An Overview of Kernel Lock Improvements

Davidlohr Bueso & Scott Norton
LinuxCon North America, Chicago
August 2014

Agenda

1. Why focus on locking?
2. Proof of concept
3. MCS Locks
4. Mutexes
5. Read/write semaphores
6. Read/write [spin]locks
7. Lockref
8. Futexes

Why Focus on Locking?

Cache Line Contention in Large NUMA Systems

- Lock contention (particularly spinning lock contention) is the primary, and probably worst, cause of cache line contention
  - Cache line contention does have a "cost" associated with NUMA systems, but it is not the same "cost" that you experience with local vs. remote memory latency in NUMA systems
- However, it's not only about lock contention
  - Cache line contention can also come from sharing cache lines due to poor data structure layout – two fields in a data structure that are accessed by completely different processes/threads, but end up in the same cache line
    - Worst case: an unrelated and frequently accessed field occupies the same cache line as a heavily contended lock
  - Other atomic operations, such as atomic-add, can also generate cache line contention
  - Additionally, the processor's cache prefetch mechanism may also cause false cache line contention

Demonstrating Cache Line Contention Effects

- Test program to show the cost of cache line contention in large NUMA systems:
  - Bind threads (1 per core) to specified cores. Memory is allocated from a specific node.
  - Once the threads are synchronized, perform a tight loop doing spin_lock/spin_unlock 1,000,000 times. This generates an extreme amount of cache line contention. The spinlock implementation was taken from a Linux 3.0 based kernel.
  - Based on the number of threads and the loop iteration count we can calculate the average number of "operations per second per CPU" when <N> CPUs are involved in the cache line contention.
  - This is not a real-world test. While this is a micro-benchmark, it does show the effects of cache line contention so that real code can be written with cache line contention in mind.
- Test systems:
  - HP DL580 Gen8: 4-socket/ 60-core Intel Xeon E7-4890 v2 2-TB
  - HP CS900: 8-socket/120-core Intel Xeon E7-2890 v2 6-TB
  - HP CS900: 16-socket/240-core Intel Xeon E7-2890 v2 12-TB

Contention within a Socket: Increasing Core Count

Execution Nodes	Memory node	Sockets Used	Cores used	Seconds	Ops per Sec per Core	% decrease from 2-core	% decrease from previous
Node 1	Node 1	1-socket	2-cores	1.704489	5,866,861	0.0%	0.0%
			3-cores	2.783121	3,593,088	38.8%	38.8%
			4-cores	4.012157	2,492,425	57.5%	30.6%
			5-cores	5.506802	1,815,936	69.0%	27.1%
			6-cores	7.110453	1,406,380	76.0%	22.6%
			7-cores	7.834159	1,276,461	78.2%	9.2%
			8-cores	10.054136	994,616	83.0%	22.1%
			9-cores	11.185041	894,051	84.8%	10.1%
			10-cores	13.508867	740,255	87.4%	17.2%
			11-cores	14.839633	673,871	88.5%	9.0%
			12-cores	16.490477	606,411	89.7%	10.0%
			13-cores	19.138960	522,494	91.1%	13.8%
			14-cores	20.704514	482,986	91.8%	7.6%

Performance degrades smoothly as more cores are involved in cache line contention

Contention across 2-Sockets

- All 30 cores in 2 nodes/sockets participate in the cache line contention:

Execution Nodes	Memory node	Sockets Used	Cores used	Seconds	Ops per Sec per Core	% decrease from 1-socket
Node 0	Node 1	1-socket	15-cores	2.107396	474,519	0.0%
Nodes 0-1	Node 1	2-socket	30-cores	14.450938	69,200	85.4%
Nodes 1-2				14.897306	67,126	86.0%
Nodes 2-3				21.742537	45,993	90.4%

- There are two interesting points here:
  1) There is a huge drop in performance when going from 15-cores on1-socket to 30-cores on 2-sockets
  2) There is a smaller drop in performance when the lock's memory location is completely remote from the sockets involved in cache line contention (nodes 1-2 vs. nodes 2-3)

Contention across 2-Sockets: Increasing Core Count

- Add one core at a time, filling node/socket-0 first, then filling node/socket-1:

Execution Nodes	Memory node	Sockets Used	Cores used	Seconds	Ops per Sec per Core	% decrease from 2-core	% decrease from previous
Node 0	Node 1	1-socket	13-cores	1.649242	606,339	92.7%	9.1%
			14-cores	1.905878	524,693	93.7%	13.5%
			15-cores	1.649242	482,435	94.2%	8.1%
Nodes 0-1	Node 1	2-sockets	16-cores	1.905878	129,309	98.4%	73.2%
			17-cores	8.348480	119,782	98.6%	7.4%
			18-cores	8.264046	121,006	98.5%	-1.0%
			30-cores	15.146260	66,023	99.2%	8.5%

- We can see that the huge drop in performance occurs once we add a single core from the second socket.
- This is due to the need to go through QPI to handle the cache-to-cache traffic to resolve the cache line contention.
- This is a significant drop in performance when going through QPI.

Contention across 2-Sockets: Round-Robin vs. Fill-First

- Contention measured across 2, 8, and 14 cores
- Cores spread among 2-sockets (round-robin) vs. all cores in one socket (fill-first):

Execution Nodes	Memory Node	Sockets used	Cores per Socket used	Cores used	Seconds	Ops per Sec per Core
Node 1	Node 1	1-socket FF	2-cores	2-cores	0.120395	8,305,993
Node 0-1	Node 1	2-sockets RR	1-core	2-cores	0.314462	3,180,034
Node 1-2					0.305783	3,270,293
Node 2-3					0.453627	2,204,454
Node 1	Node 1	1-socket FF	8-cores	8-cores	1.018527	981,810
Node 0-1	Node 1	2-sockets RR	4-cores	8-cores	3.351590	298,366
Node 1-2					3.390266	294,962
Node 2-3					5.354243	186,768
Node 1	Node 1	1-socket FF	14-cores	14-cores	2.067889	483,585
Node 0-1	Node 1	2-sockets RR	7-cores	14-cores	6.214167	160,923
Node 1-2					6.275140	159,359
Node 2-3					9.471300	105,582

1) Numa effect is visible when memory is remote

2) Best performance when all cores are in one socket

Contention across 4-Sockets: Round-Robin vs. Fill-First

- Contention measured across 4, 8, and 12 cores
- Cores spread (round-robin) among 4-sockets vs. all cores in one socket (fill-first)

Execution Nodes	Memory Node	Sockets used	Cores per Socket used	Cores used	Seconds	Ops per Sec per Core
Node 1	Node 1	1-socket FF	4-cores	4-cores	0.396550	2,521,750
Node 0-3	Node 1	4-sockets RR	1-core	4-cores	1.491732	670,362
Node 1	Node 1	1-socket FF	8-cores	8-cores	0.941517	1,062,116
Node 0-3	Node 1	4-sockets RR	2-cores	8-cores	5.421381	184,455
Node 1	Node 1	1-socket FF	12-cores	12-cores	1.794806	557,163
Node 0-3	Node 1	4-sockets RR	3-cores	12-cores	8.937035	111,894
Node 0-3	Node 1	4-sockets FF	15-cores	60-cores	49.786041	20,086

- Cache line contention is clearly better when all the contention is contained within a single socket.
- For the same core count, performance degrades as more sockets are involved in the contention

Contention across 8-Sockets: Round-Robin vs. Fill-First

- Contention measured across 8, 16 and 24 cores
- Cores spread (round-robin) among 8-sockets vs. all cores in two sockets (fill-first):

Execution Nodes	Memory Node	Sockets used	Cores per Socket used	Cores used	Seconds	Ops per Sec per Core
Node 1	Node 1	1-socket FF	8-cores	8-cores	1.185326	843,650
Node 0-7	Node 1	8-sockets RR	1-core	8-cores	10.609325	94,257
Node 0-1	Node 1	2-sockets FF	16-cores	16-cores	8.886286	112,533
Node 0-7	Node 1	8-sockets RR	2-cores	16-cores	22.296164	44,851
Node 0-1	Node 1	2-sockets FF	24-cores	24-cores	12.991910	76,626
Node 0-7	Node 1	8-sockets RR	3-cores	24-cores	36.197777	27,626
Node 0-7	Node 1	8-sockets FF	15-cores	120-cores	172.782623	5,788

- Cache line contention is clearly better when all the contention is contained within as few sockets as possible.

