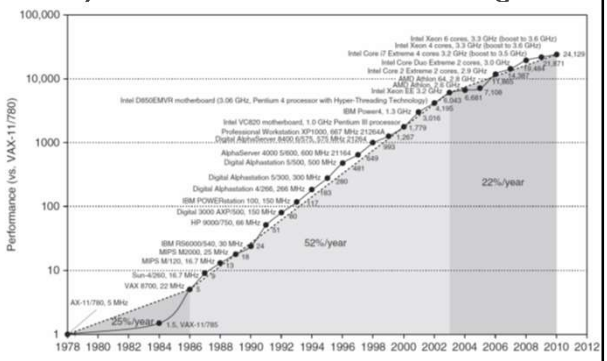
# SMP, Multicore, Memory Ordering & Locking

1



2

## CPU performance increases are slowing




Computer Architecture A Quantitative Approach Fifth Edition John L. Hennessy, David A. Patterson

3

## Multiprocessor System

A single CPU can only go so fast

- Idea: Use more than one CPU to improve performance
- Assumes
  - Workload can be parallelised
  - Workload is not I/O-bound or memory-bound

4

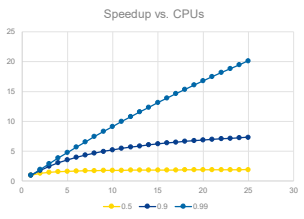
## Amdahl's Law

Given:

- Parallelisable fraction $P$
- Number of processor $N$
- Speed up $S$

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

$$S(\infty) = \frac{1}{(1-P)}$$




Parallel computing takeaway:

- Useful for small numbers of CPUs ($N$)
- Or, high values of $P$
  - *Aim for high $P$ values by design*

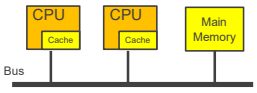
5

## Types of Multiprocessors (MPs)

Classic symmetric multiprocessor (SMP)

- Uniform Memory Access
  - Access to all memory occurs at the same speed for all processors.
- Processors with local caches
  - Separate cache hierarchy

    ⇒ Cache coherency issues

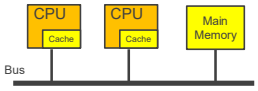



6

## Cache Coherency

What happens if one CPU writes to address 0x1234 (and it is stored in its cache) and another CPU reads from the same address (and gets what is in its cache)?

- Can be thought of as managing replication and migration of data between CPUs
- Note: The unit of replication and consistency is the cache line

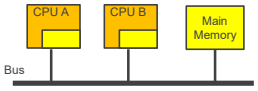



7

## Problematic Example

```
a = 1
if b == 0 then {
  /* critical section */
  a = 0
} else {
…
```

```
b = 1
if a == 0 then {
  /* critical section */
  b = 0
} else {
…
```




8

## Memory Model: Sequential Consistency

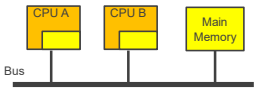
"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

9

## With sequential consistency

```
a = 1
if b == 0 then {
  /* critical section */
  a = 0
} else {
…
```

```
b = 1
if a == 0 then {
  /* critical section */
  b = 0
} else {
…
```



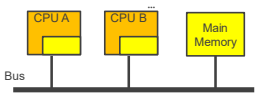

10

## Write-through Caches

- For classic SMP a hardware solution is used
  - Write-through caches
  - Each CPU cache snoops bus activity to invalidate stale lines
  - Reduces cache effectiveness – all writes go out to the bus.
    - Longer write latency
    - Reduced bandwidth

```
a = 1
if b == 0 then {
  /* critical section */
  a = 0
} else {
…
```

```
b = 1
if a == 0 then {
  /* critical section */
  b = 0
} else {
…
```

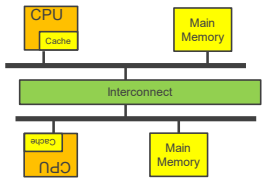



11

## Types of Multiprocessors (MPs)

NUMA MP

- Non-uniform memory access
  - Access to some parts of memory is faster for some processors than other parts of memory
- Provides high-local bandwidth and reduces bus contention
  - Assuming locality of access




12

## How is such a machine kept consistent?

Snooping caches assume

- write-through caches
- cheap "broadcast" to all CPUs

Many alternative cache coherency protocols

- They improve performance by tackling above assumptions
- We'll examine MESI (four state)
  - Optimisations exist (MOESI, MESIF)
- 'Memory bus' becomes message passing system between caches

13

## Example Coherence Protocol MESI

Each cache line is in one of four states

Invalid (I)

- This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.

