6.828 2014 Lecture 17: Scalable Locks

Plan: cost of spinlocks -- non-scalable effect on real systems scalable locks

Why this topic? Figure 2 in the paper -- disaster! (details later) the locks themselves are ruining performance rather than letting us harness multi-core to improve performance this "non-scalable lock" phenomenon is important why it happens is interesting and worth understanding the solutions are clever exercises in parallel programming

the problem is interaction of locks w/ multi-core caching so let's look at the details

back in the locking lecture, we had a fairly simple model of multiple cores cores, shared bus, RAM to implement acquire, x86's xchg instruction locked the bus provided atomicity for xchg

real computers are much more complex bus, RAM quite slow compared to core speed per-core cache to compensate hit: a few cycles RAM: 100s of cycles

how to ensure caches aren't stale? core 1 reads+caches x=10, core 2 writes x=11, core 1 reads x=?

answer: "cache coherence protocol" ensures that each read sees the latest write actually more subtle; look up "sequential consistency"

how does cache coherence work? many schemes, here's a simple one each cache line: state, address, 64 bytes of data states: Modified, Shared, Invalid [MSI] cores exchange messages as they read and write

messages (much simplified) invalidate(addr): delete from your cache find(addr): does any core have a copy? all msgs are broadcast to all cores

how do the cores coordinate with each other? I + local read -> find, S I + local write -> find, inval, M S + local read -> S S + local write -> inval, M S + recv inval -> I S + recv find -> nothing, S M + recv inval -> I M + recv find -> reply, S

can read w/o bus traffic if already S can write w/o bus traffic if already M "write-back"

compatibility of states between 2 cores:

```
    core1

    M S I

  M - - +

core2  S - + +

  I + + +
```

invariant: for each line, at most one core in M invariant: for each line, either one M or many S, never both

Q: what patterns of use benefit from this coherence scheme? read-only data (every cache can have a copy) data written multiple times by one core (M gives exclusive use, cheap writes)

other plans are possible e.g. writes update copies rather than invalidating but "write-invalidate" seems generally the best

Real hardware uses much more clever schemes mesh of links instead of bus; unicast instead of broadcast "interconnect" distributed directory to track which cores cache each line unicast find to directory

Q: why do we need 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

people build locks from h/w-supported atomic instructions xv6 uses atomic exchange other locks use test-and-set, atomic increment, &c the *_sync*... functions in the handout turn into atomic instructions

how does the hardware implement atomic instructions? get the line in M mode defer coherence msgs do all the steps (e.g. read old value, write new value) resume processing msgs

what is performance of locks? assume N cores are waiting for the lock how long does it take to hand off the lock? from previous holder to next holder bottleneck is usually the interconnect so we'll measure cost in terms of # of msgs

what performance could we hope for? if N cores waiting, get through them all in O(N) time so each critical section and handoff takes O(1) time i.e. does not increase with N

test&set spinlock (xv6/jos) waiting cores repeatedly execute e.g. atomic exchange Q: is that a problem? yes! we don't care if waiting cores waste their own time we do care if waiting cores slow lock holder! time for critical section and release: holder must wait in line for access to bus so holder's mem ops take O(N) time so handoff time takes O(N)

Q: is O(N) handoff time a problem? yes! we wanted O(1) time O(N) per handoff means all N cores takes O(N^2) time, not O(N)

ticket locks (linux): goal: read-only spin loop, rather than repeated atomic instruction goal: fairness (turns out t-s locks aren't fair) idea: assign numbers, wake up one at a time avoid constant t-s atomic instructions by waiters Q: why is it cheaper than t-s lock? Q: why is it fair? time analysis: what happens in acquire? atomic increment -- O(1) broadcast msg just once, not repeated then read-only spin, no cost until next release what happens after release? invalidate msg for now_serving N "find" msgs for each core to read now_serving so handoff has cost O(N) note: it was *reading* that was costly! oops, just as bad O() cost as test-and-set

jargon: test-and-set and ticket locks are "non-scalable" locks == cost of single handoff increases with N

is the cost of non-scalable locks a serious problem? after all, programs do lots of other things than locking maybe locking cost is tiny compared to other stuff

see paper's Figure 2 let's consider Figure 2(c), PFIND -- parallel find x-axis is # of cores, y-axis is finds completed per second (total throughput) why does it go up? why does it level off? why does it go *down*? what governs how far up it goes -- i.e. the max throughput? why does it go down so steeply?

reason for suddenness of collapse serial section takes 7% on one core (Figure 3, last column) so w/ 14 cores you'd expect just one or two in crit section so it seems odd that collapse happens so soon BUT: once P(two cores waiting for lock) is substantial, critical section + handoff starts taking longer so starts to be more than 7% so more cores end up waiting so N grows, and thus handoff time, and thus N...

some perspective acquire(l) x++ release(l) surely a critical section this short cannot affect overall performance? takes a few dozen cycles if same core last held the lock (still in M) everything operates out of the cache, very fast a hundred if lock not held, some other core previously held 10,000 if contended by dozens of cores many kernel operations only take a few 100 cycles total so a contended lock may increase cost not by a few percent but by 100x!

how to make locks scale well? we want just O(1) msgs during a release how to cause only one core to read/write lock after a release? how to wake up just one core at a time?

test-and-set with exponential backoff (t_s_exp_acquire): goal: avoid everyone jumping in at once space out attempts to acquire lock simultaneous attempts were reason for O(N) release time w/ t-s if total rate of tries is low, only one core will attempt per release why not constant delay? each core re-tries after random delay with constant average hard to choose delay time too large: waste too small: all N cores probe mult times/crit, so O(N) release time why exponential backoff? i.e. why start with small delay, double it? try to get lucky at first (maybe only a few cores attempting)

doubling means takes only a few attempts until delay >= N *crit section i.e. just one attempt per release illustration: eventually will be roughly one probe per critical section time then all will complete in that backoff round can we analyze # of probes? not that easy suppose takes time O(N) for all cores to succeed how many probes does each core make in time N? logN so total probes: N*logN so cost per release: O(logN) not O(1), but much better than O(N) problem: unlikely to be fair! some cores will have much lower delays than others will win, and come back, and win again some cores will have huge delays, will sit idle doing no harm, but doing no work

anderson: goal: O(1) release time, and fair what if each core spins on a *different* cache line? acquire cost? atomic increment, then read-only spin release cost? invalidate next holder's slots[] only they have to re-load no other cores involved so O(1) per release -- victory! problem: high space cost N slots per lock often much more than size of protected object

MCS [just diagram, no code] goal: as scalable as anderson, but less space used idea: linked list of waiters per lock idea: one list element per thread, since a thread can wait on only one lock so total space is O(locks + threads), not anderson't O(locks*threads) acquire() pushes caller's element at end of list caller then spins on a variable in its own element release() wakes up next element, pops its own element change in API (need to pass qnode to acquire and release to qnode allocation)

performance of scalable locks? figure 10 shows ticket, MCS, and optimized backoff

cores on x-axis, total throughput on y-axis

benchmark acquires and releases, critical section dirties four cache lines Q: why doesn't throughput go up as you add more cores? ticket is best on two cores -- just one atomic instruction ticket scales badly: cost goes up with more cores MCS scales well: cost stays the same with more cores

Figure 11 shows uncontended cost very fast if no contention! ticket: acquire uses a single atomic instruction, so 10s more expensive than release some what more expensive if so emother core had it last

Concl. use scalable locks even better: fix the underlying problem!

Edit By MaHua