Contention across 16-Sockets: Round-Robin vs. Fill-First

- Contention measured across 16, 32 and 64 cores
- Cores spread (round-robin) among 16-sockets vs. all cores in 1/2/4 sockets (fill-first):

Execution Nodes	Memory Node	Sockets used	Cores per Socket used	Cores used	Seconds	Ops per Sec per Core
Node 1	Node 1	1-socket FF	15-cores	15-cores	2.21096	452,292
Node 0-15	Node 1	16-sockets RR	1-core	16-cores	22.904097	43,660
Node 0-1	Node 1	2-sockets FF	15-cores	30-cores	15.706788	63,667
Node 0-15	Node 1	16-sockets RR	2-cores	32-cores	53.217117	18,791
Node 0-3	Node 1	4-sockets FF	15-cores	60-cores	74.909485	13,349
Node 0-15	Node 1	16-sockets RR	4-cores	64-cores	109.447632	9,137
Node 0-15	Node 1	16-sockets RR	15-cores	240-cores	410.881287	2,434

- Cache line contention is clearly better when all the contention is contained within as few sockets as possible.

Inter- vs Intra- Cache Line Contention Probability

Execution Nodes	Memory Node	Sockets used	Cores per Socket used	Cores used	Seconds	Ops per Sec per Core
Node 1	Node 1	1-socket FF	4-cores	4-cores	0.396550	2,521,750
Node 0-3	Node 1	4-sockets RR	1-core	4-cores	1.491732	670,362
Node 1	Node 1	1-sockets FF	8-cores	8-cores	1.185326	843,650
Node 0-7	Node 1	8-sockets RR	1-cores	8-cores	10.609325	94,257
Node 1	Node 1	1-socket FF	15-cores	15-cores	2.21096	452,292
Node 0-15	Node 1	16-sockets RR	1-core	16-cores	22.904097	43,660

- On a 4-socket/60-core system you have a 25% chance that any two random cores participating in the same cache line contention are on the same socket
- On an 8-socket/120-core system this is reduced to a 12.5% chance
- With a 16-socket/240-core system you have only a 6.25% chance

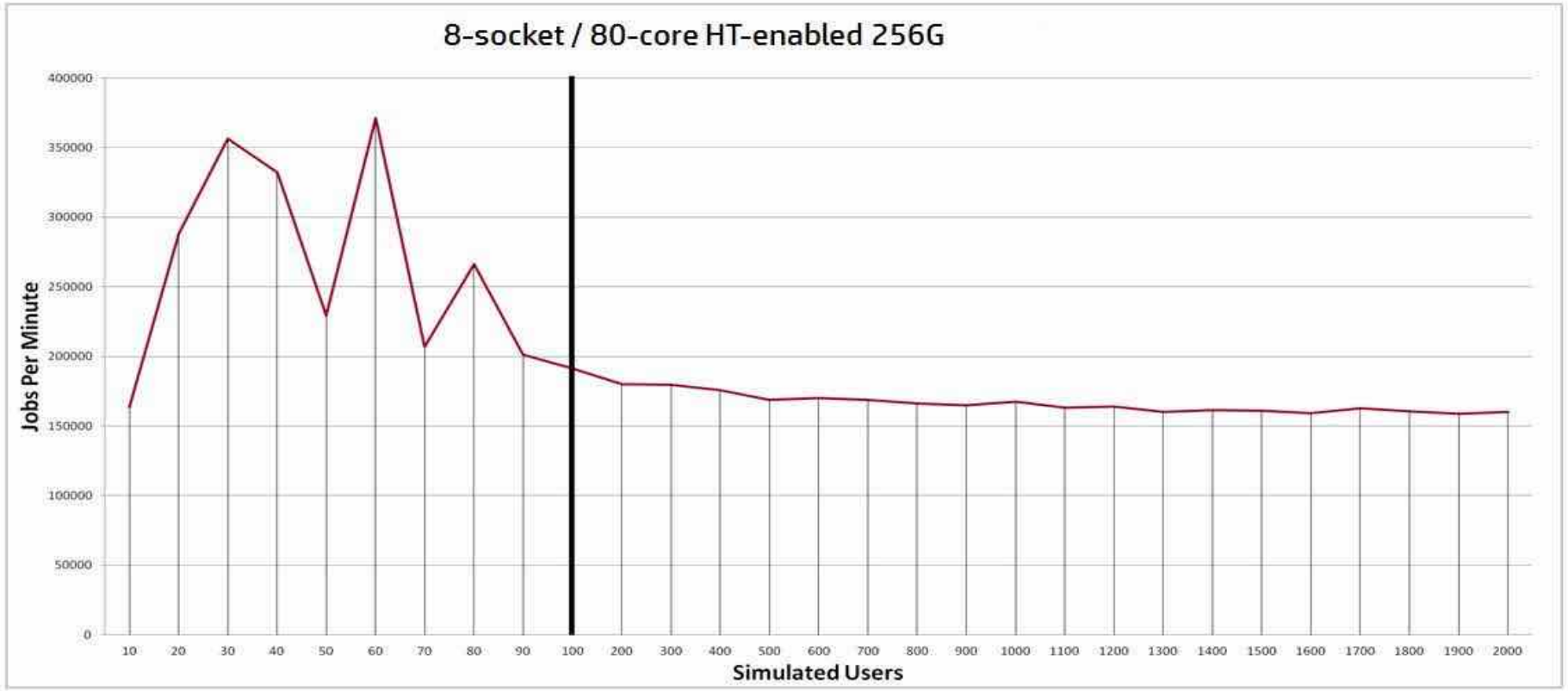
Why do we care so much?

- Many applications scale based on the number of CPUs available. For example, one or two worker threads per CPU.
- However, many applications today have been tuned for 4-socket/40-core and 8-socket/80-core Westmere platforms.
- Going from 40- or 80-cores to 240-cores (16-sockets) is a major jump.
- Scaling based only on the number of CPUs is likely to introduce significant lock and cache line contention inside the Linux kernel.
- As seen in the previous slides, the impact of cache line contention gets significantly worse as more sockets and cores are added into the system – this is a major concern when dealing with 8- and 16-socket platforms.
  - This has led us to pursue minimizing cache line contention within Linux kernel locking primitives.

Proof of Concept

Background

The AIM7 fserver workload* scales poorly on 8s/80core NUMA platform with a 2.6 based kernel



* The workload was run with ramfs.

Analysis (1-2)

From the perf -g output, we find most of the CPU cycles are spent in file_move() and file_kill().

40 Users (4000 jobs)

```
+  9.40%  reaim  reaim          [.] add_int
+  6.07%  reaim  libc-2.12.so   [.] strncat
.....
-  1.68%  reaim  [kernel.kallsyms]  [k] _spin_lock
  - _spin_lock
     + 50.36% lookup_mnt
     + 7.45% __d_lookup
     + 6.71% file_move
     + 5.16% file_kill
     + 2.46% handle_pte_fault
```

Proportion of file_move() = 1.68% * 6.71% = 0.11%

Proportion of file_kill() = 1.68% * 5.16% = 0.09 %

Proportion of file_move() + file+kill() = 0.20%

400 users (40,000 jobs)

```
- 79.53%  reaim  [kernel.kallsyms]  [k] _spin_lock
  - _spin_lock
     + 34.28% file_move
     + 34.20% file_kill
     + 19.94% lookup_mnt
+  8.13%  reaim  [kernel.kallsyms]  [k] mutex_spin_on_owner
+  0.86%  reaim  [kernel.kallsyms]  [k] _spin_lock_irqsave
+  0.63%  reaim  reaim          [.] add_long
```

Proportion of file_move() = 79.53% * 34.28% = 27.26%

Proportion of file_kill() = 79.53% * 34.20% = 27.20%

Proportion of file_move() + file+kill() = 54.46%

This is significant spinlock contention!

