Locks on Multicore and Multisocket Platforms

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Context

- Last lecture: locks and barriers
- **Lock synchronization on multicore platforms**
- Upcoming
  - transactional memory
  - practical non-blocking concurrent objects
Papers for Today


Motivation for Studying Lock Performance

- There are many types of locks and architectures
- Does lock performance depend on architecture?
- How?
- Which lock is best?
Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock
Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock
Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock

Locks we haven’t discussed
CLH List-based Queue Lock

type qnode = record
  prev : ^qnode
  succ_must_wait : Boolean

type lock = ^qnode  // initialized to point to an unowned qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
  I->succ_must_wait := true
  pred : ^qnode := I->prev := fetch_and_store(L, I)
  repeat while pred->succ_must_wait

procedure release_lock (ref I : ^qnode)
  pred : ^qnode := I->prev
  I->succ_must_wait := false
  I := pred         // take pred's qnode
CLH

has lock; in critical section
CLH Queue Lock Notes

- Discovered twice, independently
  - Travis Craig (University of Washington)
    - TR 93-02-02, February 1993
  - Anders Landin and Eric Hagersten (Swedish Institute of CS)
    - IPPS, 1994

- Space: $2p + 3n$ words of space for $p$ processes and $n$ locks
  - MCS lock requires $2p + n$ words

- Requires a local "queue node" to be passed in as a parameter

- Spins only on local locations on a cache-coherent machine

- Local-only spinning possible when lacking coherent cache
  - can modify implementation to use an extra level of indirection
    (local spinning variant not shown)

- Atomic primitives: fetch_and_store
Why Hierarchical Locks?

NUMA architectures

• Not all memory is equidistant to all cores
  — each socket has its own co-located memory
  — consequence of scaling memory bandwidth with processor count

• Today’s systems: system-wide cache coherence

• Access latency depends on the distance between the core and data location
  — memory or cache in local socket
  — memory or cache in remote socket

• Multiple levels of locality
  — 0 hop, 1 hop, 2 hop, ...
Locks on NUMA Architectures

• Problem:
  — passing locks between threads on different sockets can be costly
  — overhead from passing lock and data it protects
  — data that has been accessed on a remote socket produces long latency cache misses

• Solution:
  — design locks to improve locality of reference
  — encourage threads with mutual locality to acquire a given lock consecutively

• Benefits:
  — reduce migration of locks between NUMA nodes
  — reduce cache misses for data accessed in a critical section
Hierarchical CLH

• Structure
  —local CLH queue per cluster (socket)
  —one global queue
  —qnode at the head of the global queue holds the lock

• Operation: when a node arrives in the local queue ...
  —delay for a bit to let successors arrive
  —move a batch from a socket queue to the global queue
    – CAS local tail into global tail
    – link local head behind previous global tail

