

Shared Memory Multiprocessors

Mark Greenstreet

CpSc 418 – January 19, 2018

Outline:

- Shared-Memory Architectures
- Memory Consistency
- Coding Break
- Weak Consistency

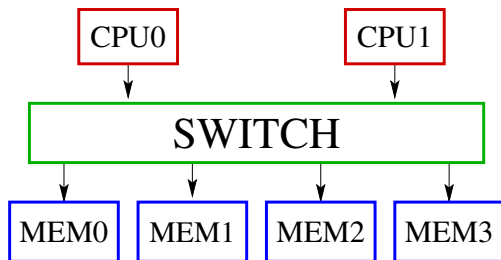


Unless otherwise noted or cited, these slides are copyright 2017 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license <http://creativecommons.org/licenses/by/4.0/>

Objectives

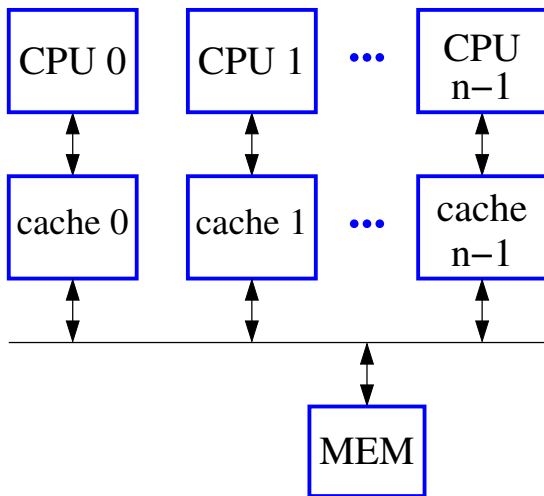
- Understand how processors can communicate by sharing memory.
- Able to explain the term “sequential consistency”
 - ▶ Describe a simple cache-coherence protocol, MESI
 - ▶ Describe how the protocol can be implemented by snooping.
 - ▶ Describe “sequential consistency”.
 - ▶ Be aware that real machines make guarantees that are weaker than sequential consistency.

An Ancient Shared-Memory Machine



- Multiple CPU's (typically two) shared a memory
- If both attempted a memory read or write at the same time
 - ▶ One is chosen to go first.
 - ▶ Then the other does its operation.
 - ▶ That's the role of the switch in the figure.
- By using multiple memory units (partitioned by address), and a switching network, the memory could keep up with the processors.
- But, now that processors are 100's of times faster than memory, this isn't practical.

A Shared-Memory Machine with Caches

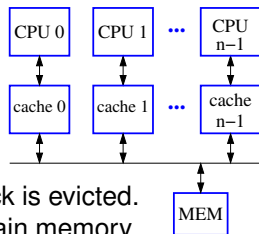


- Caches reduce the number of main memory reads and writes.
- But, what happens when a processor does a write?

Cache Inconsistency

- Assume caches are write-back:

- ▶ **write-back**: writes only update the cache. Main memory updated when the cache block is evicted.
- ▶ **write-through**: writes update cache and main memory.
- ▶ Modern processors have to use write-back for performance: Main memory is way too slow for write-through.

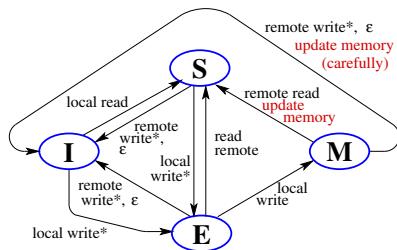


- Step 0: CPU 0 and CPU 1 have both read memory location `addr0` and `addr1` and have copies in their cache.
- Step 1: CPU 0 writes to `addr0` and CPU 1 writes to `addr1`.
- Step 2: CPU 0 reads from `addr1` and CPU 1 reads from `addr0`.
 - ▶ Both CPUs see the **old** value.
 - ▶ The writes only updated the writer's cache.
 - ▶ The readers got the old values.