Exclusive (E)

- The addressed line is in this cache only.
- The data in this line is consistent with system memory.

Shared (S)

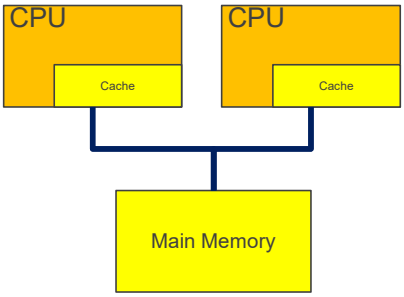
- The addressed line is valid in the cache and in at least one other cache.
- A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.

Modified (M)

- The line is valid in the cache and in only this cache.
- The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.

14

## Example




15

## MESI (with snooping/broadcast)

Events

RH = Read Hit

RMS = Read miss, shared

RME = Read miss, exclusive

WH = Write hit

WM = Write miss

SHR = Snoop hit on read

SHI = Snoop hit on invalidate

LRU = LRU replacement

Bus Transactions

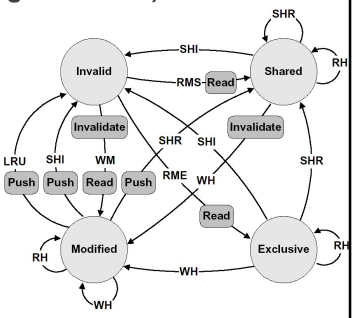
Push = Write cache line back to memory

Invalidate = Broadcast invalidate

Read = Read cache line from memory

Performance improvement via write-back caching

- Less bus traffic




16

## Directory-based coherence

Each memory block has a home node

Home node keeps directory of caches that have a copy

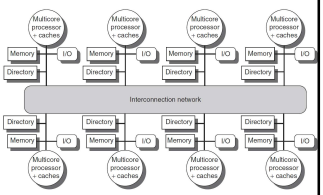
- E.g., a bitmap of processors per cache line

Pro

- Invalidation/update messages can be directed explicitly
  - No longer rely on broadcast/snooping

Con

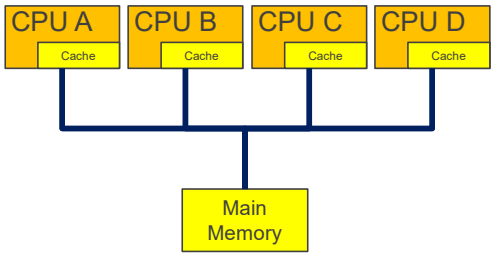
- Requires more storage to keep directory
  - E.g, each 256 bits of memory (cache line) requires 32 bits (processor mask) of directory




Computer Architecture A Quantitative Approach Fifth Edition John L. Hennessy, David A. Patterson

17

## Example




18

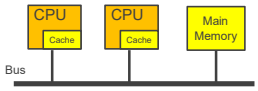
## Summary

Hardware-based cache coherency:

- Provide a consistent view of memory across the machine.
- Read will get the result of the last write to the memory hierarchy

19

## Memory Ordering




Example: a tail of a critical section

```
/* assuming lock already held */
/* counter++ */
load r1, counter
add r1, r1, 1
store r1, counter
/* unlock(mutex) */
store zero, mutex
```

Relies on all CPUs seeing update of counter before update of mutex

Depends on assumptions about ordering of stores to memory

20

## Memory Models: Strong Ordering

Sequential consistency

- the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

Traditionally used by many architectures

Assume X = Y = 0 initially

| CPU 0 | CPU 1 |
|---|---|
| store 1, X | store 1, Y |
| load r2, Y | load r2, X |

21

## Potential interleavings

At least one CPU must load the other's new value

- Forbidden result: X=0,Y=0

| | | |
|---|---|---|
| store 1, X<br>load r2, Y<br>store 1, Y<br>load r2, X<br>X=1,Y=0 | store 1, X<br>store 1, Y<br>load r2, Y<br>load r2, X<br>X=1,Y=1 | store 1, X<br>store 1, Y<br>load r2, X<br>load r2, Y<br>X=1,Y=1 |
| store 1, Y<br>load r2, X<br>store 1, X<br>load r2, Y<br>X=0,Y=1 | store 1, Y<br>store 1, X<br>load r2, X<br>load r2, Y<br>X=1,Y=1 | store 1, Y<br>store 1, X<br>load r2, Y<br>load r2, X<br>X=1,Y=1 |

22

## Realistic Memory Models

Modern hardware features can interfere with store order:

- write buffer (or store buffer or write-behind buffer)
- instruction reordering (out-of-order execution)
- superscalar execution and pipelining

Each CPU/core keeps its own execution consistent, but how is it viewed by others?

