Advanced Computer Architecture

Chapter 10 – Multicore, parallel, and cache coherency

Part 3:

Atomic operations, concurrency control primitives, and memory consistency models

November 2025

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (3rd, 4th and 5th eds),* and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

What you should get from this

Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- Why power considerations motivate multicore
- Why is shared-memory parallel programming attractive?
  - How is dynamic load-balancing implemented?
  - Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- What is the cache coherency problem
  - There is a design-space of "snooping" protocols based on broadcasting invalidations and requests
- How are atomic operations and locks implemented?
  - Eg load-linked, store conditional
- What is sequential consistency?
  - Why might you prefer a memory model with weaker consistency?
- For larger systems, some kind of "directory" is needed to avoid/reduce the broadcasting

Part 3

Synchronization and atomic operations

- Why Synchronize?
  - **We need to know when it is safe for different processes to use shared data**
- Issues for Synchronization:
  - We need some kind of uninterruptable primitive to fetch and update memory (*atomic* operation)
  - We can build user level synchronization operations using this primitive (lock/unlock, barrier, fetch-and-add, etc)
  - Synchronization can be a bottleneck – we need:
    - Fast non-contended path
    - Efficient in the high-contention case
    - fair

Hennessy and Patterson 6th ed section 5.5 pp412

Uninterruptable operations to Fetch from and Update Memory

Historically there have been several different atomic primitives directly implemented in hardware - eg

- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free
- Atomic exchange: interchange a value in a register for a value in memory

  For example you could use atomic exchange to implement a lock:

  0 => synchronization variable is free

  1 => synchronization variable is locked and unavailable

  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
  - Key is that exchange operation is indivisible

Uninterruptable operations to Fetch from and Update Memory

- Test-and-set
- Fetch-and-increment
- Atomic exchange

These operations all consist of a load *and* a store, that must be executed indivisibly

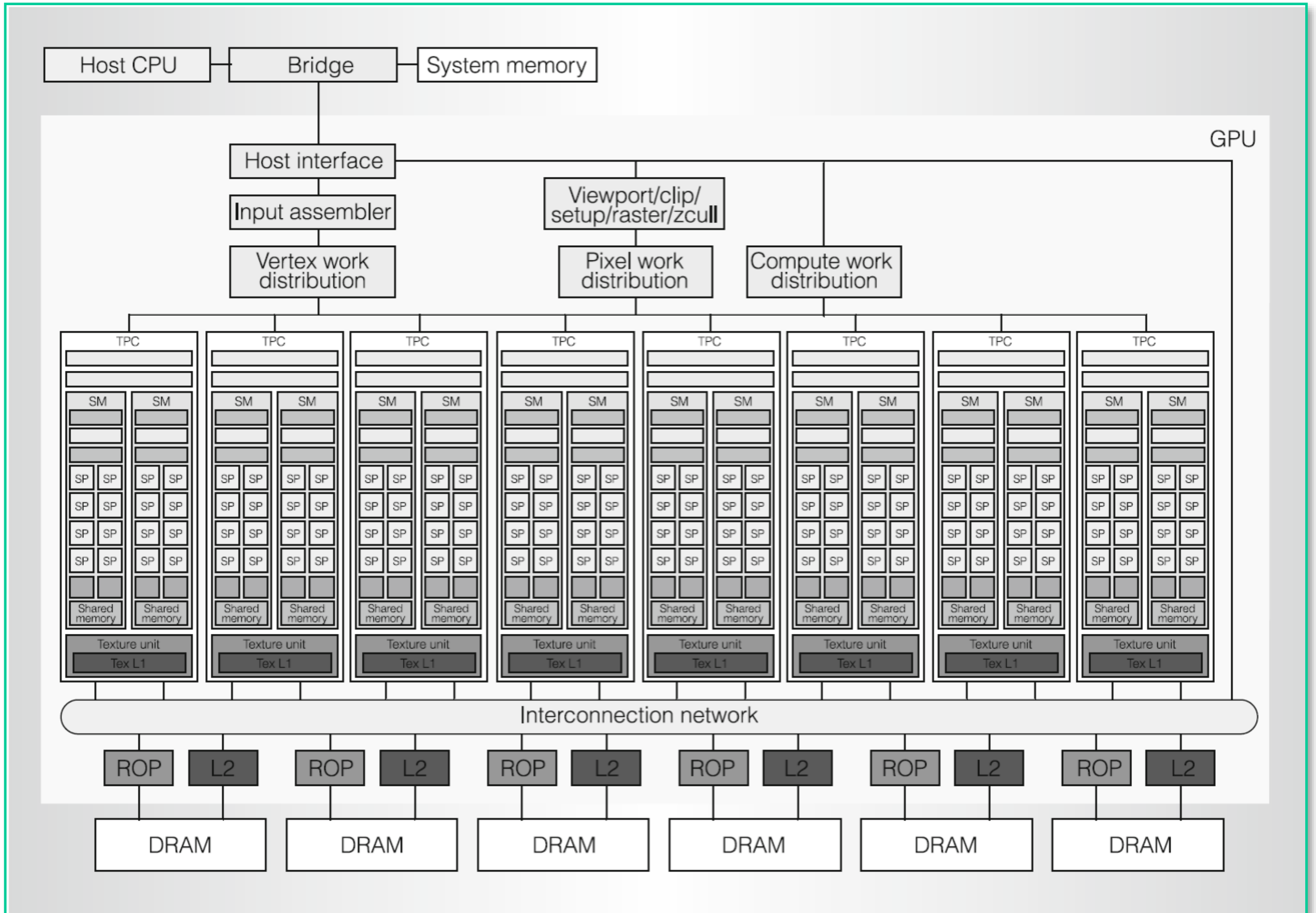
This is plausible in a single-core machine

