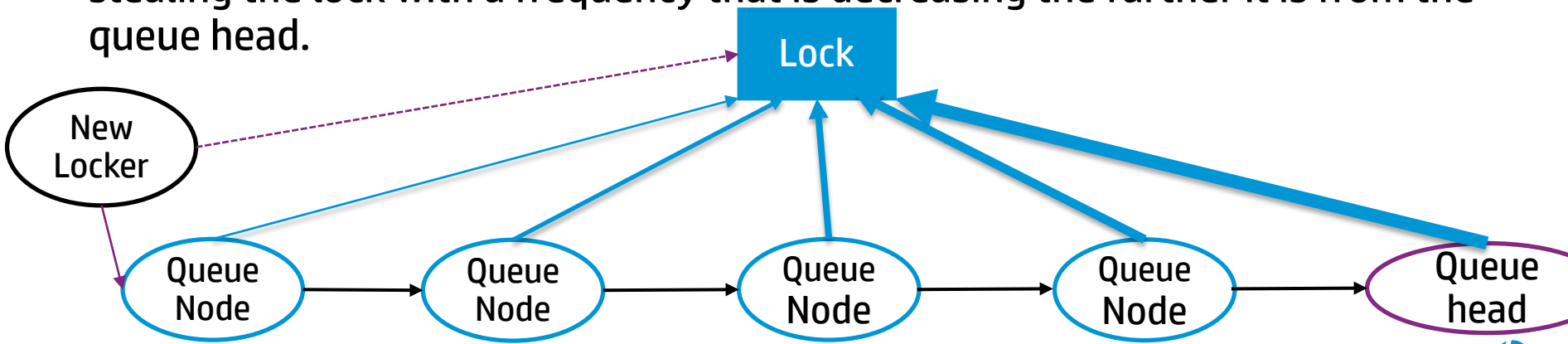
# Unfair Queued Spinlocks and Transactional Locks

Waiman Long / Aug 21, 2015
HP Server Linux Performance & Scalability Team
Version 1.0

# Unfair Queued Spinlocks

# Queued Spinlocks in a VM

- The queued spinlocks code that was merged into the 4.2 kernel provided 2 kind of locks in a VM:
  1. Para-virtualized queued spinlocks that supports both KVM and Xen.
  2. Unfair byte locks (a simple test-and-set lock) for all the other virtualization technologies (VMware, Microsoft's Hyper-V, etc) as long as the kernel knows that it is running under virtualization.
- Unfair queued spinlocks is a fairer replacement for the unfair byte locks which don't scale well with large vCPU counts.
- All the vCPUs in each of the nodes in the queue can opportunistically try stealing the lock with a frequency that is decreasing the further it is from the queue head.



# Unfair Queued Spinlocks

The advantages of unfair queued spinlocks vs. the simple byte lock are:

1. It encourages a more FIFO like ordering of acquiring the lock thus greatly diminishes the chance of lock starvation that can happen more often with the simple byte lock.
2. It can greatly reduce the amount of lock cacheline access in a contended lock thus reducing cacheline contention and improving performance when there is heavy lock contention.

The main drawback of the unfair queued spinlocks, however, is the increased size and complexity of the locking code.

On a small VM with just a few vCPUs, there isn't much performance difference between simple byte lock and unfair queued spinlocks. It is mainly with larger VMs that unfair queued spinlocks will show a performance advantage.

The para-virtualized queued spinlocks used in KVM and Xen don't perform well in over-committed VMs when total number of vCPUs > the actual number physical CPUs present. The unfair queued spinlocks, on the other hand, can more or less maintain their performance even in over-committed situations.

