling as indicated above.

How they started working on this problem?

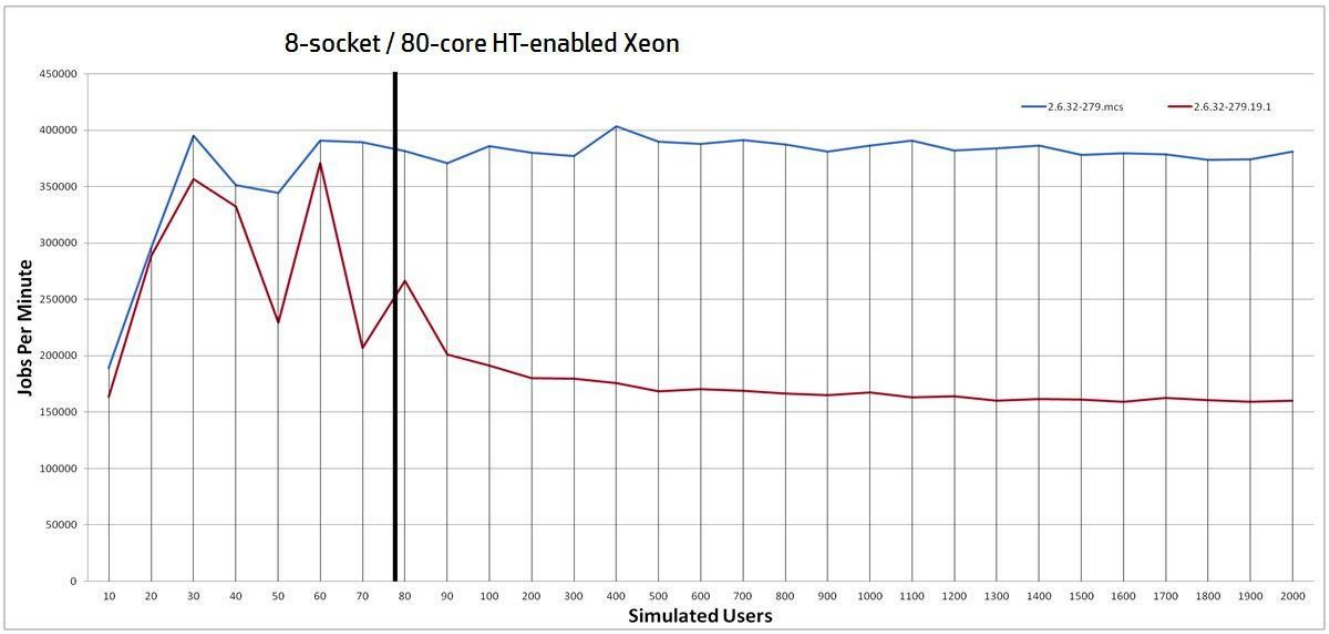
They observed a sharp degradation in the performance of one of their customer workloads AIM7 fserver workload on a 8 socket/ 80 core numa system on a 2.6 kernel.



When they analyzed using *perf -g* output they observed that majority of the time was spent in lock contention. Proportion of the **time spent on contending for lock was 50% more** when higher number of cpus were used. This was in the functions *file_move* and *file_kill()*

Solution : MCS locks

Result:



The blue line in the above graph is the improvement seen after adding MCS locks

The time spent in file_move and file_kill() dropped from 50% to 2% .

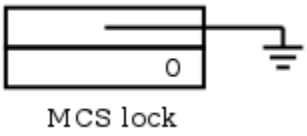
Conclusion was that reducing lock contention is one thing. But attention to lock algorithms that deal with contention **is** extremely important.

Spin-Lock Improvements

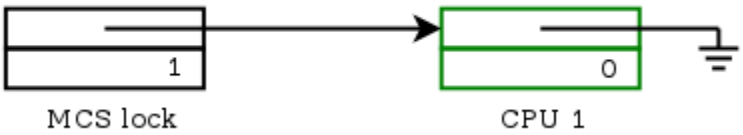
What are MCS locks? [You can find this on lwn.net too]

MCS locks are per-cpu data structures designed to avoid cache contention.