This is plausible if implemented *in the memory*

- *Eg in a GPU*

- But how can we do this efficiently in a multicore processor with a cache coherency protocol?

Atomics in GPUs



- GPUs generally have no cache coherency protocol for the L1 caches
- So atomic operations on global memory have to be handled in the L2 cache controllers

Accelerating Atomic Operations on GPGPUs Sean Franey and Mikko Lipasti
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1081.2165&rep=rep1&type=pdf

Understanding and Using Atomic Memory Operations Lars Nyland & Stephen Jones, NVIDIA GTC 2013
https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf

How can we implement an uninterruptable instruction to Fetch and update memory in a cache-coherent multicore?

- Hard to have read & write in one instruction - so use two instead
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds
    - Succeeds if there has been no other store to the same memory location since the preceding load) and 0 otherwise
    - Ie if no invalidation has been received
- Example: using LL/SC to do atomic exchange:

EXCH

```
try:  mov   R3,R4      ; mov exchange value
      ll    R2,0(R1)   ; load linked
      sc    R3,0(R1)   ; store conditional
      beqz  R3,try     ; branch store fails (R3 = 0)
      mov   R4,R2      ; put load value in R4
```

- Example: fetch & increment:

Fetch-and-inc

```
try:  ll    R2,0(R1)   ; load linked
      addi  R2,R2,#1   ; increment (OK if reg–reg)
      sc    R2,0(R1)   ; store conditional
      beqz  R2,try     ; branch store fails (R2 = 0)
```

LL and SC are used on RISCV, Alpha, ARM, MIPS, PowerPC
Eg see https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf pg 48

Implementation:

Check that no invalidation for the target line has been received

This idea generalises to …transactions...