# Unfair Queued Spinlocks Performance

The table below shows the average run times a micro-benchmark (a 1 million lock-unlock loop) for different number of contending threads running in a 32-vCPU VM on a 8-socket Westmere-EX system.

| Lock Type | Threads | Average Run Time (ms) | Standard Deviation (ms) |
|---|---|---|---|
| Unfair byte lock | 4 | 386 | 153 |
| | 8 | 940 | 117 |
| | 16 | 6,046 | 1,747 |
| | 32 | 37,028 | 10,136 |
| Unfair queued lock | 4 | 454 | 80 |
| | 8 | 1,126 | 305 |
| | 16 | 3,633 | 1,000 |
| | 32 | 7,103 | 1,231 |

# Unfair Queued Spinlocks Statistics Counts

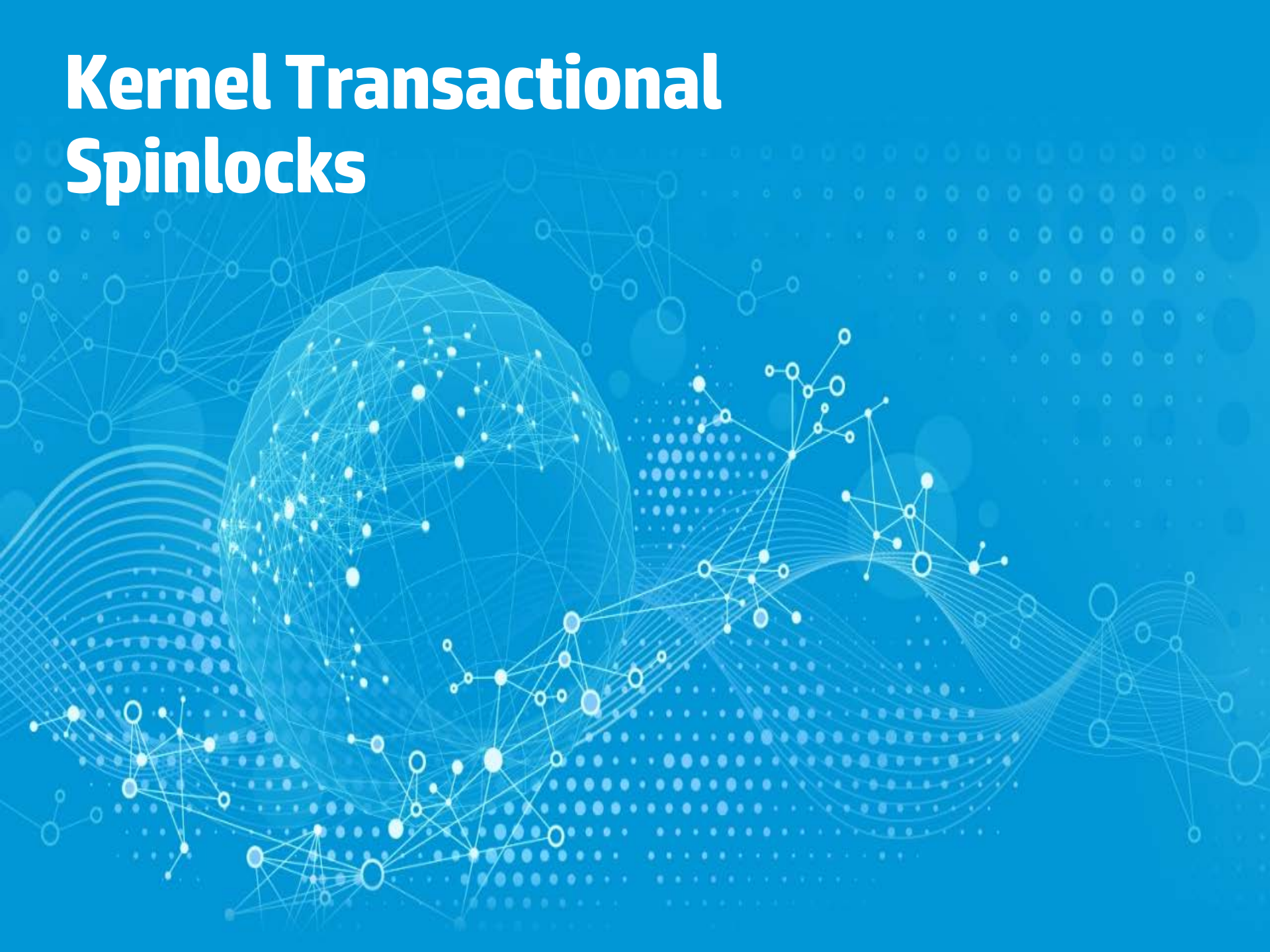
By enabling statistics count logging, the followings are sample counts after the bootup of a 32-vCPU KVM guest:

```
lsteal_cnts = 172219 2377 425 118 33 8 5 12 14 0 0 0
trylock_cnt = 1495372
```

Most of the lock stealing happened in the initial trylock before entering the queue which contributes greatly to the larger standard deviation of the result, but is also required for good performance. Once a vCPU is in the queue, the chance of getting the lock drops off significantly the further it is away from the queue head.



# Kernel Transactional Spinlocks



# Intel's Transactional Synchronization Extensions (TSX)

- Intel's TSX adds hardware transactional memory support, speeding up execution of multi-threaded software through lock elision.
- Two different software interfaces are provided for transactional execution:
  1. Hardware Lock Elision (HLE)
  2. Restricted Transaction Memory (RTM)
- TSX is enabled on Haswell-EX and the latest Broadwell and SkyLake processors.
- Memory addresses (on a cacheline granularity level) read from within a transactional region constitute the read-set of the transactional region. Addresses written to within the transactional region constitute the write-set of the transactional region.
- A transaction can be aborted due to one of the following reasons:
  1. Conflicting cacheline accesses (R-W, W-W)
  2. Conflicts in lock variable
  3. Resource constraints
  4. Interrupts, traps and special instructions (e.g. pause)

# Transactional Abort

- Transaction abort is the major limiter of transactional execution performance.
- There are several ways to handle transaction aborts.
  1. Wait a little while and retry transactional execution again
  2. Actually acquire the lock and execute critical section non-transactionally. This will likely abort all the transactional executions that are in progress if the lock is in the read-set.
  3. Notify the callers for the transactional abort and let them handle the failure. That will break the typical locking interface.



# Kernel Transactional Spinlocks

- Most transactional lock implementation includes the lock in the read-set. This is problematic in handling the transition from transactional to non-transactional executions as it will cause more transaction aborts.
- Above a certain transaction abort percentage threshold, all the locks will have to be acquired non-transactionally.
- A somewhat different way of doing transactional locking is proposed here.
- The transactional execution regions and non-transactional execution regions will be separated by encapsulating them into the read and write critical sections of a traditional rwlock respectively. The rwlock itself will not be in the read or write set of the transaction regions.
- Transactional execution – acquire read lock, enter transaction region with RTM, release read lock
- Non-transactional execution – acquire write lock, non-transaction critical region, release write lock
- Transitioning to a non-transactional execution will not cause an transaction abort in another transaction region.
- The use of qrwlock will ensure that lock starvation will not happen.

# Transactional Spinlock APIs and Limitations

- Locking APIs:
  tx_spin_lock(txlock_t *lock) – Acquire the lock transactionally
  tx_spin_lock_non(txlock_t *lock) – Acquire the lock non-transactionally
  tx_spin_unlock(txlock_t *lock) – Release the lock (transactional or non-transactionally)
  tx_spin_lock_retry(txlock_t *lock, int retry) – Acquire the lock transactionally with the given retry count
- The current implementation does have its limitations:
  1. The rwlock cannot be in the read and write-sets of the transaction region. So it may need to occupy a full cacheline.
  2. The overhead of taking and releasing the read lock will limit performance gain for small critical regions.
  3. On a contended lock, the performance of queued write lock isn't as good as queued spinlock.
  4. Nesting, though supported by RTM, is not currently implemented.

# Proof of Concept

- The inode_sb_list_lock in fs/inode.c was converted into a tx_lock_t.
- The inode_sb_list_del() function which deletes an inode from the superblock i_sb_list linked list is modified to execute transactionally.
- A micro-benchmark that spawns a large number of threads, creates a large number of temporary inodes and exits simultaneously was run.
- At the exiting phase:

```
# perf stat -e "tx-start,tx-commit,tx-abort,tx-conflict" -a sleep 5

 Performance counter stats for 'system wide':

       10,593,300      cpu/tx-start/          [100.00%]
       10,587,579      cpu/tx-commit/         [100.00%]
            5,553      cpu/tx-abort/          [100.00%]
            4,083      cpu/tx-conflict

      5.012774554 seconds time elapsed
```

# Next Step

- The inode_sb_list_lock conversion shown in the previous slide did not produce any performance gain because of the small critical section.
- Looking for suitable use cases that can show performance benefit.



# Thank you

# Q & A

Waiman.Long@hp.com