Analysis (2-2)

We use the ORC tool to monitor the coherency controller results

(ORC is a platform dependent tool from HP that reads performance counters in the XNC node controllers)

Coherency Controller Transactions Sent to Fabric Link (PRETRY number)

Socket	Agent	10users	40users	400users
0	0	17,341	36,782	**399,670,585**
0	8	36,905	45,116	**294,481,463**
1	0	0	0	49,639
1	8	0	0	25,720
2	0	0	0	1,889
2	8	0	0	1,914
3	0	0	0	3,020
3	8	0	0	3,025
4	1	45	122	1,237,589
4	9	0	110	1,224,815
5	1	0	0	26,922
5	9	0	0	26,914
6	1	0	0	2,753
6	9	0	0	2,854
7	1	0	0	6,971
7	9	0	0	6,897

- PRETRY indicates the associated read needs to be re-issued.
- We can see that when users increase, PRETRY on socket 0 increases rapidly.
- There is serious cache line contention on socket 0 with 400 users. Many jobs are waiting for the memory location on Socket 0 which contains the spinlock.
- PRETRY number on socket 0:
  400 users = 400M + 294M = 694M

Removing Cache Line Contention

- Code snippet from the 2.6 based kernel for file_move() and file_kill():

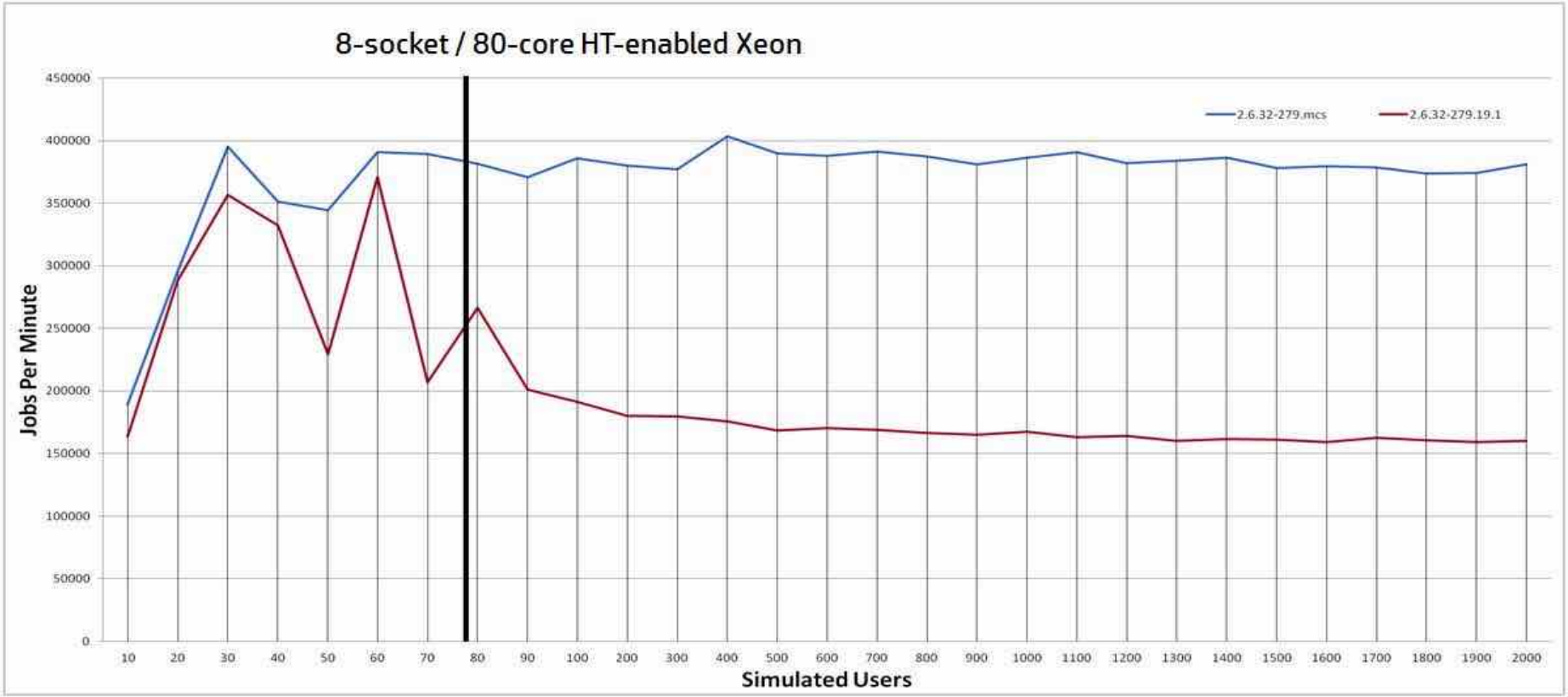
```
extern spinlock_t files_lock;
#define file_list_lock()   spin_lock(&files_lock);
#define file_list_unlock() spin_unlock(&files_lock);

void file_move(struct file *file,
               struct list_head *list)
{
    if (!list)        return;
    file_list_lock();
    list_move(&file->f_u.fu_list, list);
    file_list_unlock();
}
```

```
void file_kill(struct file *file)
{
    if (!list_empty(&file->f_u.fu_list)) {
        file_list_lock();
        list_del_init(&file->f_u.fu_list);
        file_list_unlock();
    }
}
```

- Contention on this global spinlock is the cause of all the cache line contention
- We developed a prototype MCS/Queued spinlock to see its effect on cache line traffic
  - MCS/Queued locks are NUMA aware and each locker spins on local memory rather than the lock word
  - Implementation is available in the back-up slides
- No efforts were made to make this a finer grained lock

Prototype Benchmark Results

Comparing the performance of the new kernel (blue line) vs. the original kernel (red line)



2.4x improvement in throughput with the MCS/Queued spinlock prototype!

Prototype Analysis (1-2)

perf –g output of the kernel with MCS/Queued spinlock prototype:

400 users(40000 jobs)

44.71% reaim [kernel.kallsyms] [k] _spin_lock

-60.94%-- lookup_mnt

....

22.13% reaim [kernel.kallsyms] [k] mutex_spin_on_owner

-96.16%-- __mutex_lock_slowpath

.....

1.19% reaim [kernel.kallsyms] [k] file_kill

1.19% reaim [kernel.kallsyms] [k] file_move

Proportion of lookup_mnt() = 27.2%

Proportion of __mutex_lock_slowpath() = 21.3%

Proportion of file_move() + file+kill() 2.38%

- The proportion of time for the functions file_move() and file_kill() is now small in the 400 users case when using an MCS/Queued spinlock (dropped from 54.46% to 2.38%)
- The functions lookup_mnt() and __mutex_lock_slowpath() now take most of the time.

Prototype Analysis (2-2)

Coherency controller results of the kernel with the MCS/Queued spinlock

Coherency Controller Transactions Sent to Fabric Link (PRETRY number)

Socket	Agent	10users	40users	400users
0	0	18,216	24,560	**83,720,570**
0	8	37,307	42,307	**43,151,386**
1	0	0	0	0
1	8	0	0	0
2	0	0	0	0
2	8	0	0	0
3	0	0	0	0
3	8	0	0	0
4	1	52	222	16,786
4	9	28	219	10,068
5	1	0	0	0
5	9	0	0	0
6	1	0	0	0
6	9	0	0	0
7	1	0	0	0
7	9	0	0	0

- We can see that as users increase, PRETRY in socket 0 also increases – but it is significantly lower than the kernel without the MCS/Queued lock.
- The PRETRY number for socket 0:
  400 users = 84M + 43M = 127M.
- This value is about 1/5 of the original kernel (694M).
- This shows the MCS/Queued spinlock algorithm reduces the PRETRY traffic that occurs in file_move() and file_kill() significantly even though we still have the same contention on the spinlock.

Proof of Concept - Conclusions

- The MCS/Queued spinlock improved the throughput of large systems just by minimizing the inter-socket cache line traffic generated by the locking algorithm.
- The MCS/Queued spinlock did not reduce the amount of contention on the actual lock. We have the same number of spinners contending for the lock. No code changes were made to reduce lock contention.
- However, the benchmark throughput improved from ~160,000 to ~390,000 jobs per minute due to the reduced inter-socket cache-to-cache traffic.
  - System time spent spinning on the lock dropped from 54% to 2%.
- Lock algorithms can play a huge factor in the performance of large-scale systems
  - The impact of heavy lock contention on a 240-core system is much more severe than the impact of heavy lock contention on a 40-core system
- This is not a substitute for reducing lock contention… Reducing lock contention is still the best solution, but attention to lock algorithms that deal with contention *is* extremely important and can yield significant improvements.