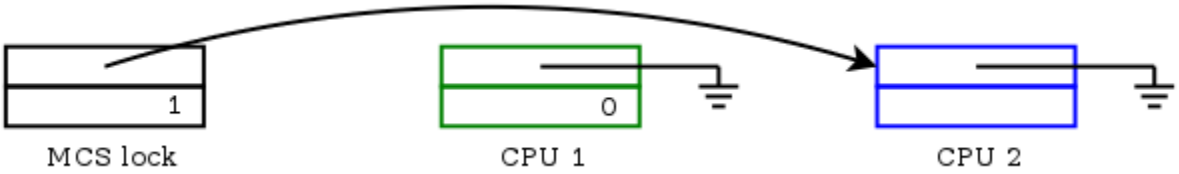
Global Lock : MCS lock



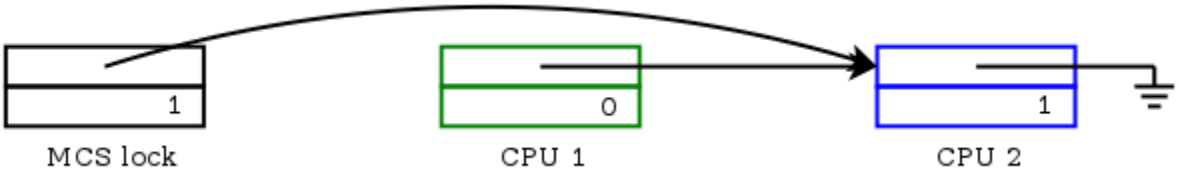
When CPU1 tries to get the lock, it gets it and the *next ptr of MCS lock points to CPU1's per-cpu structure.



When CPU2 tries to get the lock, it sees the *next ptr of MCS lock pointing to CPU1. And spins on its own locked value, with the *next_ptr of MCS lock changing to CPU2



CPU1's *next ptr also now points to CPU2 as next in line



When CPU1 releases the lock, it changes the locked value in CPU2's structure. CPU2 stops spinning and proceeds.

The idea is that each CPU spins on a per-cpu lock with the global lock's *next_ptr always pointing to the head in queue and the per-cpu *next ptr pointing to the next in line. This avoids lock contention because spinning is on a per-cpu structure.

Mutex Lock improvements:

What happens in a mutex lock ?

```
if (atomic_cmpxchg(&lock->count, 1, 0) == 1 )
```

Optimization 1

This will lead to lock contention because the value of the lock is written into.

We can instead do:

```
if(atomic_read(&lock->count) == 1) && (atomic_cmpxchg(&lock->count, 1, 0) == 1))
```

You can avoid writing into the lock count if it is already taken and only attempt to do so if it is un-contended.

Optimization 2

Shorten the critical region. This way the cpus waiting on mutex need not wait too long in a blocked state.

Optimization 3

Optimize wait_list handling

Optimization 4

Add MCS lock inside a mutex.

These changes brought about a 248% improvement in performance over baseline.

Read write semaphore Improvements

Read write semaphores suffer from two fundamental issues:

1. Writer starvation
2. Lack of optimistic spinning on lock acquisition

1.

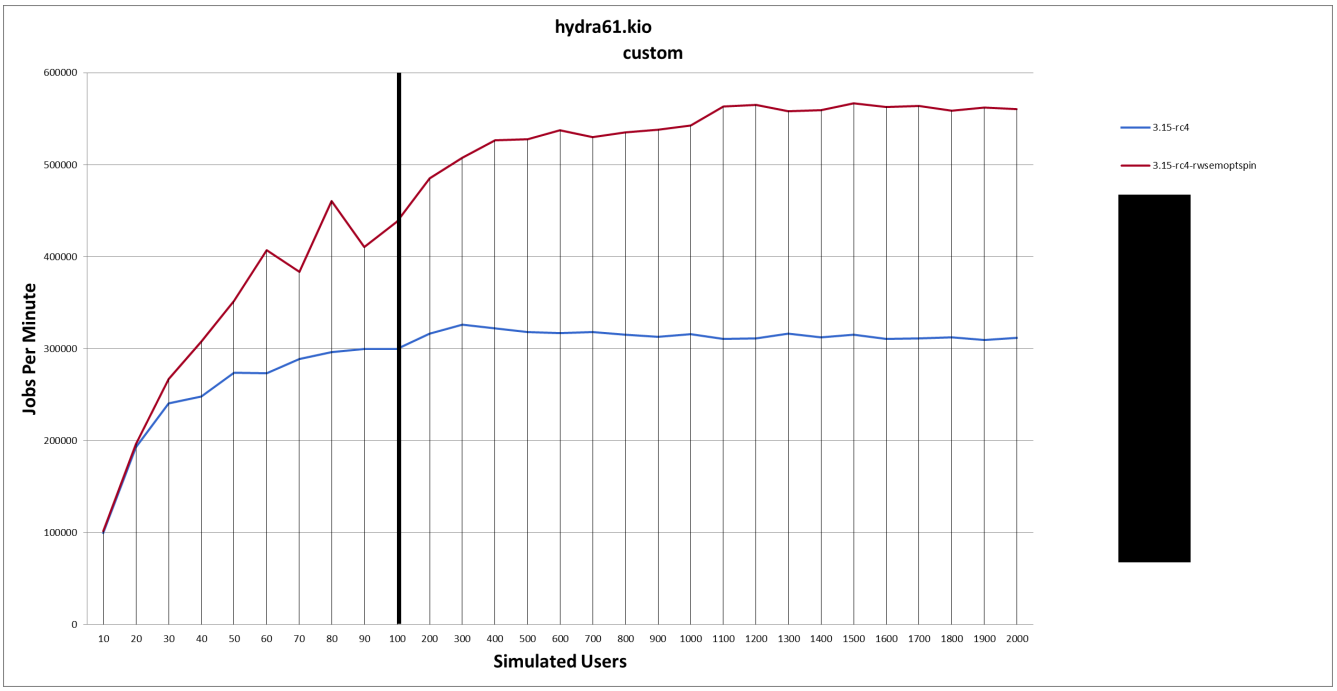
a. When a writer semaphore is free, the next writer contending on the lock has to be woken up. But he is blocked and is yet to wakeup and take the semaphore. **Till then all other writers are blocked**.
b. This is because **r/w semaphores follow a strict FIFO sequential write ownership**. Hence implement **writer lock stealing** which will make another writer steal lock from the one which is yet to wakeup and acquire the lock. **Results in better CPU usage time.**

