

Systems@ETH.zurich

Lecture 24: Multiprocessing


Computer Architecture and
Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012


Systems@ETH.zurich


Most of the rest of this course...

- Multiprocessing:
 - Hardware architecture
 - Performance implications
 - Programming facilities for synchronization
- Formerly something of a niche
 - Mainly supercomputing
- Now mainstream, with a vengeance!
 - Multicore processors
 - We will see why...


Systems@ETH.zurich


Today

- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- Sequential Consistency
- Snoopy Caches
 - MSI cache coherence
 - MESI
 - MOESI
- Relaxing memory consistency models

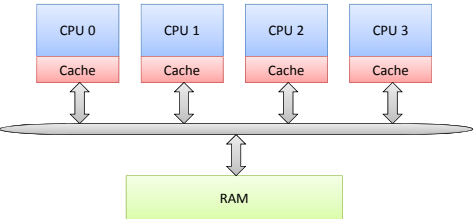

Systems@ETH.zurich

Very simple start...

- Problem:
 - One processor isn't fast enough
- Observation:
 - Some jobs can parallelize
 - Multiple jobs can run at the same time
- Solution:
 - Attach multiple processors to the system bus



Systems@ETH.zurich

Symmetric multiprocessing (SMP)



SMP only works because of caches!

- Shared memory rapidly becomes bottleneck


Systems@ETH.zurich

Today

- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- Sequential Consistency
- Snoopy Caches
 - MSI cache coherence
 - MESI
 - MOESI
- Relaxing memory consistency models

Coherency and Consistency



- As with DMA, memory can change under a cache
 - Writes from other processors to memory
 - Leads to 2 important concepts:
- 1. **Coherency:**
 - Values in caches all match each other
 - Processors all see a coherent view of memory
- 2. **Consistency:**
 - The order in which changes to memory are seen by different processors

Cache coherency



- Most CPU cores on a modern machine are **cache coherent**
 - Behave as if all accessing a single memory array
 - We'll see what this *really* means in a moment
- Big advantage: ease of programming
 - Shared-memory programming models work!
 - Pthreads, OpenMP, etc.
- Disadvantages:
 - Complex to implement (lots of transistors, bug-prone)
 - Memory is slower as a result

Memory consistency



- When several processors are reading and writing memory, what value is read by each processor?
 - Not an easy question to answer
 - "Last value written":
 - By which processor?
 - What do we mean by "last"?
- Important to have an answer!
 - Defines semantics of order-dependent operations
 - E.g. does Dekker's algorithm work?
 - How to ensure that it does work?
- There are many memory consistency **models**

Consistency models: terminology



- **Program order:** order in which a program on a processor appears to issue reads and writes
 - Refers only to local reads/writes
 - Even on a uniprocessor \neq order the CPU issues them!
 - Write-back caches, write buffers, out-of-order execution, etc.
- **Visibility order:** order which all reads and writes are seen by one or more processors
 - Refers to all operations in the machine
 - Might not be the same for
 - Each processor reads the value written by the last write in visibility order

Today



- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- **Sequential Consistency**
- Snoopy Caches
 - MSI cache coherence
 - MESI
 - MOESI
- Relaxing memory consistency models

Sequential consistency



1. Operations from a processor appear (to all others) in program order
2. Every processor's visibility order is the same interleaving of all the program orders.

Requirements:

- Each processor issues memory ops in program order
- RAM totally orders all operations
- Memory operations are atomic

Sequential consistency example



Results:

CPU A	CPU B
a ₁ : *p = 1;	b ₁ : u = *q;
a ₂ : *q = 1;	b ₂ : v = *p;

- (u=1, v=1):
 - Possible under SC: (a₁, a₂, b₁, b₂)
 - (a₁, a₂) and (b₁, b₂) are both program orders
- (u=1, v=0):
 - Impossible under SC:
 - No interleaving of program orders that generates this result
 - Would require: a₂ > b₁ > b₂ > a₁

Sequential consistency example



Results:

CPU A	CPU B
a ₁ : *p = 1;	b ₁ : *q = 1;
a ₂ : u = *q;	b ₂ : v = *p;

- (u=1, v=1):
 - Possible under SC: (a₁, b₁, a₂, b₂)
 - (a₁, a₂) and (b₁, b₂) are both program orders
- (u=0, v=0):
 - Impossible under SC:
 - No interleaving of program orders that generates this result
 - Would require: a₂ > b₁ > b₂ > a₁

Sequential consistency



- Advantages:
 - Easy to understand for the programmer
 - Easy to write correct code to
 - Easy to analyze automatically
- Disadvantages:
 - Hard to build a fast implementation
 - Cannot reorder reads/writes
 - even in the compiler
 - even from a single processor!
 - Cannot combine writes to same cache line (write buffer)
 - Serializing ops at memory controller is too restrictive:
 - see NUMA later

Today



- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- Sequential Consistency
- Snoopy Caches
 - MSI cache coherence
 - MESI
 - MOESI

Implementing SC with a snoopy cache

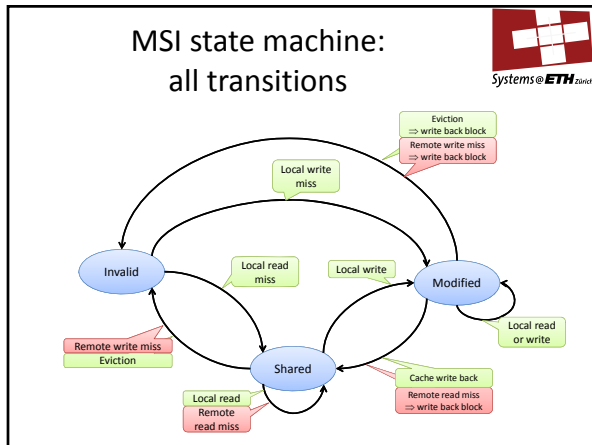
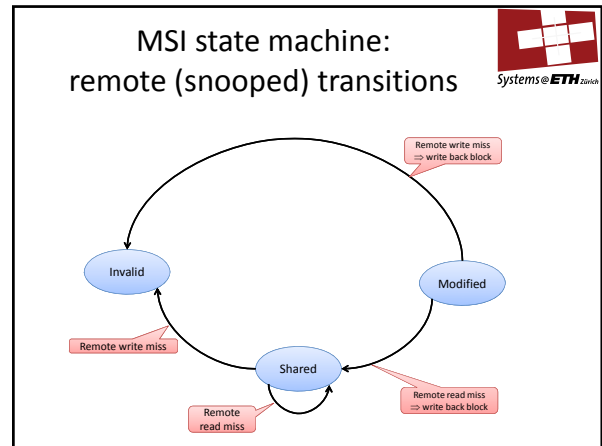
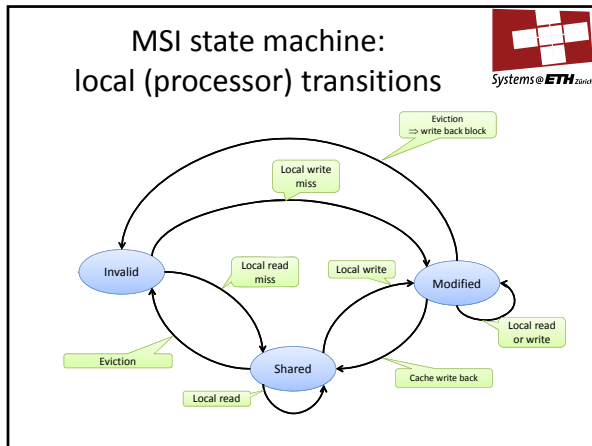


- Cache “snoops” on reads/writes from other processors
- If a line is valid in local cache:
 - Remote (other processor) write to line
⇒ invalidate local line
- Requires a write-through cache!
 - But coherency mechanism ⇒ sequential consistency
- Line can be valid in many caches, until a write

What about write-back caches?



- Cache lines can now be “dirty” (modified)
- Requires a **cache coherency protocol**
- Simplest protocol: MSI
 - Each line has 3 states: Modified, Shared, Invalid
 - Line can only be dirty in one cache
- Cache logic must respond to:
 - Processor reads and writes
 - Remote bus reads and writes
- and must:
 - Change cache line state
 - Write back data (**flush**) if required



MSI issues

Assumes we can distinguish remote processor read and write misses

- In I state, executing a write miss:
 - Need to first **read** line (allocate)
 - If someone else has it in M state, need to wait for flush
- In M state, other core observes a remote read:
 - Must flush line (required)
 - Invalidate line?
 - But what if you want read sharing? Extra cache miss!
 - Transition to shared?
 - But what if it's actually a remote **write** miss? Extra invalidate!

MESI protocol

- Add a new line state: "exclusive"
 - Modified:** This is the only copy, it's dirty
 - Exclusive:** This is the only copy, it's clean
 - Shared:** This is one of several copies, all clean
 - Invalid**
- Add a new bus signal: **RdX**
 - "Read exclusive"
 - Cache can load into either "shared" or "exclusive" states
 - Other caches can see the type of read
- Also: **HIT** signal
 - Signals to a remote processor that its read hit in local cache.
- First x86 appearance in the Pentium