MCS Locks

MCS Locks in the Kernel

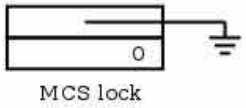
- A new locking primitive inside Linux
  - Currently only used inside other locking primitives and not general usage throughout the kernel
- Each locker spins on a local variable while waiting for the lock rather than spinning on the lock itself.
- Eliminates much of the cache-line bouncing experienced by simpler locks, especially in the contended case when simple CAS (Compare-and-Swap) calls fail.
- Fair, passing the lock to each locker in the order that the locker arrived.
- Specialized cancelable MCS locking.
- Applied internally to mutexes and rwsems.
  - Specialized cancelable MCS locking.
  - Failed attempts have been made for regular ticket spinlocks – but it cannot fit inside 4-bytes!

MCS Locks

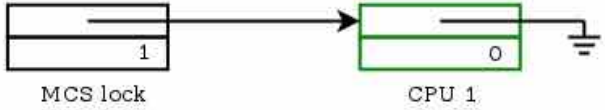
```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked;
};

void mcs_spin_lock(struct mcs_spinlock **lock,
                   struct mcs_spinlock *node);

void mcs_spin_unlock(struct mcs_spinlock **lock,
                     struct mcs_spinlock *node);
```

- Little more complicated than a regular spinlock.
- Pointer in the "main" lock is the tail of the queue of waiting CPUs.

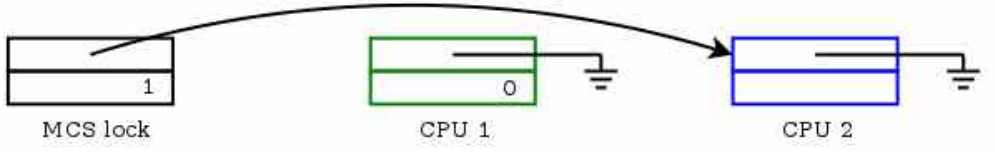
Empty lock



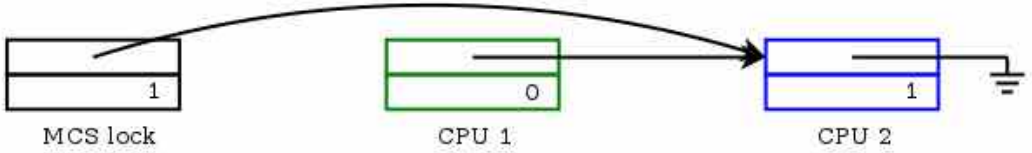
CAS cpu1 mcs_spinlock with lock, acquire it.



CAS cpu2 mcs_spinlock with lock, prev != NULL, lock taken.



Repeat previous operation: CAS cpu2 with CPU1's mcs_spinlock



CPU1 done with the lock.

Cancellable MCS Locks

- The cancellable MCS lock is a specially tailored lock for MCS: when needing to reschedule, we need to abort the spinning in order to block.
- The most popular use of the cancellable MCS lock is embedded into the optimistic spinning path of both mutexes and rw-semaphores.

Mutexes

Mutex Overview

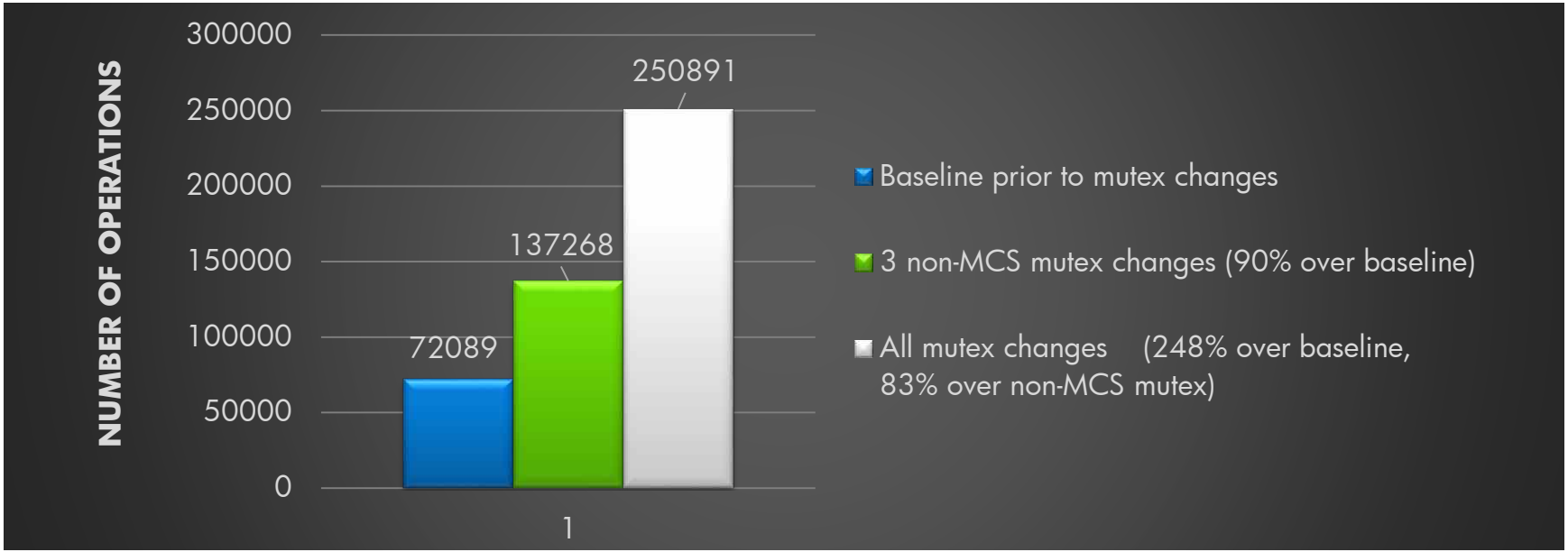
- Allows a single thread to enter a critical section at a time.
  - Introduced as an optimized alternative to regular semaphores.
- Sleeping lock – beneficial for long hold times (larger critical regions).
- More beneficial to use a mutex if the critical section is long.
- When taking the lock, there are three possible paths that can be taken, depending on the state of the lock at that point.
  1. Fast path:
     - Attempt to acquire the lock by atomically decrementing the count variable. If previous mutex count was 1, return, otherwise try the slow path. Architecture specific.
  2. Mid path (optimistic spinning):
     - When the lock is contended, instead of immediately adding the task to the wait-queue and blocking, busy-wait as a regular spinlock.
  3. Slow path:
     - Take the `wait_lock` spinlock, add the task to the list and block until the lock is available.

Mutex Lock Improvements

- Reduce the number of atomic operations
  - Read values first - only use `xchg/cmpxchg` if it might be successful. Also known as *compare-compare-and-swap*.
  - Reduces cache line contention
  - For example:
    - Orig: `if (atomic_cmpxchg(&lock->count, 1, 0) == 1)`
    - New: `if ((atomic_read(&lock->count) == 1) && (atomic_cmpxchg(&lock->count, 1, 0) == 1))`
- Slow path optimizations
  - Shorten critical region.
  - Optimize wait_list handling, etc.
- Add the MCS lock inside of a Mutex
  - Fixes fairness and cache line contention problems with optimistic spinning
    - The mutex is granted in the order of which it arrives (similar to ticket spinlocks).
    - Spin on our own cache line.

Mutex Lock Performance Improvements

- The previous changes were back-ported to a 3.0 based kernel
- Performance measurements were done with a popular Java based workload (higher number of operations is better)
- System used: HP ConvergedSystem 900 for SAP HANA
  - 16-sockets, 240-cores, 480-threads, 12-TB memory



Read / Write Semaphores

Read/Write Semaphore Overview

- Read/write semaphores allow the lock to be shared among readers or exclusively owned by one writer at a time.
  - Two kinds of implementations: faster xadd, slower spinlock.
  - Sleeping lock - beneficial for long hold times (larger critical regions).
  - Fairness towards writers.

- Lock acquisition handling is *similar* to mutexes: fast path, mid path (with optimistic spinning), and slow path.

Read/Write Semaphore Problems

- Many workloads exposed that mutexes could out-perform writer semaphores.
  - Semantically writer semaphore are identical to mutexes.
  - Simply replacing the semaphore with a mutex would return performance.
  - Users had to consider this performance penalty when choosing what type of primitive to use.

