Scalable Locking

Adam Belay <abelay@mit.edu>
Problem: Locks can ruin performance

Figure 2: Sudden performance collapse with ticket locks.

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

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

Locking overhead dominates
Problem: Locks can ruin performance

• Locks themselves can prevent us from harnessing multi-core to improve performance
• Why it happens is interesting and worth understanding
• The solutions are clever exercises in parallel programming
Locking bottleneck is caused by interaction with multicore caching
Recall picture from last time

CPU 0

CPU 1

CPU 2

CPU 3

RAM

bus

LOCK (e.g. XCHG)
Not how multicores actually work

• RAM is much slower than processor, need to cache regions of RAM to compensate

• **Cache Consistency:** *order* of reads and writes between memory locations

• **Cache Coherence:** data *movement* caused by reads and writes for a single memory location
Imagine this picture instead...

- CPU 0
- CPU 1
- CPU 2
- CPU 3

Cache

RAM

bus

LOCK (e.g. XCHG)
How does cache coherence work?

• Many different schemes are possible
• Here’s a simple but relevant one:
  • Divide cache into fixed-sized chunks called “cache-lines”
  • Each cache-line is 64 bytes and in one of three states:
    (M)odified, (S)hared, or (I)nvalid
• Cores exchange messages as they read and write
  • invalidate(addr): delete from your cache
  • find(addr): does any core have a copy?
  • all messages are broadcast to all cores
MSI state transitions

Invalid:
• On CPU read -> find, Shared
• On CPU write -> find, invalidate, Modified
• On message find, invalidate -> do nothing
MSI state transitions

**Shared:**
- On CPU read, find -> do nothing
- On CPU write -> invalidate, **Modified**
- On message invalidate -> **Invalid**
- On message find -> do nothing
MSI state transitions

Modified:

• On CPU read and write --> do nothing
• On message find --> Shared
• On message invalidate --> Invalid
Compatibility of states between cores

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>S</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>I</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Invariants:

- At most one core can be in M state
- Either one M or many S, never both
What access patterns work well?
What access patterns work well?

• Read only data: S allows every core to keep a copy
• Data written multiple times by a core: M gives exclusive access, reads and writes are free basically after first state transition
Real caches are hierarchical

Latency
- < 1 cycle: Core, Core (128 bytes)
- 3 cycles: L1D, L1C, L1D, L1C (2x32 KB)
- 11 cycles: L2, L2 (256 KB)
- 44 cycles: LLC (20 MB)
- 355 cycles: RAM (Many GBs)

Capacity
- < 1 cycle: 128 bytes
- 3 cycles: 2x32 KB
- 11 cycles: 256 KB
- 44 cycles: 20 MB
- 355 cycles: Many GBs
Still a simplification

• Real CPUs use complexer state machines (e.g. MOESI)
  • Why? Fewer bus messages, no broadcasting, etc.

• Real CPUs have complex interconnects
  • Buses are broadcast domains, can’t scale
  • On-chip network for communication within die
    • Data sent in special packets called “flits”
  • Off-chip network for communication between dies
    • E.g. Intel QPI (Quick-Path Interconnect)

• Real CPUs have “Cache Directories”
  • Central structure that tracks which CPUs have copies of data

• Consider taking 6.823!
Why locks if we have cache coherence?
Why locks if we have cache coherence?

- **cache coherence** ensures that cores read fresh data
- Locks avoid lost updates in read-modify-write cycles and prevent anyone from seeing partially updated data structures
- Locks are a tool to control **cache consistency** --- the order of reads and writes
Locks are built from atomic instructions

- So far we use XCHG in xv6 (via AMOSWAP)
- Many other atomic ops, including add, test-and-set, CAS, etc.
- How does hardware implement locks?
  - Get the line in **M** state
  - Defer coherence messages
  - Do all the steps (read and write)
  - Resume handling messages
Locking performance criteria