User level synchronization operations using exchange

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
lockit:    li      R2,#1
           EXCH    R2,0(R1)        ;atomic exchange
           bnez    R2,lockit       ;already locked?
```

- What about in a multicore processor with cache coherency?
  - Want to spin on a cache copy to avoid keeping the memory busy
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:       li      R2,#1
lockit:    lw      R3,0(R1)        ;load var
           bnez    R3,lockit       ;not free=>spin
           EXCH    R2,0(R1)        ;atomic exchange
           bnez    R2,try          ;already locked?
```

- What happens when a lock is released when many cores are spinning on the lock?
- How much data moves? Who wins?

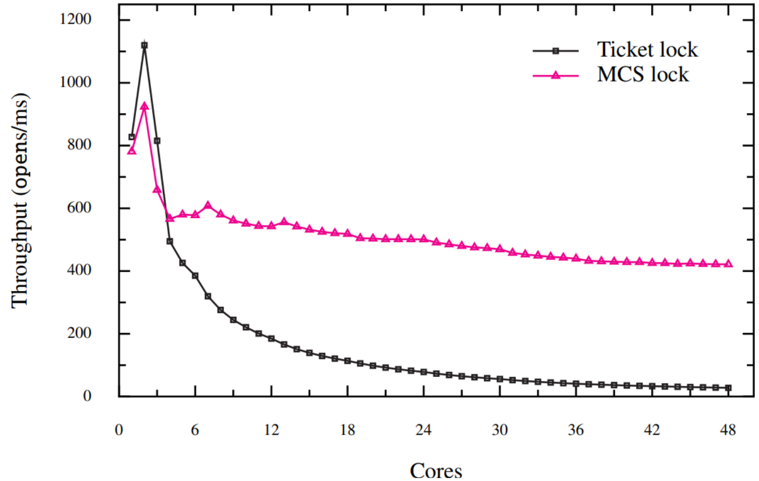
Fairness: ticket locks

```
ticketLock_init(int *next_ticket, int *now_serving)
{
    *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
    my_ticket = fetch_and_inc(next_ticket);
    while (*now_serving != my_ticket) {} // Spin
}

ticketLock_release(int *now_serving)
{
    ++*now_serving;
}
```

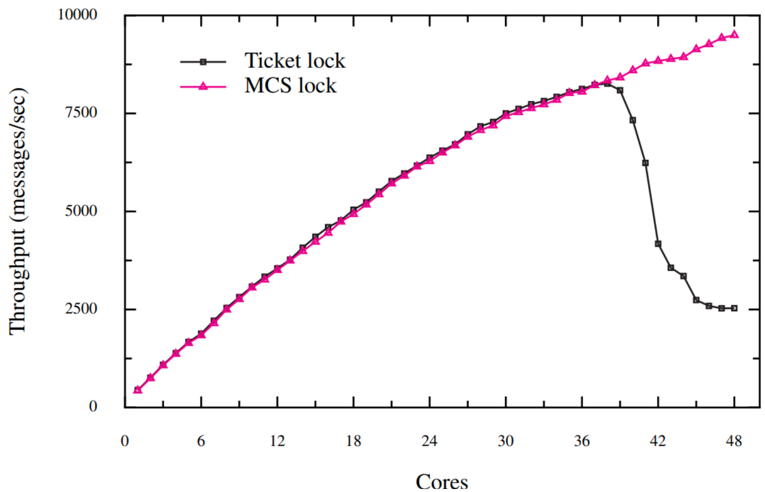
- Ticket lock: explicitly hand off access to the next in line
- Since the my_ticket values are acquired in the order of thread arrival at the lock, subsequent acquisition of the lock is guaranteed to also be in this same order. Thus, fairness of lock acquisition is ensured, enforcing a FIFO ordering.

https://en.wikipedia.org/wiki/Ticket_lock
https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf

Lock behaviour with high core counts



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



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.

"A scalable lock is one that generates a constant number of cache misses per acquisition and therefore avoids the collapse that non-scalable locks exhibit. All of these locks maintain a queue of waiters and each waiter spins on its own queue entry."