- Two issues cause this problem:
  1. Writer starvation
  2. Lack of optimistic spinning on lock acquisition

Writer Lock Stealing

- Read/write semaphores suffer from a strict, FIFO sequential write-ownership of rwsems.
- Mutexes do not suffer from this as they handle checks for waiters differently.
- To resolve this issue Writer Lock Stealing was implemented in read/write lock semaphores:
  - While task X is in the process of acquiring a lock, process Y can atomically acquire the lock and essentially steal it.
  - Results in better CPU usage (less idle time) and fewer context switches.

Optimistic Spinning

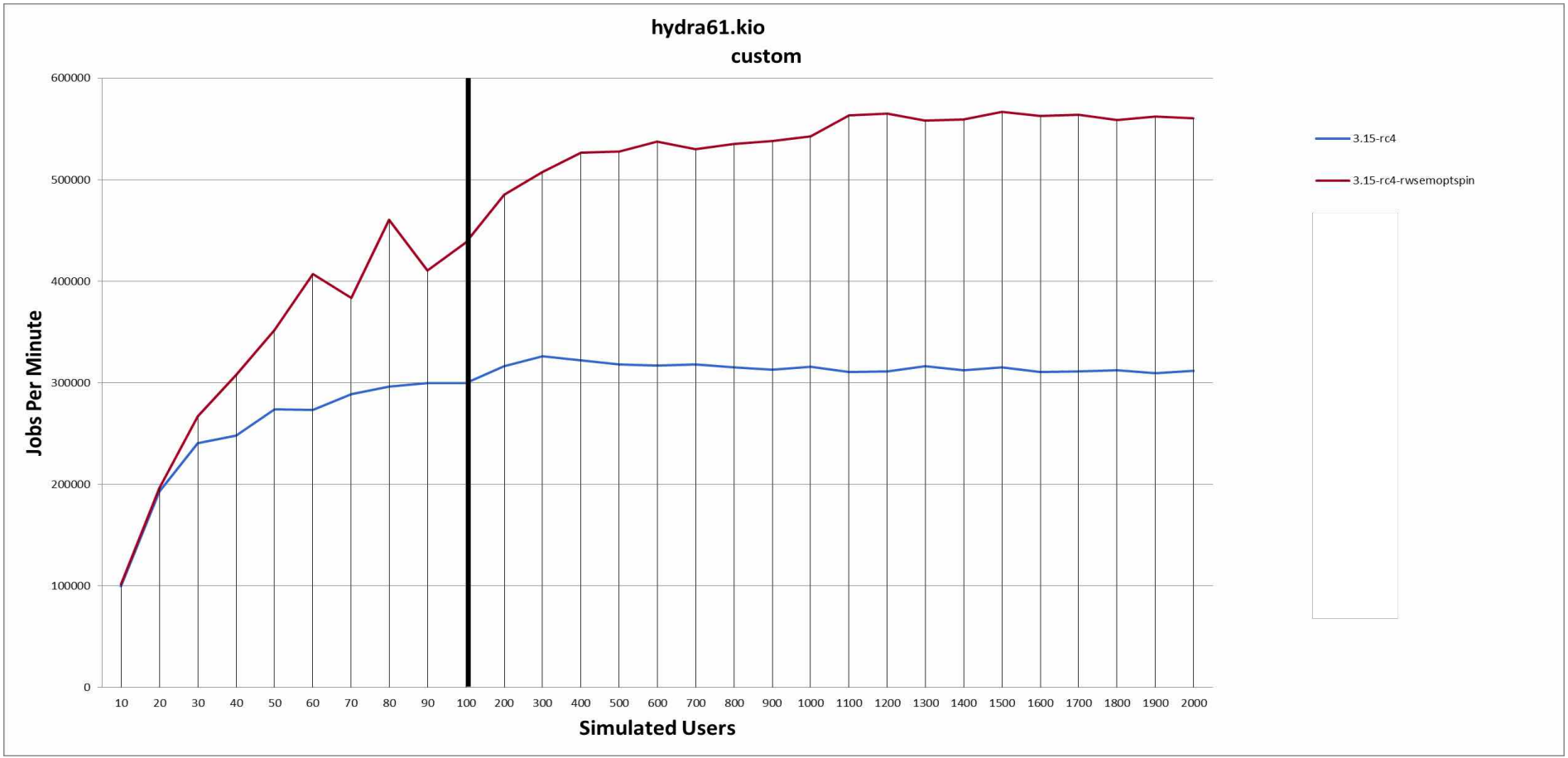
- Read/write semaphores would try to acquire the lock once. If not available the locker would immediately block waiting for the lock
  - The overhead of blocking caused significant performance problems.
- When the semaphore is contended, instead of immediately adding the locker to the wait queue and blocking, we now busy-wait for the semaphore (similar to a regular spinlock.
  - If the current lock holder is not running or needs to be rescheduled then block waiting for the semaphore.
  - Based on what mutexes have been doing since early 2009. With roots in Real-Time.
  - Adds the notion of a hybrid locking type.
  - Rwsems can still be used in sleeping context.
- Particularly useful locks that have short or medium hold times.

Caveats to Optimistic Spinning with rwsems

1. The 'struct rw_semaphore' data structure is now the largest lock in the kernel.
   - Larger structure sizes mean more CPU cache and memory footprint.
   - Dave Chinner reported that it increased the size of *xfs_inode* by 4%.
     - *"That's enough to go from having a well balanced workload to not being able to fit the working set of inodes in memory."*

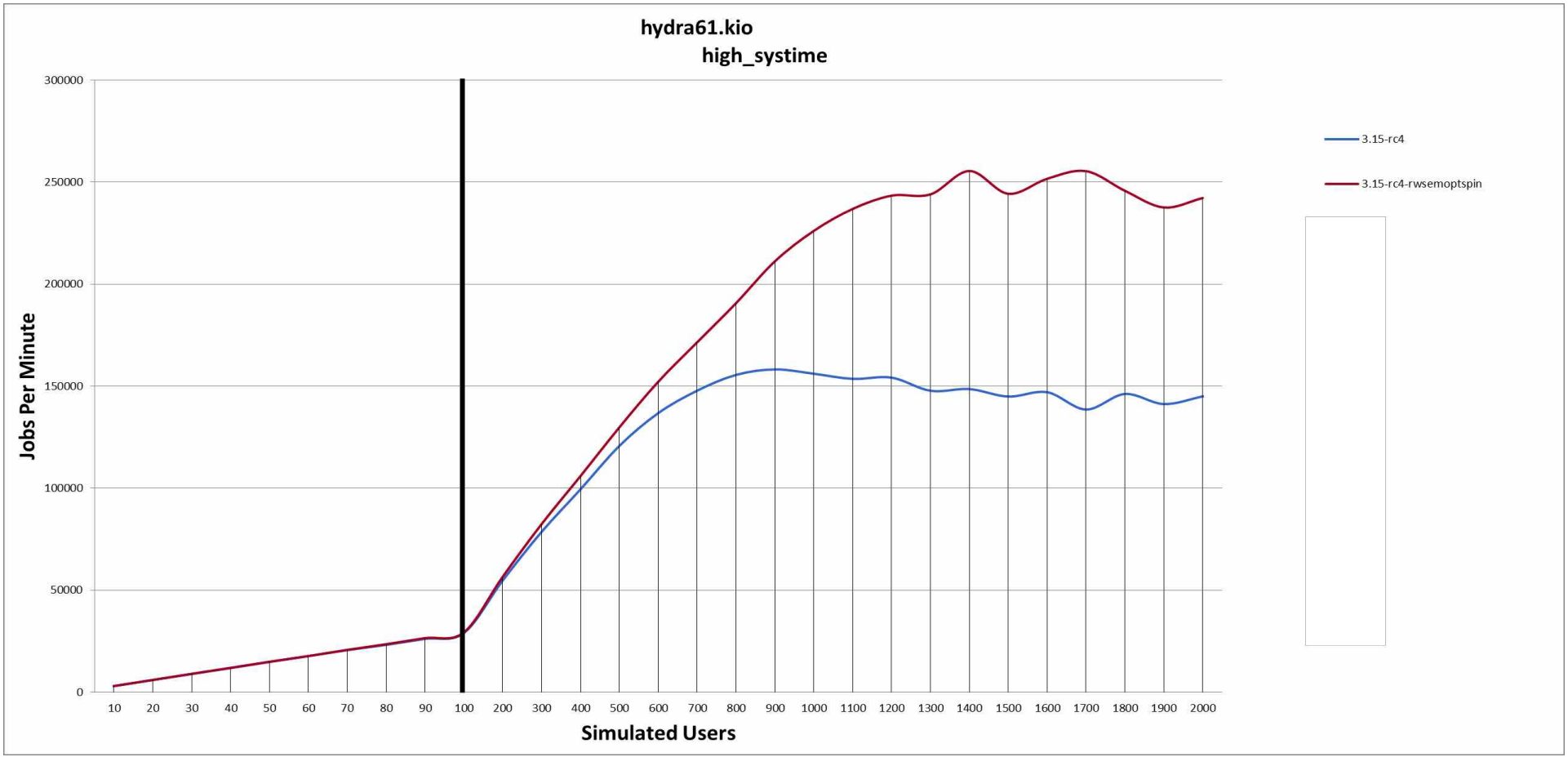
2. When workloads use both lock readers and writers, there is a chance that optimistic spinning can make writers spin excessively for the lack of ownership.
   - When readers acquire the lock there is no concept of ownership
   - Consequently writers may spin excessively as there is no owner to optimistically spin on
   - This is a major difference between read/write semaphores and mutexes

AIM7 Benchmark Results (1-2)



~ 1.85x improvement in throughput (JPM)!

AIM7 Benchmark Results (2-2)



~ 1.7x improvement in throughput (JPM)!

Read / Write Spinlocks

Read/Write Spinlock Overview and Problems

- Overview:
  - Reader/writer variant of a regular spinlock.
  - Known to perform significantly slower than regular spinlocks.
- Problems:
  - Unfair: readers can starve writers – new readers may acquire the lock even when writers are waiting.
  - Unfair on a NUMA machine: Better chance of acquiring if the lock is on the local memory node.

Queue-based read/write lock implementation

- Uses spinlock as a waitqueue, thus relying on its ticket fairness.
- Does not enlarge rw locks (fits in same 8-bytes as the regular rwlock_t)
  - Reader and writer counters into a single integer (4-bytes) + waitqueue (4-bytes)
- While it solves the fairness problem, there are really no performance gains yet.
  - We need a way of having MCS-like spinlocks without increasing its size.
- Queued spinlocks:
  - Not yet in the kernel.
  - An alternative to the ticket spinlock.
  - Improve scenarios under high lock contention.
  - Perform better than ticket spinlocks in the uncontended case.
    - When unlocking: read-modify-write (add) vs simple write: from ~14.1ns down to ~8.8ns.
    - Smaller systems can see an improvement.

Performance Results

- Gains in *disk* AIM7 workload

Changes	% increase
ramdisk	95%
ext4	116%

Today – without qspinlock

```
26.19% reaim [kernel.kallsyms] [k] _raw_spin_lock
--- _raw_spin_lock
              |--47.28%-- evict
              |--46.87%-- inode_sb_list_add
              |--1.24%-- xlog_cil_insert_items
              |--0.68%-- __remove_inode_hash
              |--0.67%-- inode_wait_for_writeback
              [...]
```

With qspinlock

```
2.40% reaim [kernel.kallsyms] [k] queue_spin_lock_slowpath
            |--88.31%-- _raw_spin_lock
            |             |--36.02%-- inode_sb_list_add
            |             |--35.09%-- evict
            |             |--16.89%-- xlog_cil_insert_items
            |             |--6.30%-- try_to_wake_up
            |             |--2.20%-- _xfs_buf_find
            |             |--0.75%-- __remove_inode_hash
            |             |--0.72%-- __mutex_lock_slowpath
            |             |--0.53%-- load_balance
            |--6.02%-- _raw_spin_lock_irqsave
            |             |--74.75%-- down_trylock
            [...]
```

Lockref

Reference counting in the Kernel

- Normally used to track the lifecycle of data structures.
  - A reference count of zero means the structure is unused and is free to be released
  - A positive reference count indicates how many tasks are actively referencing this structure
  - This is usually handled in put() and get() calls
  - atomic_t and struct kref are some techniques for reference counters
- When embedded into a data structure, it is not uncommon to have to acquire a lock (spinlock) just to increment or decrement the reference count variable.
- Under heavy load, this lock can become quite contended.
  - What about converting the variable to an atomic type and avoid taking the lock?

Lockref

- Generic mechanism to atomically update a reference count that is protected by a spinlock without actually acquiring the spinlock itself.
- The lockref patch introduced a new mechanism for a lockless update of a spinlock protected reference count.
- Bundle a 4-byte spinlock and a 4-byte reference count into a single 8-byte word that can be updated atomically while no one is holding the lock.
- The VFS layer makes heavy use of lockref for dentry operations.
  - Workloads that create lots of fs activity can be bottlenecked by the spinlock contention on the dentry reference count update.
  - The lockref patch resolves this contention by doing the update without taking the lock.

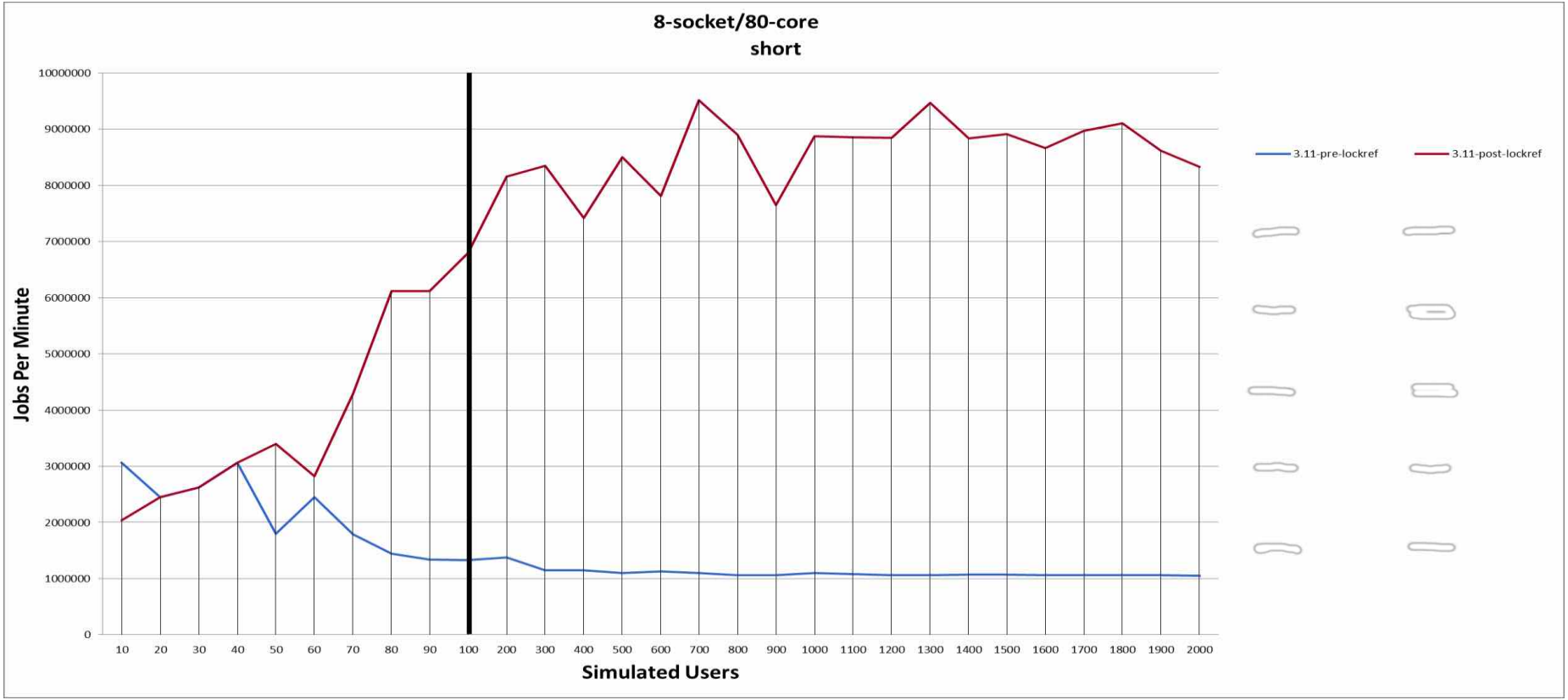
Lockref

```
struct lockref {
        spinlock_t lock;
        unsigned int count;
};

void lockref_get(struct lockref *lockref);
int lockref_get_not_zero(struct lockref *lockref);
int lockref_put_or_lock(struct lockref *lockref);
```

1. Fast path:
   - Atomically check that the lock is not taken so that lockless updates can never happen while someone else holds the spinlock.
   - Do the reference count update using a CAS loop. Architectures must define ARCH_USE_CMPXCHG_LOCKREF
   - Semantically identical to doing the reference count update protected by the lock
2. Slow path:
   - Take the spinlock, then normally inc/dec the counter the traditional way.