MESI invariants

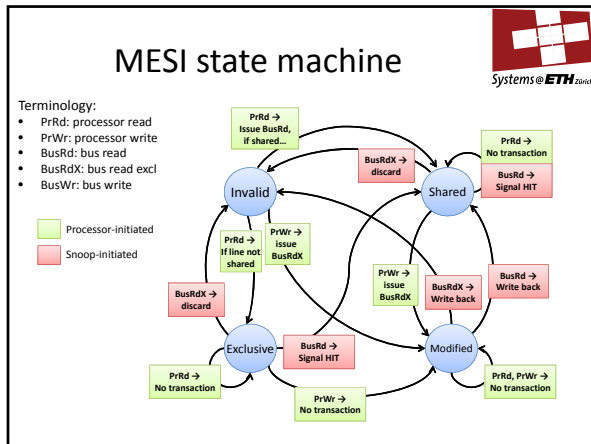
- Allowed combination of states for a line between any pair of caches:

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

- Protocol must preserve these invariants

	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

MSI invariants:



- ### MESI observations
- Dirty data always written through memory
 - No cache-cache transfers
 - “Invalidation-based” protocol
 - Data is always either:
 - Dirty in one cache
 - ⇒ must be written back before a remote read
 - Clean
 - ⇒ can be safely fetched from memory
- Good if:
- latency of memory << latency of remote cache

- ### MOESI protocol
- Add new “Owner” state: allow line to be modified, but other unmodified copies to exist in other caches.
- Modified:** No other cached copies exist, local copy dirty
 - Owner:** Unmodified copies may exist, local copy is dirty
 - Exclusive:** No other cached copies exist, local copy clean
 - Shared:** Other cached copies exist, local copy clean
One other copy might be dirty (state Owner)
 - Invalid:** Not in cache.

- ### MOESI invariants
- Allowed combination of states for a line between any pair of caches:
- | | | | | | |
|---|---|---|---|---|---|
| | M | O | E | S | I |
| M | | | | | ✓ |
| O | | | | | ✓ |
| E | | | | | ✓ |
| S | ✓ | | | | ✓ |
| I | ✓ | ✓ | ✓ | ✓ | ✓ |
- MOESI can satisfy a read request in state I from a remote cache in state O, for example.
- Good if:
- latency of remote cache < latency of main memory


- ### Relaxing sequential consistency
- | | | |
|------------------|---------|--------------------------|
| | CPU A | CPU B |
| a ₁ : | *p = 1; | b ₁ : u = *q; |
| a ₂ : | *q = 1; | b ₂ : v = *p; |
- Recall program order requirement for SC:
 - Out-of-order execution might reorder (b₂, b₁)
 - Write buffer might reorder (a₁, a₂)
 - a₁ might miss in the cache, and a₂ hit
 - Compiler might reorder operations in each thread
 - Or optimize out entire reads or writes
 - What can be done?

- ### Relaxing sequential consistency
- Many, many different ways to do this!

E.g.:


 - Write-to-read: later reads can bypass earlier writes
 - Write-to-write: later writes can bypass earlier writes
 - Break write atomicity (no single visibility order)
 - Weak ordering: no implicit order guarantees at all
 - Explicit synchronization instructions
 - x86: lfence (load fence), sfence (store fence), mfence (memory fence)
 - Alpha: mb (memory barrier), wmb (write memory barrier)

Processor Consistency



- Also PRAM (Pipelined Random Access Memory)
 - Implemented in Pentium Pro, now part of x86 architecture.
- Write-to-read relaxation: later reads can bypass earlier writes
 - All processors see **writes** from one processor in the order they were issued.
 - Processors can see **different interleavings of writes** from different processors.


Processor (PRAM) Consistency



CPU A	CPU B	CPU C
a ₁ : *p = 1;	b ₁ : u = *p;	c ₁ : v = *q;
	b ₂ : *q = 1;	c ₂ : w = *p;

- (u,v,w) = (1,1,0) is possible in PC
 - B sees visibility order (a₁, b₂)
 - C sees visibility order (b₂, a₁)

Other consistency models



	Alpha	PA-RISC	Power	X86_32	X86_64	ia64	zSeries
Reads after reads	✓	✓	✓			✓	
Reads after writes	✓	✓	✓			✓	
Writes after reads	✓	✓	✓	✓	✓	✓	✓
Writes after writes	✓	✓	✓			✓	
Dependent reads	✓					✓	
lfetch after write	✓		✓	✓		✓	✓


Cache is incoherent: requires explicit lcache flushes for self-modifying code

Read of value can be seen before read of address of value!

Not shown: SPARC, which supports 3 different memory models


Portable languages like Java must define their own memory model, and enforce it!

Summary



- Symmetric Multiprocessing (SMP)
- Consistency and Coherence
- Sequential Consistency
- Snoopy Caches
 - MSI cache coherence
 - MESI
 - MOESI
- Relaxing memory consistency models

Next time: synchronization



- Barriers and fences
- Test and Set
- Compare and Swap
- Load-Locked / Store Conditional
- Limits of symmetric multiprocessing
- Simultaneous multithreading (SMT)