For example:

- **MCS lock maintains an explicit queue of qnode structures**
- **A core acquiring the lock adds itself with an atomic instruction to the end of the list of waiters by having the lock point to its qnode,**
- **and then sets the next pointer of the qnode of its predecessor to point to its qnode**
- **If the core is not at the head of the queue, then it spins on its qnode.**

- Ticket locks are better but still behave really badly in bad cases. For better answers, see:
  - Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris and Nickolai Zeldovich, "Non-Scalable Locks are Dangerous", in Proceedings of Linux Symposium (OLS2012):121-132. https://people.csail.mit.edu/nickolai/papers/boyd-wickizer-locks.pdf

Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g. consider:

Thread 1

```
P1:  A = 0;
     .....
     A = 1;
L1:  if (B == 0) ...
```

Thread 2

```
P2:     B = 0;
        .....
        B = 1;
L2:     if (A == 0) ...
```

Hennessy and Patterson 6th ed section 5.6 pp417

- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Different processor families implement different *memory consistency models*
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

Memory Consistency Models

- Weak consistency can be faster than sequential consistency
- Several processors provide fence instructions to enforce sequential consistency when an instruction stream passes such a point. Expensive!
- Not really an issue for most programs if they are *explicitly synchronised*
  - A program is synchronised if all access to shared data are ordered by synchronisation operations

```
write (x)
...
release (s) {unlock}
...
acquire (s) {lock}
...
read(x)
```

- Only those programs willing to be nondeterministic are not synchronised: programs with "data races"
- There are several variants of weak consistency, characterised by attitude towards: RAR, WAR, RAW, WAW to different addresses

Summary and Conclusions

- Shared memory parallel programs must synchronise
- Synchronisation primitives can be executed either
  - at the memory (as seen in GPUs)
  - On in the CPU – but this leads to issues cache coherency traffic when spinning, and when a contended lock is released
- While older ISAs offer test&set, compare-and-swap and atomic exchange as primitives, these are hard to implement
- Load-linked, store-conditional provides a solution that is easy to implement on a cache-coherent CPU
  - Key idea: operation only succeeds if no invalidation occurs in-between
- Test-and-test-and-set reduces contention for cache line ownership when spinning
- Ticket locks provide fairness
- Scalable locks limit coherency traffic on lock release
- Weak coherency results from not wanting to stall until invalidation is acknowledged
- Weak memory consistency models mean processes cannot reliably observe ordering of remote events unless explicit synchronisation takes place

Student question: ticket locks

Fairness: ticket locks

```
ticketLock_init(int *next_ticket, int *now_serving)
{
    *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
    my_ticket = fetch_and_inc(next_ticket);
    while (*now_serving != my_ticket) {} // Spin
}

ticketLock_release(int *now_serving)
{
    ++*now_serving;
}
```

- Ticket lock: explicitly hand off access to the next in line

Hi Professor, could you please explain again how ticket lock explicitly hand off access to the next in line?
Thanks.

- **The point of the ticket lock** is ***fairness***: we ensure that every thread that attempts to acquire the lock eventually gets it.

TL;DR:

- Since the my_ticket values are acquired in the order of thread arrival at the lock, subsequent acquisition of the lock is guaranteed to also be in this same order. Thus, fairness of lock acquisition is ensured, enforcing a FIFO ordering.

In more detail:

- So suppose one thread A is holding the lock, and will release it shortly. It holds a value of my_ticket which matches the now_serving value.
- We then have several other threads, B, C, D etc, that attempt to acquire the lock. They each do the fetch-and-increment in some sequence - so each thread gets its own value for my_ticket, each one bigger than the previous thread.
- Their my_ticket values are all larger than now_serving.
- Now, thread A releases the lock: it increments now_serving. Only one thread holds a my_ticket that matches that value - B, since B was the first. So B will exit the while loop and proceed, holding the lock.
- When B releases the lock, it increments now_serving, and thread C will gain the lock.

https://en.wikipedia.org/wiki/Ticket_lock
https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf