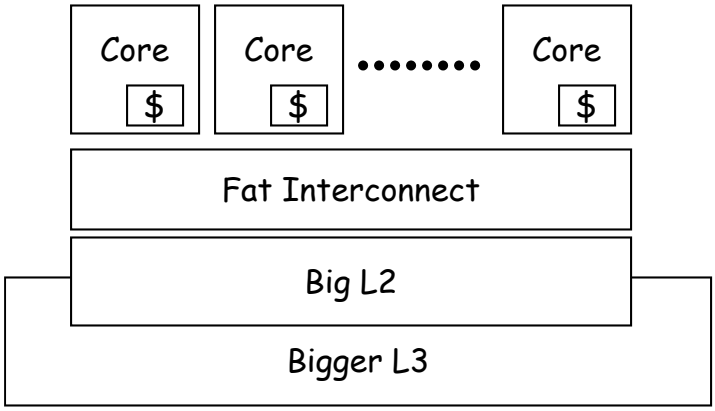
18-447 Lecture 23: Cache Coherence and Synchronizations

Nikos Hardavellas
Dept of ECE, CMU
April 20, 2009

Announcements: Read P&H Ch5.8 and Ch2.11 for today's lecture
(this lecture not covered on the final)
HW 4 extended to next Monday 4/27

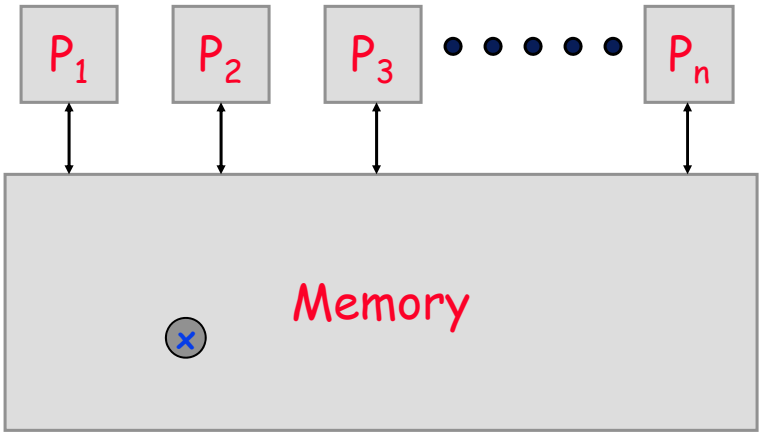
Handouts:

Chip-Multiprocessor

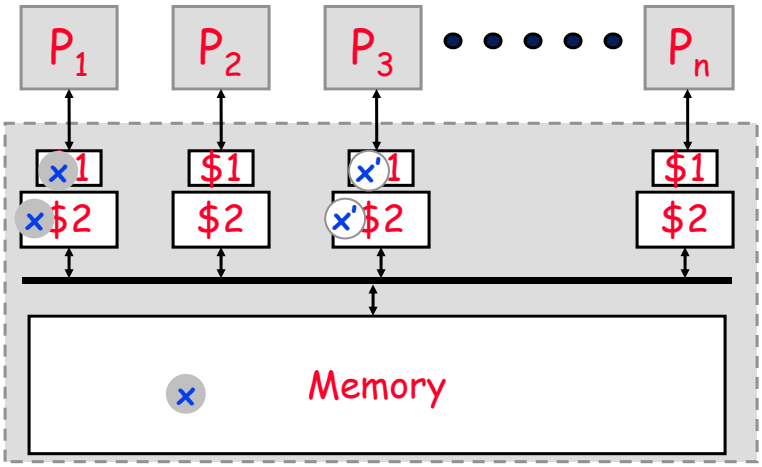


- Current CMPs adopt the familiar SMP paradigm
- future design focus on the "uncore"
  - how to support interprocessor communication
  - how to support programmability

Shared Memory Abstraction



Multiprocessor with Private Caches



The goal of cache coherence is to make all the processors believe they are connected to the same memory directly (warning: oversimplified)

Extreme Solutions to Cache Coherence

- Disallow caching of shared variables
- Only allow only one copy of a mem location at a time
  - If location X is cached in one cache then it is not valid in memory or another cache
  - Another processor must have a way to find out who has location X and take over ownership before reading or writing
  - thus, can only have one reader/writer per location
- Allow multiple copies, but make sure they all have the same value at all times
  - updates to one copy must be visible to all copies where ever they may be (memory and all of the caches)
  - thus, can have multiple readers and writers at once

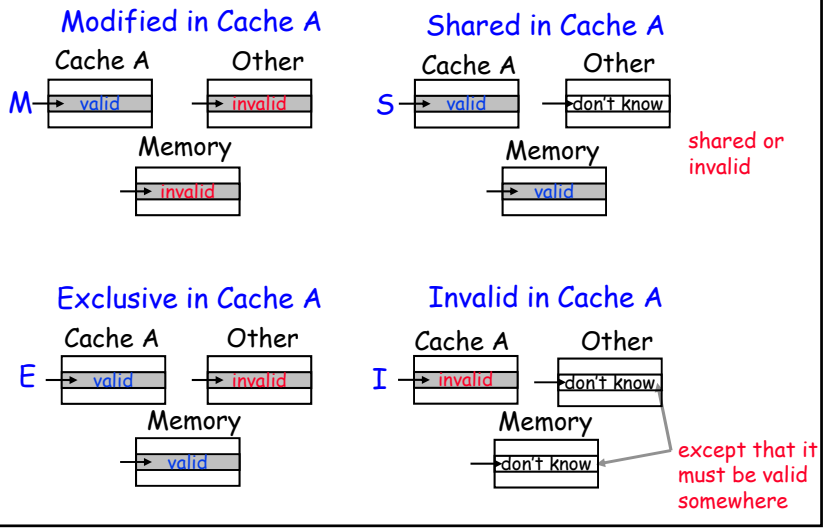
A cache coherence protocol is the "rule of conduct" between caches to enforce a particular policy

CC Protocol for Bus-based Systems

- Bus is a broadcast medium, bus "snooping" allows every cache to see what everyone else wants to do
- A cache can even intervene in another cache's bus transaction, e.g. a cache might ask another cache to "retry" the transaction later or respond in place of the memory
- Besides the usual status bits, additional information might have to be recorded with each cache line, aka cache coherence states, e.g.
  - **Invalid:** cache line does not have valid data
  - **Modified:** cache line has been written to since it was brought in
  - **Shared:** valid line, but other caches may have copies (presumably all identical and unchanged from memory)
  - **Exclusive:** valid line, unchanged from memory but no other cache has a copy

MESI States

Given the state of an address in one cache, what can one infer about the possible state of the same address elsewhere?



Example: Multiple Identical Copies

- A cache line can be either **Shared** or **Invalid**
- Based on a write-through scheme
  - a cache issues a read transaction on a read or write miss
  - a cache issues a write transaction to memory whenever the cache line is changed by the processor
  - a cache does not need to write back when a line is displaced
- All writes are write-through so the writer's cache is coherent with memory
- All caches "snoop" the bus for other's write transactions
  - check if the write is to a currently cached location
  - if a write goes to a cached location, overwrite the old (aka stale) value with the new snooped value
  - else do nothing

All copies are coherent all the time

Example: One Copy at All Time

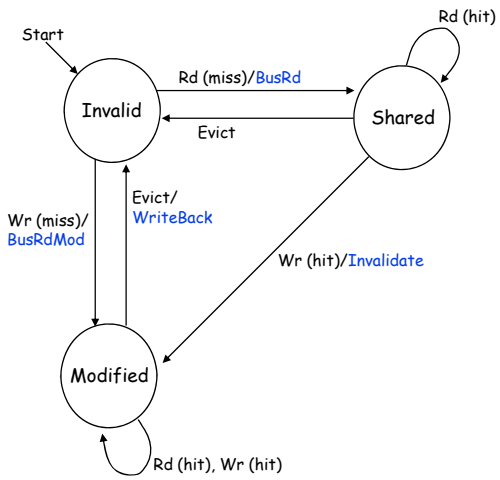
- A cache line can be either **Modified** or **Invalid**
- Based on a write-back scheme
  - a cache issues a read transaction on a read or write miss
  - a cache issues a write-back to memory when a line is displaced
- All caches "snoop" the bus for other's read transactions
  - If a cache observes a request to a currently cached line, then respond with a value in place of memory and mark its own copy **Invalid**
  - Alternatively, a cache can also ask the requestor to retry later and, in the meantime, write back its copy to memory

Why don't caches need to snoop for write-back transactions?

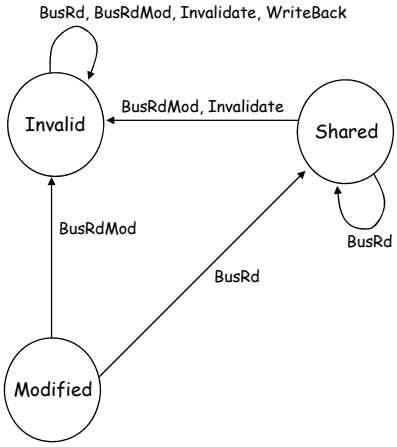
MSI and MESI Cache Coherence

- An efficient policy for single-writer/multi-reader usage
  - Allow multiple read-only copies (all identical) **(Shared)**
  - Allow only a single writable copy **(Exclusive, Modified)**
  - Minimizes the number of bus transactions
- Based on a write-back scheme
  - On a read miss, issue a read transaction for a read-only copy
  - On a write miss, issue a "read-with-intent-to-modify" for an exclusive copy
  - On a write hit to a read-only copy, issue an "invalidate" transaction
  - When displacing a **Exclusive** (i.e., "clean") line, do nothing
  - When displacing a **Modified** line, write the dirty value back to memory
- All caches "snoop" the bus for other caches' read, RWITM and invalidate transactions

MSI State Transition Diagram



a. State transitions using signals from the processor



b. State transitions using signals from the bus

Implementation of Snoopy Busses

- Every bus snoop requires a cache lookup
  - needs a dual ported cache or at least an extra tag lookup port
  - in an inclusive L1 and L2 arrangement, snoop only goes to L2 and does not contend with processor for L1 bandwidth
- Broadcast medium is not very scalable
  - physical limits (such as number of drops and physical extent of the bus) force bigger busses to clock slower
  - bus bandwidth is divided when you add more processors
- Snoopy protocols are conceptually simple but "high-performance" implementations can get very complicated
  - MESI state transitions are not really atomic
  - CPU and bus transactions are not atomic
  - CC issues can become intertwined with memory consistency

Synchronizations

Multiprocessor Memory Consistency

- A memory consistency model tells the programmer for each load which store wrote the value to be returned
- Intuition: a load should return the value of the "last" store to the same memory address
- Suppose in a multiprocessor system, each of P1, P2 and P3 performs a stream of reads and writes

  . . . . . $W_{P1}(x)$ . . . . . .
  . . . . . $W_{P2}(x)$,$W_{P2}(y)$, $R_{P2}(x)$, $R_{P2}(y)$ . . . . . .
  . . . . . $W_{P3}(x)$. . . . $W_{P3}(y)$. . . . . $W_{P3}(x)$ . . . . . .

- Who performed the last write to x before x is read by P2?
- How do you establish a global ordering of memory operations? Do you need a global ordering?

Sequential Consistency [Lamport]

- For this lecture, let's only consider the "strongest" consistency model, but know that there are others
- Sequential Consistency (should match your intuition)
  - a processor perceives its own memory ops in program order
  - memory ops from different processors can be interleaved arbitrarily (different interleaving are allowed on different runs)
  - but for each run, all processors must agree on the same total ordering

Example: Thread A and Thread B share variables X and Y (initially X = 0, Y = 0)

Thread A:	Thread B:
.....	.....
M[X] <- 1;	r1 <- M[Y];
M[Y] <- 1;	r2 <- M[X];

SC says the final values of r1 and r2 may be different from run to run but for sure if r1 is 1 then r2 cannot be 0

(BTW, r2 could be 0 in some weaker consistency models)

Data Races when Sharing Memory

- Two threads A and B repeatedly increment a shared variable C in memory (assuming SC)

```
//Thread A
_loop:
        R8<-M[C]
        R8<-R8+1
        M[C]<-R8
        goto _loop
```

```
//Thread B
_loop:
        R8<-M[C]
        R8<-R8+1
        M[C]<-R8
        goto _loop
```

- The intention of the program is clear, but what actually happens depends on what Thread B does between when Thread A reads and writes to C (and vice versa)

Atomic Read-Modify-Write

- A special class of memory instructions is need for synchronization (between concurrent threads or between time-multiplexed threads)
- An semantically atomic instruction that
  - read a shared memory location
  - perform some simple computation
  - write something back to the same memory location

  Note: atomic means the memory location cannot be written to by anyone else between the read and the write back
- E.g.,

```
Swap(addr,r):
  temp=M[addr];
  M[addr]=r;
  r=temp;
```

or

```
Fetch&Add(addr,v):
  temp=M[addr];
  M[addr] = temp+v;
```

- This class of instructions is very expensive to implement and to perform

RMW Examples

- We could rewrite the earlier counting example using atomic Fetch&Add, or
- A general technique is to develop a "lock" acquire and release protocol to protect "critical sections" that should not be interleaved
- Given a special lock variable L

```
//Thread A
_loop:
        Acquire(L)
        R8<-M[C]
        R8<-R8+1
        M[C]<-R8
        Release(L)
        goto _loop
```

```
//Thread B
_loop:
        Acquire(L)
        R8<-M[C]
        R8<-R8+1
        M[C]<-R8
        Release(L)
        goto _loop
```

Acquire and Release

- Using Swap, L initially 0

```
Acquire(L)
 do {
        r_temp <- 1;
        swap(L, r_temp);
 } while (r_temp);
```

```
Release(L)
  M[L]<-0;
```

- Using Fetch&Add, L initially 0

```
Acquire(L)
 do {
        fetch&Add(L,1);
 } while (M[L] !=1 );
```

```
Release(L)
  M[L]<-0;
```

- Many equally powerful RMW instructions have been proposed

Special Load and Store for RMW

- Atomic RMW can be mimicked by a pair of load and store instructions with "extra" semantics
- Each core needs a special < reserved, address> register, and a store-conditional status bit

```
load-linked(r, addr):
    r=M[addr];
    < reserved, address> = < 1, addr >

store-conditional(r, addr):
    if < reserved, address> = < 1, addr > then
            M[addr]=r
            set status bit to succeed
    else
            set status bit to fail
```

- Much more efficient to implement than true atomic ops

LL/SC Examples

- Basic strategy when mimicking an atomic RWM operation ➔ keep retrying until you know you have succeeded uninterrupted)

```
Swap(addr,r):
loop:   ll(r_temp,addr)
        sc(r,addr)
        if status=failed then go to _loop
        r=r_temp
```

if m is modified in between sc will fail and you try again

```
Fetch&Add(m,v):
loop:   ll(r_temp, addr)
        r_temp= r_temp+ v
        store-conditional(r_temp, addr)
        if status=failed then go to loop
```