Hierarchical CLH in Action

Socket J

Socket K

G

L_J

L_K

has lock; in critical section

spin

spin
Hierarchical CLH in Action

Socket J

Socket K

LJ

LK

G

G has lock; in critical section

spin

spin

pause
Hierarchical CLH in Action

Socket J

Socket K

G

L_J

L_K

has lock; in critical section

spin

pause

spin
Hierarchical CLH in Action

Socket J

Socket K

G

LJ

LK

has lock; in critical section

spin

spin

pause

spin

pause
Hierarchical CLH in Action

Socket J

Socket K

has lock; in critical section
Hierarchical CLH in Action

G

Socket J

L_J

Socket K

L_K

has lock; in critical section

linking spin spin

pause spin
Hierarchical CLH in Action

Socket J

Socket K

LJ

LK

G

has lock; in critical section

spin

spin

linking

spin

spin

pause

spin
Hierarchical CLH in Action

Socket J

Socket K

L_J

L_K

has lock; in critical section

spin

spin

spin

spin

pause

spin
Hierarchical CLH in Action

Socket J

Socket K

has lock; in critical section

spin

spin

spin

spin

spin

L_J

L_K

linking

spin
Hierarchical CLH in Action

- Socket J
  - L_J
  - Pause

- Socket K
  - L_K
  - Linking
  - Spin

G

Spin
Spin
Spin
Spin
Spin
Spin

Has lock; in critical section
Hierarchical CLH in Action

Socket J

Socket K

pause

has lock; in critical section

spin

spin

spin

spin

spin

linking

spin

G

L_J

L_K
Hierarchical CLH in Action

Has lock; in critical section

Socket J

Socket K

Pause
Hierarchical Ticket

• Two levels of ticket locks
  — global
  — local: one per socket

• Two-level ticket lock (cohorting version by Dice et al.)
  — acquire
    – acquire local ticket
    – if flag “global granted” is set, proceed
    – else acquire global ticket lock
  — release
    – if successors available in local lock, set “global granted” for local lock and increment local ticket
    – otherwise, clear “global granted” for local lock and increment global ticket

• “Everything…” paper used a more complex version
  — https://github.com/tudordavid/libslock/blob/master/src/htlock.c
Hierarchical Backoff Lock

- Test-and-test-and-set lock with back off scheme to reduce cross node contention of a lock variable
- Use thread locality to tune backoff delay
  - when acquiring a lock
    - assign thread ID to lock state
  - when spin waiting
    - compare thread ID with lock holder and back off proportionally
- Limitations:
  - reduce lock migration only probabilistically
  - lots of invalidation traffic: costly for NUMA

Systems with Different Characteristics

- **Opteron**: 4 x AMD Opteron 6172 (48 cores)
  - directory based cache coherence
  - directory located in LLC

- **Xeon**: 8 x Intel Xeon E7-8867L (80 cores; SMT disabled)
  - broadcast snooping

- **Niagara**: SUN UltraSPARC-T2 (8 cores; 64 threads)
  - coherence via shared L2 cache on far side of chip

- **Tilera**: TILE-Gx CPU (36 cores)
  - coherence via distributed, shared L2 cache
Opteron Platform

- Opteron: 4 x AMD Opteron 6172 (48 cores)
- Each chip contains two 6-core dies
- MOESI protocol, directory based cache coherence
  —directory located in LLC
- Average distance: 1.25 hops

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP ’13. ACM, New York, NY, USA, 33-48.
Xeon Platform

- Xeon: 8 x Intel Xeon E7-8867L (80 cores; SMT disabled)
  - broadcast snooping
- 10 cores per socket
- Average distance: 1.375 hops

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP ’13. ACM, New York, NY, USA, 33-48.
Niagara

- Niagara: SUN UltraSPARC-T2 (8 cores; 64 threads)
  - coherence via shared L2 cache on far side of chip

Figure credit: Niagara: A 32-way Multithreaded SPARC Processor; P. Kongetira, K. Aingaran, K. Olukotun
Tilera

- Tilera: TILE-Gx CPU (36 cores)
  - coherence via distributed, shared L2 cache

Figure credit: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf
Latencies depend upon distance and (sometimes) state

<table>
<thead>
<tr>
<th>System</th>
<th>Opteron (2.1 GHz)</th>
<th>Xeon (2.13 GHz)</th>
<th>Niagara (1.2 GHz)</th>
<th>Tilera (1.2 GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>same die</td>
<td>same MCM</td>
<td>one hop</td>
<td>two hops</td>
</tr>
<tr>
<td>Modified</td>
<td>81</td>
<td>161</td>
<td>172</td>
<td>252</td>
</tr>
<tr>
<td>Owned</td>
<td>83</td>
<td>163</td>
<td>175</td>
<td>254</td>
</tr>
<tr>
<td>Exclusive</td>
<td>83</td>
<td>163</td>
<td>175</td>
<td>253</td>
</tr>
<tr>
<td>Shared</td>
<td>83</td>
<td>164</td>
<td>176</td>
<td>254</td>
</tr>
<tr>
<td>Invalid</td>
<td>136</td>
<td>237</td>
<td>247</td>
<td>327</td>
</tr>
</tbody>
</table>

Opteron: load latency independent of state

Xeon: load latency depends on state

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP ’13. ACM, New York, NY, USA, 33-48.
Variation in Performance of Atomics

Observations

—relative performance of atomic primitives and cache operations varies widely in the hardware

—varying performance of locks is in part due to varying performance of atomic operations
Observations

—throughput on multi-socket systems is lower than on single chips
—there is no universally best lock
Figure 6: Uncontested lock acquisition latency based on the location of the previous owner of the lock.

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.
Impact of Contention on Performance

**Figure 7:** Throughput of different lock algorithms using 512 locks.

**Figure 5:** Throughput of different lock algorithms using a single lock.
Study Conclusions

• Crossing sockets is expensive
  — 2x to 7.5x slower than intra-socket
  — hard to avoid cross-socket communication
    — e.g., Opteron: incomplete cache directory (no sharer info)

• Loads, stores can be as expensive as atomic operations
  — non-local access can be a bottleneck

• Intra-socket non-uniformity matters (e.g., Tilera vs. Niagara)
  — hierarchical locks scale better on non-uniform systems

• Simple locks can be effective
  — ticket lock performs best in many cases

• There’s no universally optimal lock
  — optimal lock depends upon architecture and expected contention
Simple locks are powerful. Overall, an efficient implementation of a ticket lock is the best performing synchronization scheme in most low contention workloads. Even under rather high contention, the ticket lock performs comparably to more complex locks, in particular within a socket. Consequently, given their small memory footprint, ticket locks should be preferred, unless it is sure that a specific lock will be very highly contended.
Locks in Linux
Non-scalable Locks are Dangerous

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

Non-scalable locks are dangerous Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.*
### Linux Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Operation time (cycles)</th>
<th>Top lock instance name</th>
<th>Acquires per operation</th>
<th>Average critical section time (cycles)</th>
<th>% of operation in critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOPS</td>
<td>503</td>
<td>d_entry</td>
<td>4</td>
<td>92</td>
<td>73%</td>
</tr>
<tr>
<td>MEMPOPOP</td>
<td>6852</td>
<td>anon_vma</td>
<td>4</td>
<td>121</td>
<td>7%</td>
</tr>
<tr>
<td>PFIND</td>
<td>2099 M</td>
<td>address_space</td>
<td>70 K</td>
<td>350</td>
<td>7%</td>
</tr>
<tr>
<td>EXIM</td>
<td>1156 K</td>
<td>anon_vma</td>
<td>58</td>
<td>165</td>
<td>0.8%</td>
</tr>
</tbody>
</table>

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

FOPS creates a single file and starts one process on each core. Each thread repeatedly opens and closes the file.

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.

MEMPOPOP creates one process per core. Each process repeatedly mmaps 64 kB of memory with the MAP_POPULATE flag, then munmaps 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.

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].

Background

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

* The workload was run with ramfs.
Why is Scaling Poor?

Analysis (1-2)

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

<table>
<thead>
<tr>
<th>40 Users (4000 jobs)</th>
<th>400 users (40,000 jobs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 9.40% reaim reaim</td>
<td>- 79.53% reaim [kernel.kallsyms] [k] _spin_lock</td>
</tr>
<tr>
<td>+ 6.07% reaim libc-2.12.so</td>
<td>+ _spin_lock</td>
</tr>
<tr>
<td>- 1.68% reaim [kernel.kallsyms] [k] _spin_lock</td>
<td></td>
</tr>
<tr>
<td>- _spin_lock</td>
<td>+ 34.28% file_move</td>
</tr>
<tr>
<td>+ 50.36% lookup_mnt</td>
<td>+ 34.20% file_kill</td>
</tr>
<tr>
<td>+ 7.45% __d_lookup</td>
<td>+ 19.94% lookup_mnt</td>
</tr>
<tr>
<td>+ 6.71% file_move</td>
<td>+ 8.13% reaim [kernel.kallsyms] [k] mutex_spin_on_owner</td>
</tr>
<tr>
<td>+ 5.16% file_kill</td>
<td>+ 0.86% reaim [kernel.kallsyms] [k] _spin_lock_irqsave</td>
</tr>
<tr>
<td>+ 2.46% handle_pte_fault</td>
<td>+ 0.63% reaim reaim</td>
</tr>
</tbody>
</table>

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%

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!
Why is Scaling Poor?