Cache Coherence Protocols

- **Big idea:** caches communicate with each other so that:
 - ▶ Multiple CPUs can have read-only copies for the same memory location.
 - ▶ If a cache has a dirty block, then no other cache has a copy of that block.

The MESI protocol



I = invalid
S = shared
E = exclusive
M = modified

write* = write-through
(to memory)

write = write-back
(local-cache only)

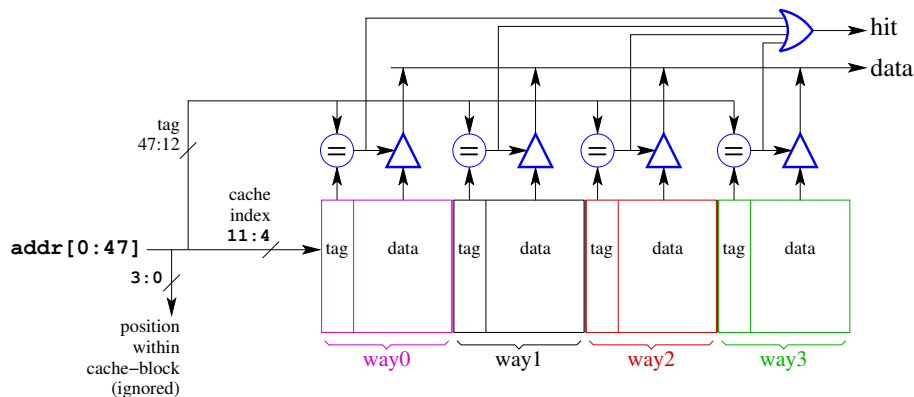
ϵ = "spontaneous"
transition

- Caches can **share read-only** copies of a cache block.
- When a processor writes a cache block, the first write goes to main memory.
 - ▶ The other caches are notified and invalidate their copies.
 - ▶ This ensures that **writable blocks are exclusive**.

How caches work

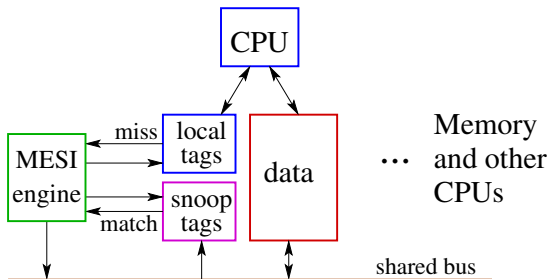
- Caching rhymes with hashing and the two ideas are similar.
 - ▶ Caches store data in “blocks” – the block size is a small power-of-two times the machine word size.
 - ▶ A cache has one or more “ways” – each way holds a power-of-two number number of blocks.
 - ▶ A hash-value is computed from the address.
 - ★ `blockAddr = addr / blockSize; % right shift`
 - ★ `blockIndex = blockAddr % (BlocksPerWay-1); % bit masking`
- Read:
 - ▶ The `blockIndex` is used to look up one entry in each “way”.
 - ▶ Each block has a tag that includes the full-address for the data stored in that block.
 - ▶ The tags from each way are compared with the tag of the address:
 - ★ If any tag matches, that way provides the data.
 - ★ If no tags match, then a cache miss occurs.
 - ★ Some current block is evicted from the cache to make room for the incoming block.
- Writes are similar to reads.

A typical cache



- Only the read-path is shown. Writing is similar.
- This is a 16K-byte, 4-way set-associative cache, with 16 byte cache blocks.

Implementing MESI: Snooping



- Caches read and write main memory over a shared memory bus.
- Each cache has two copies of the tags: one for the CPU, the other for the bus.
- If the cache sees another CPU reading or writing a block that is in this cache, it takes the action specified by the MESI protocol.

Implementing MESI: Directories

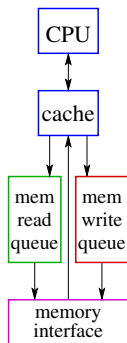
- Main memory keeps a copy of the data **and**
 - ▶ a bit-vector that records which processors have copies, and
 - ▶ a bit to indicate that one processor has a copy and it may be modified.
- A processor accesses main memory as required by the MESI protocol.
 - ▶ The memory unit sends messages to the other CPUs to direct them to take actions as needed by the protocol.
 - ▶ The ordering of these messages ensures that memory stays consistent.
- Comparison:
 - ▶ Snooping is simple for machines with a small number of processors.
 - ▶ Directory methods scale better to large numbers of processors.

Sequential Consistency

Memory is said to be **sequentially consistent** if

- All memory reads and writes from all processors can be arranged into a single, sequential order, such that:
 - ▶ The operations for each processor occur in the global ordering in the same order as they did on the processor.
 - ▶ Every read gets the value of the preceding write to the same address.
- Sequential consistency corresponds to what programmers think “ought” to happen.
 - ▶ Very similar to “serializability” for database transactions.
- MESI guarantees sequential consistency

Weak Consistency



- CPUs typically have “write-buffers” because memory writes often come in bursts.
- Typically, reads can move ahead of writes to maximize program performance.
- Why?
 - ▶ Because there may be instructions waiting for the data from a load.
 - ▶ A transition from “shared” to “modified” requires notifying **all** processors – this can take a long time.
 - ▶ Memory writes don’t happen until the instruction commits.
- This means that real computers don’t guarantee sequential consistency.
 - ▶ **Warning:** classical algorithms for locks and shared buffers fail when run on a real machines!

Programming Shared Memory Machines

- Shared memory make parallel programming “easier” because:
 - ▶ One thread can pass an entire data structure to another thread just by giving a pointer.
 - ▶ No need to pack-up trees, graphs, or other data structures as messages and unpack them at the receiving end.
- Shared memory make parallel programming **harder** because:
 - ▶ It’s easy to overlook synchronization (control to shared data structures). Then, we get data races, corrupted data structures, and other hard-to-track-down bugs.
 - ▶ A defensive reaction is to wrap *every* shared reference with a lock. But locks are slow (that λ factor for communication), and this often results is slow code, or even deadlock.
- In practice, shared memory code that works often has a message-passing structure.
- Finally, beware of weak consistency
 - ▶ Use a thread library.
 - ▶ There are elegant algorithms that avoid locking overhead, even with weak consistency, but they are beyond the scope of this class.

Shared Memory and Performance

- Shared memory can offer better performance than message passing because
 - ▶ High bandwidth: the buses that connect the caches can be very wide, especially if the caches are on a single chip.
 - ▶ Low latency: the hardware handles moving the data – no operating system calls and context-switch overheads.
- But, shared memory doesn't scale as well as message passing
 - ▶ For large machines, the latency of directory accesses can severely degrade performance.
 - ★ In a message passing machine, each CPU has its own memory, nearby and fast.
 - ★ For shared memory, each CPU has part of the shared main memory – accessing a directory may require accessing the memory of a distant CPU.
 - ▶ Shared memory moves the data after the cache miss
 - ★ this stalls a thread
 - ★ message passing can send data in advance and avoid these stalls

Summary

● Shared-Memory Architectures

- ▶ Use cache-coherence protocols to allow each processor to have its own cache while maintaining (almost) the appearance of having one shared memory for all processors.
 - ★ A typical protocol: MESI
 - ★ The protocol can be implemented by snooping or directories.
- ▶ Using cache-memory interconnect for interprocessor communication provides:
 - ★ High-bandwidth
 - ★ Low-latency, but watch out for fences, etc.
 - ★ High cost for large scale machines.

● Shared-Memory Programming

- ▶ Need to avoid interference between threads.
 - ★ Assertional reasoning (e.g. invariants) are crucial, much more so than in sequential programming.
 - ★ There are too many possible interleavings to handle intuitively.
 - ★ In practice, we don't formally prove complete programs, but we use the ideas of formal reasoning.
- ▶ Real computers don't provide sequential consistency.
 - ★ Use a thread library.

Preview

January 22: Distributed-Memory Machines

Reading: Pacheco, Chapter 2, Sections 2.4 and 2.5.

January 24: Speed-up

Reading: McCool *et al.*, Chapter 2, Section 2.5.

January 26: Energy, Power, and Time

January 29: Performance Loss

Reading: McCool *et al.*, Chapter 2, Section 2.6.

Homework: **HW 2 earlybird** (11:59pm), HW 3 goes out.

January 31: Parallel Performance: Models

Homework: **HW 2 due** (11:59pm).

February 2-9: Sorting

Review

- What is sequential consistency?
- Using the MESI protocol, can multiple processors simultaneously have entries in their caches for the same memory address?
- Using the MESI protocol, can multiple processors simultaneously modify entries in their caches for the same memory address?
- How can a cache-coherence protocol be implemented by snooping?
- How can a cache-coherence protocol be implemented using directories?
- What is false sharing (in the reading, but not covered in these slides)?
- Do real machines provide sequential consistency?
- How do these issues influence good software design practice?

Classifying Cache Misses

- **Compulsory:** The first reference to a cache block will cause a miss.
 - ▶ Note that the first access should be a write – otherwise the location is uninitialized.
 - ▶ A cache can avoid stalling the processor by using “allocate on write”.
 - ▶ If a miss is a write, assign a block for the line, start the main memory read, track which bytes have been written, and merge with the data from memory when it arrives.
- **Capacity:** The cache is not big enough to hold all of the data used by the program.
- **Conflict:** Many active memory locations map to the same cache index.
 - ▶ If there are more such references than the associativity of the cache, these will cause conflict misses.
- **Coherence:** A cache block was evicted because another CPU was writing to it.
 - ▶ A subsequent read incurs a cache miss.

Cache Design Trade-Offs (1 of 2)

- **Capacity:** Larger caches have lower miss rates, but longer access times. This motivates using multiple levels of caches.
 - ▶ L1: closest to the CPU, smallest capacity (16-64Kbytes), fastest access (1-3 clock cycles).
 - ▶ L2: typically 128Kbytes to 1Mbyte, 5-10 cycle access time.
 - ▶ L3: becoming common, several Mbytes of capacity.
- **Block Size:**
 - ▶ Larger blocks can lower miss rate by exploiting spatial locality.
 - ▶ Larger blocks can raise miss rate due to conflict and coherence misses.
 - ▶ Larger blocks increase miss penalty by requiring more time to transfer all that data.
 - ▶ Typical block sizes are 16 to 256 bytes – sometimes block size changes with cache level.

Cache Design Trade-Offs (2 of 2)

● **Associativity:**

- ▶ Increasing associativity generally reduces the number of conflict misses.
- ▶ Increasing associativity makes the cache hardware more complicated.
- ▶ Typical caches are direct mapped to four- or eight-way associative.
- ▶ Associativity doesn't need to be a power of two!

● **Other stuff**

- ▶ cache inclusion: is everything in the L1 also in the L2?
- ▶ interaction with virtual memory: are cache addresses virtual or physical?
- ▶ coherence protocol details:
Example, Intel uses MESIF, the “F” stands for “forwarding”. If a processor has a read miss, and another cache has a copy, one of the caches with a copy will be the “forwarding cache”. The forwarding cache provides the data because it's much faster than main memory.
- ▶ error detection and creation – caches + cosmic rays = flipped bits.
- ▶ and all kinds of other optimizations that are beyond the scope of this class.

False Sharing

- False sharing occurs when two CPUs are actively writing different words in the same cache block.
 - ▶ Each write forces the other CPU to invalidate its cache block.
 - ▶ Each read forces the other CPU to change its cache block from `modified` or `exclusive` to `shared`.
- Example: count 3s
 - ▶ Here's an implementation with awful performance.
 - ▶ We create a global array of `ints` to hold the accumulators for each process.
 - ▶ Each time a process finds a `3`, it writes to its element in the array.
 - ▶ This forces the other CPUs whose accumulators are in the same block to invalidate their cache entry.
 - ▶ This turns accumulator accesses into main memory accesses.
 - ▶ And these accesses are serialized: one CPU at a time.