Benchmark Results (1-2)

Comparing a pre (blue line) and post (red line) lockref vanilla 3.11 kernel



~ 8x improvement in throughput (JPM)!

Benchmark Results (2-2)

The following before/after perf output shows the reduced time spent spinning on the lock:

Prior to the lockref patches

```
-  83.74%  reaim [kernel.kallsyms] [k] _raw_spin_lock
   - _raw_spin_lock
      - 49.96% dget_parent
         - __fsnotify_parent
            + 20.16% __fput
            + 20.10% security_file_permission
            + 20.00% fsnotify_access
            + 19.92% do_sys_open
            + 19.82% security_file_open
      - 49.68% dput
         - 99.97% __fsnotify_parent
            + 20.18% security_file_permission
            + 20.02% __fput
            + 20.02% fsnotify_access
            + 20.00% do_sys_open
            + 19.78% security_file_open

-   2.24%  reaim [kernel.kallsyms] [k] update_cfs_rq_blocked_load
-   0.39%  reaim [kernel.kallsyms] [k] intel_pmu_disable_all
```

After applying the lockref patches

```
-  13.84% reaim [kernel.kallsyms] [k] _raw_spin_lock_irqsave
  - _raw_spin_lock_irqsave
     + 49.07% tty_ldisc_try
     + 48.80% tty_ldisc_deref

-  12.97%  reaim [kernel.kallsyms]  [k] lg_local_lock
  - lg_local_lock
     + 72.31% mntput_no_expire
     + 20.28% path_init
     + 4.55% sys_getcwd
     + 2.71% d_path

-   5.34%  reaim [kernel.kallsyms]  [k] _raw_spin_lock
  - _raw_spin_lock
     + 41.54% d_path
     + 39.37% sys_getcwd
     + 4.10% prepend_path
     + 1.86% __rcu_process_callbacks
     + 1.30% do_anonymous_page
     + 0.95% sem_lock
     + 0.95% process_backlog
     + 0.70% enqueue_to_backlog
     + 0.64% unix_stream_sendmsg
     + 0.64% unix_dgram_sendmsg
```

Futexes

Futex Bottlenecks

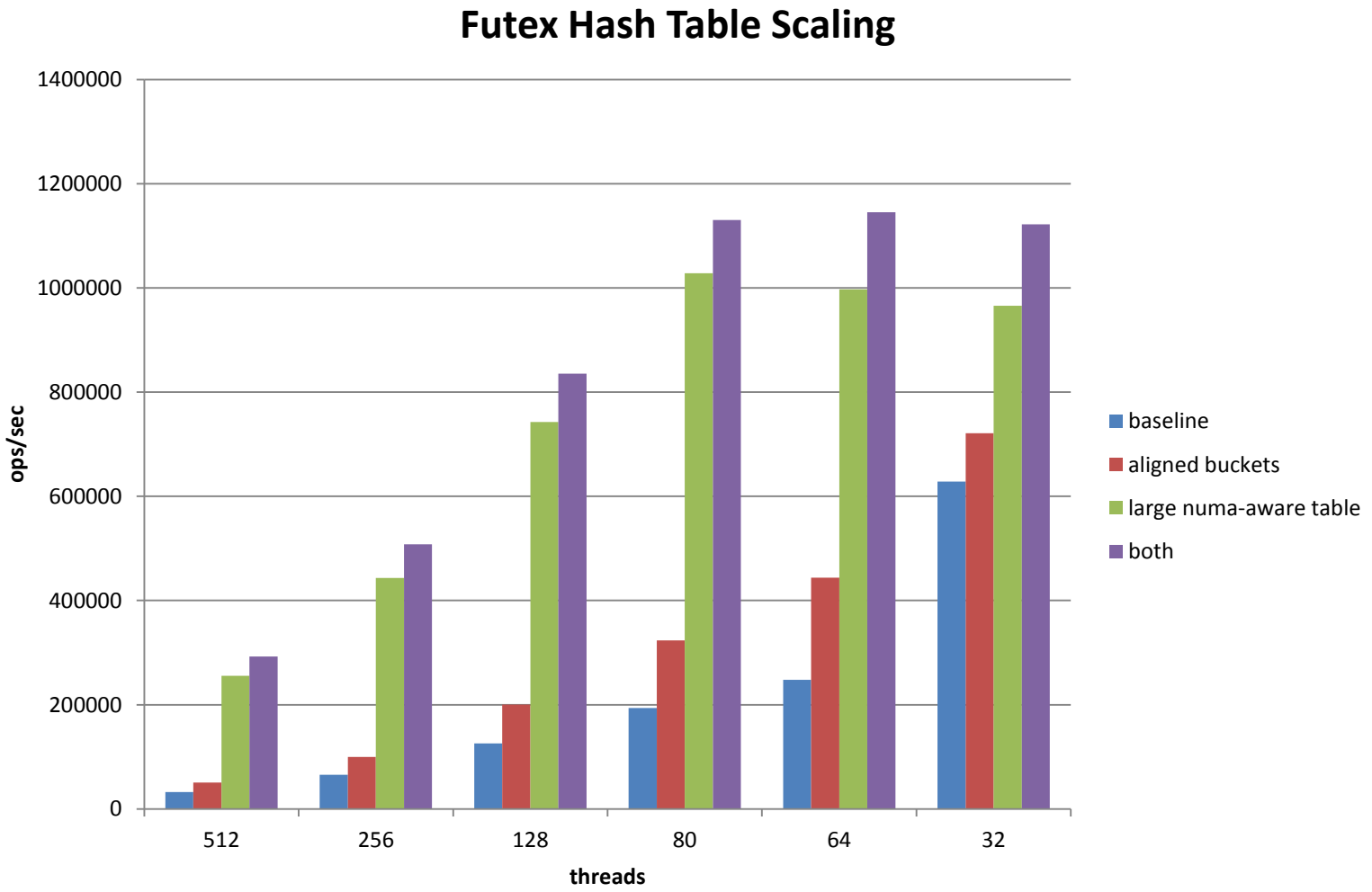
- Impacts all types of futexes.
- Futexes suffer from its original design: a unique, shared hash table.
  - Each bucket in a priority list (FIFO), serialized by a spinlock (*hb->lock*).
  - For NUMA systems, all memory for the table is allocated on a single node.
  - Very small size (256 hash buckets) for today's standards. More collisions.
  - Both problems hurt scalability, considerably.
- *hb->lock* hold times can become quite large:
  - Task, mm/inode refcounting.
  - Wake up tasks.
  - Plist handling.

Larger, NUMA-aware Hash Table (1-2)

- 256 hash buckets per CPU
  - 256 * nCPUs cache line aligned hash buckets
  - Less collisions and more spinlocks leading to more parallel futex call handling.
- Distribute the table among NUMA nodes instead of a single one.
- The perfect hash size will of course have one to one hb:futex ratio.

- Performance measured by stressing the internal futex hashing logic.
- As expected, the benefits become more evident as more futexes are used. On a 1Tb, 80-core, 8-socket x86-64 (Westmere):
  - 1024 * 32 futexes -> ~78% throughput increase.
  - 1024 * 512 futexes -> ~800% throughput increase.

NUMA-aware Larger Hash Table (2-2)



Lockless Waitqueue Size

- A common misuse of futexes is to make FUTEX_WAKE calls when there are no waiters.
- In FUTEX_WAKE, there's no reason to take the *hb->lock* if we already know the list is empty and thus one to wake up.
- Use an independent atomic counter to keep track of the list size.
- This can drastically reduce contention on the hash bucket spinlock.

Futex Performance Improvements

- The following before/after perf output from a customer database workload on a large 16-socket / 240-core system shows the reduced time spent spinning on hb->lock:

Prior to the futex patches

```
43.71%  762296 xxx [kernel.kallsyms] [k] _raw_spin_lock
            --- _raw_spin_lock
               |
               |--53.74%-- futex_wake
               |          do_futex
               |          sys_futex
               |          system_call_fastpath
               |          |
               |          |--99.40%-- 0x7fe7d44a4c05
               |          |           zzz
               |--45.90%-- futex_wait_setup
               |          futex_wait
               |          do_futex
               |          sys_futex
               |          system_call_fastpath
               |          0x7fe7ba315789
               |          syscall
               ...
```

After applying the futex patches

```
0.10%    49  xxx  [kernel.kallsyms] [k] _raw_spin_lock
              --- _raw_spin_lock
                 |
                 |--76.06%-- futex_wait_setup
                 |          futex_wait
                 |          do_futex
                 |          sys_futex
                 |          system_call_fastpath
                 |          |
                 |          |--99.90%-- 0x7f3165e63789
                 |          |          syscall|
                 |          ...
                 |--6.27%-- futex_wake
                 |          do_futex
                 |          sys_futex
                 |          system_call_fastpath
                 |          |
                 |          |--54.56%-- 0x7f317fff2c05
                 ...
```

Future Work

Possible Future Work

- Internal API for wait-wound, real-time and regular mutexes.
  - All should make use of optimistic spinning functionality.
  - Better, clearer and more maintainable code.
- Queued spinlocks.
  - Based on the MCS locks but the spinlock still fits in a 32-bit word.
- Spinning futexes.
  - Allows userspace to overcome the preemption problem with building spinlocks.
  - Avoid blocking calls and reduce scheduling overhead by spinning for a short time.

Acknowledgements

- HP Linux Kernel Performance team contributing this work:
  - Waiman Long
  - Jason Low
  - Davidlohr Bueso
  - Scott Norton
  - Aswin Chandramouleeswaran
- Linux kernel community
  - Peter Zijlstra
  - Linus Torvalds
  - Ingo Molnar
  - Thomas Gleixner
  - Paul McKenney
  - and many others…

Thank you

Back-up Slides

General Locking (1-2)

- Locks available to kernel hackers:
  1. *Spinning:* spinlock, rwlock.
  2. *Sleeping:* rwsem, semaphore, mutex.
  3. Others: RCU, etc.

- What can contribute to scaling problems when using locks?
  - Length of the critical region
    - Reducing the length of the critical section can certainly help alleviating lock contention.
    - Choice of locking type can be paramount.
  - Lock granularity
    - Fine-grained locks are usually more scalable than coarse-grained locks. It also can shorten the critical region, allowing fewer CPUs to contend for the lock at any given time.
  - Cache line bouncing
    - Tasks spinning on a lock will try to fetch the lock cache line repeatedly. If the lock-protected data structure is in the same cache line, it can significantly slow down the progress of the lock holder leading to a much longer lock hold time.

General Locking (2-2)

- Other considerations
  - Lock overhead
    - The extra resources for using locks, like the memory space allocated for locks, the CPU time to initialize and destroy locks, and the time for acquiring or releasing locks. The more locks a program uses, the more overhead associated with the usage.
    - This overhead, for instance with space, can cause a workload's working set to significantly increase, having to go further down in the memory hierarchy and therefore impacting performance.
  - Read:write ratio
    - Ration between the number of read-only critical regions to the number of regions where the data in question is modified. Read locks can, of course, have multiple tasks acquire the lock simultaneously.

MCS/Queued Lock Prototype (1-2)

We developed a prototype MCS/Queued lock to see the effect on cache line traffic

(MCS/Queued locks are NUMA aware and each locker spins on local memory rather than the lock word)

```
typedef struct _local_qnode {
    volatile bool waiting;
    volatile struct _local_qnode *volatile next;
} local_qnode;

static inline void
mcsfile_lock_acquire(mcsglobal_qlock *global,
                     local_qnode_ptr  me)
{
    local_qnode_ptr pred;

    me->next = NULL;
    pred = xchg(global, me);
    if (pred == NULL)
           return;

    me->waiting = true;
    pred->next = me;

    while (me->waiting);  /*spin on local mem*/
}
```

```
static inline void
mcsfile_lock_release(mcsglobal_qlock *global,
                     local_qnode_ptr  me)
{
    local_qnode_ptr succ;

    if (!(succ = me->next))  {
        if ( cmpxchg(global, me, NULL) == me ) return;
        do {
             succ = me->next;
         } while (!succ);  /* wait for succ ready */
     }
    succ->waiting = false;
}
```

MCS/Queued Lock Prototype (2-2)

Replacing the files_lock spinlock with the prototype mcsfiles_lock MCS/Queued spinlock

```
extern mcsglobal_qlock mcsfiles_lock;
#define file_list_lock(x) mcsfile_lock_acquire(&mcsfiles_lock, &x);
#define file_list_unlock(x) mcsfile_lock_release(&mcsfiles_lock, &x);
```

```
void file_move(struct file *file,
               struct list_head *list)
{
    volatile local_qnode lq;

    if (!list)
           return;
    file_list_lock(lq);
    list_move(&file->f_u.fu_list, list);
    file_list_unlock(lq);
}
```

```
void file_kill(struct file *file)
{
    volatile local_qnode lq;

    if (!list_empty(&file->f_u.fu_list)) {
        file_list_lock(lq);
        list_del_init(&file->f_u.fu_list);
        file_list_unlock(lq);
    }
}
```

AIM7 Benchmark Suite

- Traditional UNIX system-level benchmark (written in C).
- Multiple forks, each of which concurrently executes a common, randomly-ordered set of subtests called *jobs*.
- Each of the over fifty kind of jobs exercises a particular facet of system functionality
  - Disk IO operations, process creation, virtual memory operations, pipe I/O, and compute-bound arithmetic loops.
  - *AIM7* includes disk subtests for sequential reads, sequential writes, random reads, random writes, and random mixed reads and writes.
- An AIM7 run consists of a series of subruns with the number of tasks, *N*, being increased after the end of each subrun.
- Each subrun continues until each task completes the common set of jobs. The performance metric, "Jobs completed per minute", is reported for each subrun.
- The result of the entire *AIM7* run is a table showing the performance metric versus the number of tasks, *N*.
- Reference: *"Filesystem Performance and Scalability in Linux 2.4.17"*, 2002.

perf-bench futex (1-2)

- To measure some of the changes done by the futex hastable patchset, a futex set of microbenchmarks are added to *perf-bench*:
  - perf bench futex [<operation> <all>]

- Measures latency of different operations:
  - Futex hash
  - Futex wake
  - Futex requeue/wait

perf-bench futex (2-2)

```
$ perf bench futex wake
# Running 'futex/wake' benchmark:
Run summary [PID 4028]: blocking on 4 threads (at futex 0x7e20f4), waking up 1 at a time.

[Run 1]: Wokeup 4 of 4 threads in 0.0280 ms
[Run 2]: Wokeup 4 of 4 threads in 0.0880 ms
[Run 3]: Wokeup 4 of 4 threads in 0.0920 ms
…
[Run 9]: Wokeup 4 of 4 threads in 0.0990 ms
[Run 10]: Wokeup 4 of 4 threads in 0.0260 ms
Wokeup 4 of 4 threads in 0.0703 ms (+-14.22%)
```

```
$ perf bench futex hash
# Running 'futex/hash' benchmark:
Run summary [PID 4069]: 4 threads, each operating on 1024 futexes for 10 secs.

[thread 0] futexes: 0x1982700 ... 0x19836fc  [ 3507916 ops/sec ]
[thread 1] futexes: 0x1983920 ... 0x198491c [ 3651174 ops/sec ]
[thread 2] futexes: 0x1984ab0 ... 0x1985aac [ 3557171 ops/sec ]
[thread 3] futexes: 0x1985c40 ... 0x1986c3c  [ 3597926 ops/sec ]

Averaged 3578546 operations/sec (+- 0.85%), total secs = 10
```