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)

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

- 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():

```c
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!
Lock Cohortting
Lock Cohorting

- Idea: use two levels of locks
  - global locks
  - local locks, one for each socket or cluster (NUMA node)
- First in socket to acquire local lock
  - acquire socket lock then the global lock
  - pass local lock to other waiters in the local node
  - eventually relinquish global lock to give other nodes a chance
- Recipe for NUMA-aware locks without special algorithms
- Cohorting can compose any kind of lock into a NUMA lock
  - augments properties of cohorted locks with locality preservation
- Benefits
  - reduces average overhead of lock acquisition
  - reduces interconnect traffic for lock and protected data
Global and Local Locks for Cohorting

• Global lock G
  —thread-oblivious: acquiring thread can differ from releasing thread
  —globally available to all nodes of the system

• Local lock S
  —supports cohort detection
    - a releasing thread can detect if other threads waiting
  —records last state of release as global or local

• Once S is acquired
  —local release → proceed to critical section
  —global release → try to acquire G

• Upon release of S
  —if NOT (may_pass_local OR alone) → release globally
  —else → release locally
Global and Local Locks for Cohorting

- C-BO-BO lock
  - Global backoff (BO) lock and local backoff locks per node
  - Requires additional cohort detection mechanism in local BO lock
- C-TKT-TKT lock
  - Global ticket lock and local ticket (TKT) locks per node
- C-BO-MCS lock
  - Global backoff lock and local MCS lock
- C-MCS-MCS lock
- C-TKT-MCS lock

Use of abortable locks in cohort designs needs extra features to limit aborting while in a cohort
  - A-C-BO-BO lock
  - A-C-BO-CLH lock (queue lock of Craig, Landin, & Hagersten)
Experiments

• Microbenchmark LBench is used as a representative workload

• LBench launches identical threads

• Each thread loops as follows
  — acquire central lock
  — access shared data in critical section
  — release lock
  — ~4ms of non-critical work

• Run on Oracle T5440 series machine
  — 256 hardware threads
  — 4 NUMA clusters

• Evaluation shows that cohort locks outperform previous locks by at least 60%
Average Throughput vs. # of Threads

Throughput/sec vs. # of Threads for various thread configurations:
- MCS
- HBO
- HCLH
- FC-MCS
- C-BO-BO
- C-TKT-TKT
- C-BO-MCS
- C-TKT-MCS
- C-MCS-MCS

The graph shows how the throughput changes as the number of threads increases, with different thread configurations exhibiting varying performance characteristics.
Conclusions: Cohorting is Useful

- Useful design methodology
  - no special locks required
  - can be extended to additional levels of locality
    - e.g., tile based systems where locality is based on grid position
    - multiple levels of lock cohorts

- Cohort locks improve performance over previous NUMA aware lock designs

- Performance scaling with thread count is better with locality-preserving cohort locks
New Work: Adaptive HMCS Lock

- Tree of MCS locks to exploit multiple levels of locality
- Fast path: directly acquire root if lock is available
- Hysteresis: adaptively select at which level to compete
Performance of AHMCS on Power 4-Socket

Throughput: Higher is better

(a) 1 cache line
(b) 2 cache lines
(c) 4 cache lines
(d) 64 cache lines

Throughput: Higher is better
Figure 6. Total throughput for the key-value map microbenchmark.
Throughput: Higher is better
Dice and Kogan’s CNA Lock in Linux

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Contended spin locks</th>
<th>Call sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock1_threads</td>
<td>files_struct.file_lock</td>
<td>__alloc_fd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fcntl1_setlk</td>
</tr>
<tr>
<td>lock2_threads</td>
<td>file_lock_context.flc_lock</td>
<td>posix_lock_inode</td>
</tr>
<tr>
<td>open1_threads</td>
<td>files_struct.file_lock</td>
<td>__alloc_fd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__close_fd</td>
</tr>
<tr>
<td></td>
<td>lockref.lock</td>
<td>dput</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d_alloc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lockref_get_not_zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lockref_get_not_dead</td>
</tr>
<tr>
<td>open2_threads</td>
<td>files_struct.file_lock</td>
<td>__alloc_fd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>__close_fd</td>
</tr>
</tbody>
</table>

**Table 1.** Contention in the will-it-scale benchmarks.

Throughput: Higher is better
Dice and Kogan’s CNA Lock in Linux

![Graphs showing performance results for different benchmarks.](image)

**Figure 15.** Performance results for the will-it-scale benchmarks.

Throughput: Higher is better