2.

a. **Instead of blocking** when the semaphore is for a short critical section do an **optimistic spinning.** The overhead of blocking is high. Only if the current lock holder is not running or needs to be rescheduled, then block wait. **This is what mutexes have been doing**. This idea saw roots in the real time kernel.

The above optimizations however **bloat the rw_semaphore structure**. And when there are both readers and writers , the writer can do **excessive spinning** as there is **no owner ship** to optimistically spin on.

This is the major difference between r/w semaphores and mutexes.



The red line in the above graph is the result after adding rw semaphore optimizations.

Read write spinlocks improvements

R/W spinlocks suffered from

a. Fairness issue: Writers could get starved if there are multiple readers or if the lock is found local to the node.
B . The performance was poor

While **ticket spinlock solved the fairness issue** and it also ensured that it did not bloat up the r/w lock data structure, it **did not solve the performance issue.**

Solution for Performance:

MCS lock was proposed for this too. But MCS locks are **too large in size**. They are embedded into kernel data structures that could lead to **cache miss penalties**. So the idea is to use **queued spinlocks**.

Queued spinlocks:

This is a 32 bit data structure which has one bit reserved for the CPU at the head of the queue

1. The CPU at the head of the wait-queue will spin on the global lock. Hence it is the global lock that gets modified when the lock is released and not the per-cpu mcs data structure.
2. The other CPUs in the wait-queue spin on their own per-cpu lock.

This means besides **avoiding cache line contention among waiters since they spin on their per-cpu data structure**, cache miss penalty can be avoided for the first waiter since he modifies the lock bit in the 32 bit value instead of the per-cpu mcs data structure.

This optimization resulted in a performance improvement between 95-110%.

Lockrefs improvements:

1. Reference counts keep track of the number of users of a data structure

2. Reference counts are protected by locks. So there again the issue of lock contention kicks in.

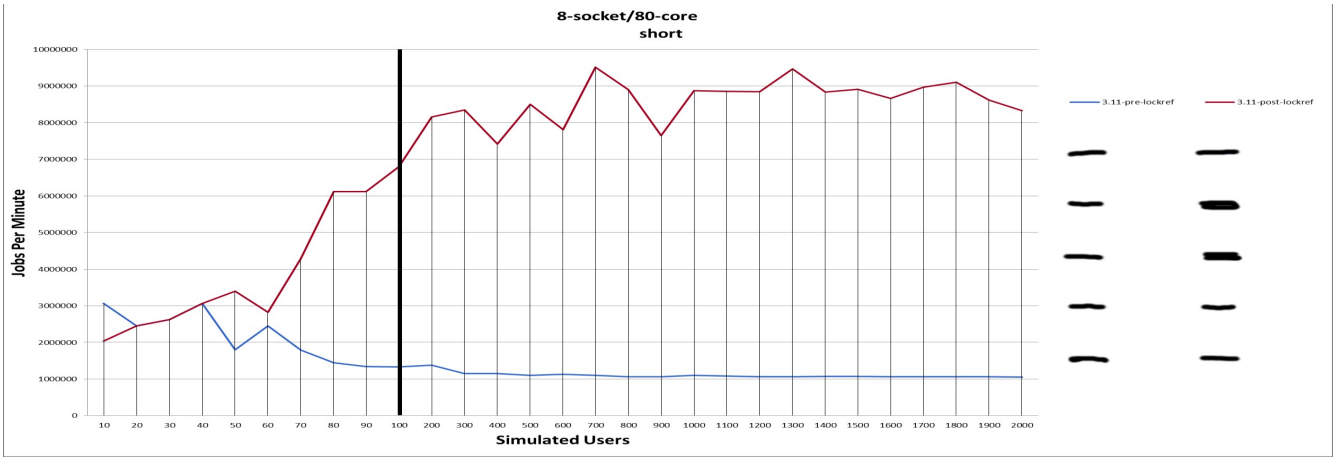
3. Reference counts are usually atomic variables. But they cannot be updated atomically if they are part of a larger data structure.

4. Therefore the optimization is to pack the spinlock and the reference count variable into a 8 byte word which can be atomically updated .

Optimization

So the lock is verified and ref count is updated atomically. If you are the only one to update the ref count you wont even touch the spinlock.

But if there are contenders , you fall back to to the old fashioned way of acquiring the spinlock before updating the refcount.



The red line in the graph shows the improvement after the optimization

The time spent in raw_spin_lock code went down from 80% to 13%