23

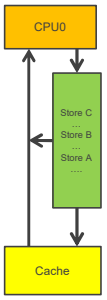
## Write-buffers and SMP

Stores go to *write buffer* to hide memory latency

- And cache invalidates

Loads read from write buffer if possible

```
CPU 0
store r1, A
store r2, B
store r3, C
load  r4, A
```




24

## Write-buffers and SMP

When the buffer eventually drains, what order does CPU1 see CPU0's memory updates?

```
CPU 0
store r1, A
store r2, B
store r3, C
```

```
CPU 1
```


What happens in our example?



## Total Store Ordering (e.g. x86)

Stores are guaranteed to occur in FIFO order

```
CPU 0
store 1, A
store 2, B
store 3, C
```

```
CPU 1 sees
A=1
B=2
C=3
```




## Total Store Ordering (e.g. x86)

Stores are guaranteed to occur in FIFO order

```
/* counter++ */
  load r1, count
  add r1, r1, 1
  store r1, counter
/* unlock(mutex) */
store zero, mutex
```

```
CPU 1 sees
count updated
 mutex = 0
```



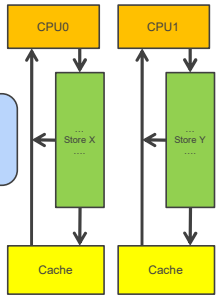

## Total Store Ordering (e.g. x86)

Assume X = Y = 0 initially

```
CPU 0
store 1, X
load r2, Y
```

```
CPU 1
store 1, Y
load r2, X
```

What is the problem here?






## Total Store Ordering (e.g. x86)

Stores are buffered in write-buffer and don't appear on other CPU in time.

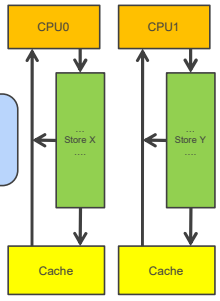
Can get X=0, Y=0!!!!

Loads can "appear" re-ordered with preceding stores

```
CPU 0
store 1, X
load r2, Y
```

```
CPU 1
store 1, Y
load r2, X
```

```
load r2, Y
load r2, X
store 1, X
store 1, Y
```






## Memory "fences"

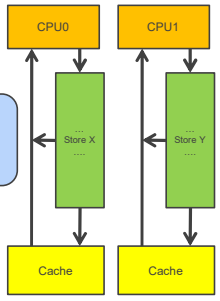
Also called "barriers"

The provide a "fence" between instructions to prevent apparent re-ordering

Effectively, they drain the local CPU's write-buffer before proceeding.

```
CPU 0
store 1, X
fence
load r2, Y
```

```
CPU 1
store 1, Y
fence
load r2, X
```



## Total Store Ordering

Stores are guaranteed to occur in FIFO order

Atomic operations?

```
CPU 0
ll r1, addr1
sc r1, addr1
```

```
CPU 1
ll r1, addr1
sc r1, addr1
```


- Need hardware support, e.g.
  - atomic swap
  - test & set
  - load-linked + store-conditional
- Stall pipeline and drain (and/or bypass) write buffer
- Ensures addr1 held exclusively

UNSW

31

## Partial Store Ordering (e.g. ARM MPcore)

All stores go to write buffer

Loads read from write buffer if possible

*Redundant stores are cancelled or merged*

```
CPU 0
store BUSY, addr1
store VAL, addr2
store IDLE, addr1
```

```
CPU 1 sees
addr2 = VAL
addr1 = IDLE
```


- Stores can appear to overtake (be re-ordered) other stores
- Need to use *memory barrier*

UNSW

32

## Partial Store Ordering (e.g. ARM MPcore)

The barriers prevent preceding stores appearing after successive stores

- Note: Reality is a little more complex (read barriers, write barriers), but principle similar.

```
load r1, counter
add r1, r1, 1
store r1, counter
barrier
store zero, mutex
```


- Store to counter can overtake store to mutex
  - i.e. update move outside the lock
- Need to use *memory barrier*
- Failure to do so will introduce subtle bugs:
  - Critical section "leaking" outside the lock

UNSW

33

## MP Hardware Take Away

Each core/cpu sees sequential execution of own code

Other cores see execution affected by

- Store order and write buffers
- Cache coherence model
- Out-of-order execution

Systems software needs to understand:

- Specific system (cache, coherence, etc..)
- Synch mechanisms (barriers, test_n_set, load_linked – store_cond).

…to build cooperative, correct, and scalable parallel code

UNSW

34

## MP Hardware Take Away

Existing sync primitives (e.g. locks) will have appropriate fences/barriers in place

- In practice, correctly synchronised code can ignore memory model.

However, racey code, i.e. code that updates shared memory outside a lock (e.g. lock free algorithms) must use fences/barriers.

- You need a detailed understanding of the memory coherence model.
- Not easy, especially for partial store order (ARM).

UNSW

35

## Memory ordering for various Architectures

| Type | Alpha | ARMv7 | PA-RISC | POWER | SPARC RMO | SPARC PSO | SPARC TSO | x86 | x86 oostore | AMD64 | IA-64 | zSeries |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Loads reordered after loads | Y | Y | Y | Y | Y | | | | Y | | Y | |
| Loads reordered after stores | Y | Y | Y | Y | Y | | | | Y | | Y | |
| Stores reordered after stores | Y | Y | Y | Y | Y | Y | | | Y | | Y | |
| Stores reordered after loads | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Atomic reordered with loads | Y | Y | | Y | Y | | | | | | Y | |
| Atomic reordered with stores | Y | Y | | Y | Y | Y | | | | | Y | |
| Dependent loads reordered | Y | | | | | | | | | | | |
| Incoherent instruction cache pipeline | Y | Y | | Y | Y | Y | Y | Y | Y | | Y | Y |

UNSW

36

## Concurrency Observations

Locking primitives require exclusive access to the "lock"

- Care required to avoid excessive bus/interconnect traffic

37

## Kernel Locking

Several CPUs can be executing kernel code concurrently.

Need mutual exclusion on shared kernel data.

Issues:

- Lock implementation
- Granularity of locking (i.e. parallelism)

38

## Mutual Exclusion Techniques

Disabling interrupts (CLI — STI).

- Insufficient for multiprocessor systems.

Spin locks.

- Busy-waiting wastes cycles.

Lock objects (locks, semaphores).

- Flag (or a particular state) indicates object is locked.
- Manipulating lock requires mutual exclusion.

39

## Hardware Provided Locking Primitives

```
int test_and_set(lock *);
int compare_and_swap(int c,
                     int v, lock *);
int exchange(int v, lock *)
int atomic_inc(lock *)

v = load_linked(lock *) / bool
  store_conditional(int, lock *)
```

- LL/SC can be used to implement all of the above

40

## Spin locks

```
void lock (volatile lock_t *l) {
   while (test_and_set(l)) ;
}
void unlock (volatile lock_t *l) {
   *l = 0;
}
```

Busy waits. Good idea?

41

## Spin Lock Busy-waits Until Lock Is Released

Stupid on uniprocessors, as nothing will change while spinning.

- Should release (block) thread on CPU immediately.

Maybe ok on SMPs: locker may execute on other CPU.

- Minimal overhead (if contention low).
- Should only spin for short time.

Generally restrict spin locking to:

- *short* critical sections,
- unlikely to (or preferably can't) be contended by thread on same CPU.
  - local contention can be prevented
    - » by design (per-CPU data structure)
    - » by turning off interrupts

42

## Spinning versus Switching

- Blocking and switching
  - to another process takes time
    - » Save context and restore another
    - » Cache contains current process not new
      - Adjusting the cache working set also takes time
    - » TLB is similar to cache
  - Switching back when the lock is free encounters the same again
- Spinning wastes CPU time directly

Trade off

- If lock is held for less time than the overhead of switching to and back

⇒ It's more efficient to spin

43

## Spinning versus Switching

The general approaches taken are

- Spin forever
- Spin for some period of time, if the lock is not acquired, block and switch
  - The spin time can be
    - » Fixed (related to the switch overhead)
    - » Dynamic
      - Based on previous observations of the lock acquisition time

44

## Interrupt Disabling

Assume no local contention by design, is disabling interrupt important?

Hint: What happens if a lock holder is preempted (e.g., at end of its timeslice)?

All other processors spin until the lock holder is re-scheduled

45

## Alternative to spinning: Conditional Lock (TryLock)

```
bool cond_lock (volatile lock t *l) {
  if (test_and_set(l))
      return FALSE; //couldn't lock
  else
      return TRUE; //acquired lock
}
```

Can do useful work if fail to acquire lock.

**But** may not have much else to do.

Livelock: May never get lock!

46

## Another alternative to spinining.

```
void mutex lock (volatile lock t *l) {
  while (1) {
      for (int i=0; i<MUTEX N; i++)
            if (!test and set(l))
                  return;
      yield();
  }
}
```

Spins for limited time only

- assumes enough for other CPU to exit critical section

Useful if critical section is shorter than N iterations.

Starvation possible.

47