• Assume N cores are waiting for a lock
• How long does it take to hand off from previous to next holder?
• Bottleneck is usually the interconnect
  • So measure cost in terms of # of messages

• What can we hope for?
  • If N cores waiting, get through them all in O(N) time
  • Each handoff takes O(1) time; does not increase with N
Test & set (TAS) spinlocks

struct lock { int locked; };

acquire(l){
    while(1){
        if(!xchg(&l->locked, 1))
            break;
    }
}

Release(l){
    l->locked = 0;
}
Test & set (TAS) spinlocks

• Spinning cores repeatedly execute atomic exchange

• Is this a problem?
  • Yes!
    • It’s okay if waiting cores waste their own time
    • But bad if waiting cores slow lock holder!

• Time for critical section and release:
  • Holder must wait in line for access to bus
  • So holder’s handoff takes O(N) time

• O(N) handoff means all N cores take O(N^2)!
Ticket locks (Linux)

- Goal: read-only spinning rather than repeated atomic instructions
- Goal: fairness -> waiter order preserved
- Key idea: assign numbers, wake up one waiter at a time
Ticket locks (Linux)

```c
struct lock {
    uint16 current_ticket; uint16 next_ticket;
}

acquire(l) {
    int t = atomic_fetch_and_inc(&l->next_ticket);
    while (t != l->current_ticket); /* spin */
}

void release(l) {
    l->current_ticket++;
}
```
Ticket lock time analysis

- Atomic increment – $O(1)$ broadcast message
  - Just once, not repeated
- Then read-only spin, no cost until next release
- What about release?
  - Invalidate message sent to all cores $O(1)$
  - Then $O(N)$ find messages, as they re-read
- Oops, still $O(N)$ handoff work!
- But fairness and less bus traffic while spinning
TAS and Ticket are “non-scalable” locks

Cost of handoff scales with number of waiters
Let’s reconsider figure 2 (PFIND)

![Graph showing sudden performance collapse with ticket locks.](image)

Figure 2: Sudden performance collapse with ticket locks.
Reasons for collapse

• Critical section takes just 7% of request time
  • So with 14 cores, you’d expect just one core wasted by serial execution

• So it’s odd that the collapse happens so soon

• However, once cores waiting for unlock is substantial, critical section + handoff takes longer

• Slower handoff time makes N grow even further
Perspective

Consider:

```c
acquire(&l); x++; release(&l);
```

- uncontended: ~40 cycles
- if a different core used the lock last: ~100 cycles
- With dozens of cores: thousands of cycles
So how can we make locks scale?

• Goal: $O(1)$ message release time
• Can we wake just one core at a time?
• Idea: Have each core spin on a different cache-line
MCS Locks

• Each CPU has a qnode structure in its local memory

typedef struct qnode {
    struct qnode *next;
    bool locked;
} qnode;

• A lock is a qnode pointer to the tail of the list
• While waiting, spin on local locked flag
MCS locks

Owner → Waiter → Waiter → NULL

*L
Acquiring MCS locks

```c
acquire (qnode *L, qnode *I) {
    I->next = NULL;
    qnode *predecessor = I;
    XCHG (*L, predecessor);
    if (predecessor != NULL) {
        I->locked = true;
        predecessor->next = I;
        while (I->locked) ;
    }
}
```
Releasing MCS locks

release (lock *L, qnode *I) {
    if (!I->next)
        if (CAS (*L, I, NULL))
            return;
    while (!I->next) ;
    I->next->locked = false;
}
Locking strategy comparison

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

CLH lock. The CLH lock 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 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 x86 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 x86 multicore machine, it does not matter much which scalable lock to choose.

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).
But not a panacea

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` originally 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
But not a panacea

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` originally 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

- Actual Goal

![Graph showing performance comparison between ticket locks and MCS locks]
Conclusion

• Scalability is limited by length of critical section
• Scalable locks can only avoid collapse
• Better plan: Use algorithms that avoid contention all together
• Example in next lecture!