Shared Memory Multiprocessors

- Shared memory multiprocessors
- Shared cache
- Private cache/Dancehall
- Distributed shared memory
- Shared vs. private in CMPs
- Cache coherence
- Cache coherence: Example
- What went wrong?
- Implementations

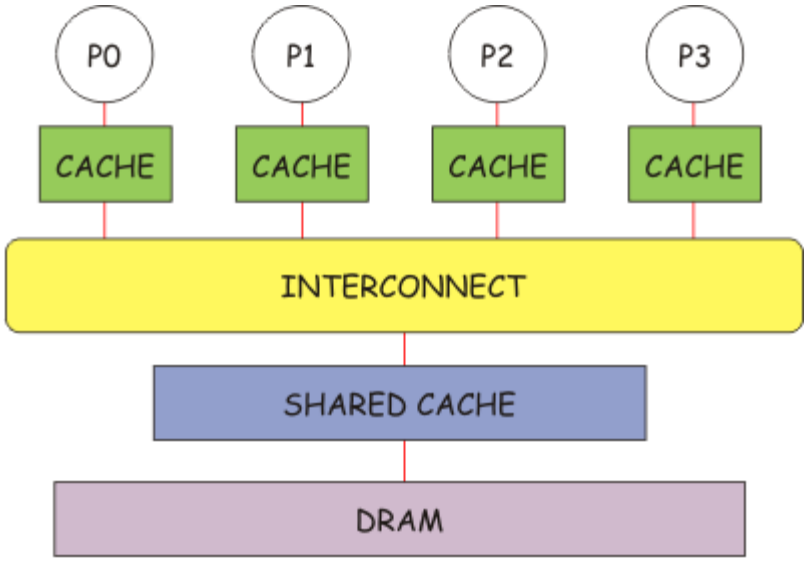
Previous Next

Module 9: "Introduction to Shared Memory Multiprocessors"
Lecture 16: "Multiprocessor Organizations and Cache Coherence"

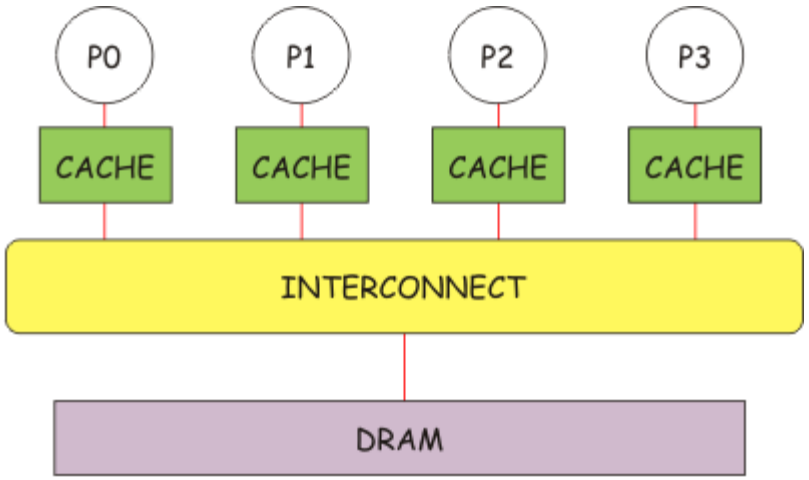
Shared memory multiprocessors

- What do they look like?
  - We will assume that each processor has a hierarchy of caches (possibly shared)
  - We will not discuss shared memory in a time-shared single-thread computer
- A degenerate case of the following
  - Shared cache (popular in CMPs)
  - Private cache (popular in CMPs and SMPs)
  - Dancehall (popular in old computers)
  - Distributed shared memory (popular in medium to large-scale servers)

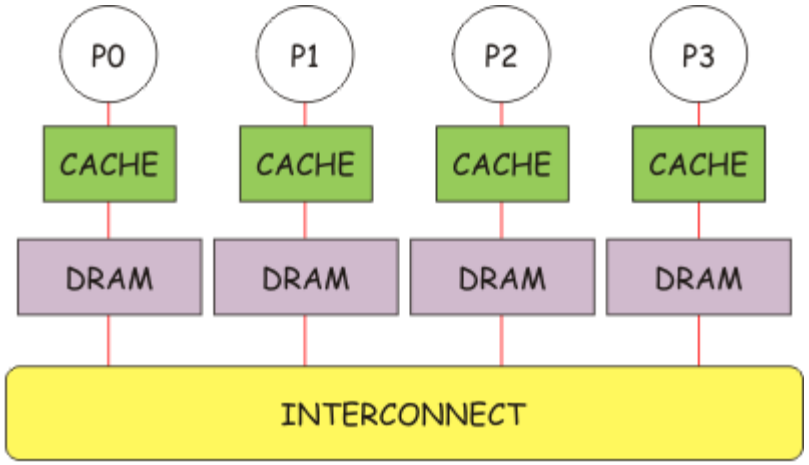
Shared cache



Private cache/Dancehall



Distributed shared memory

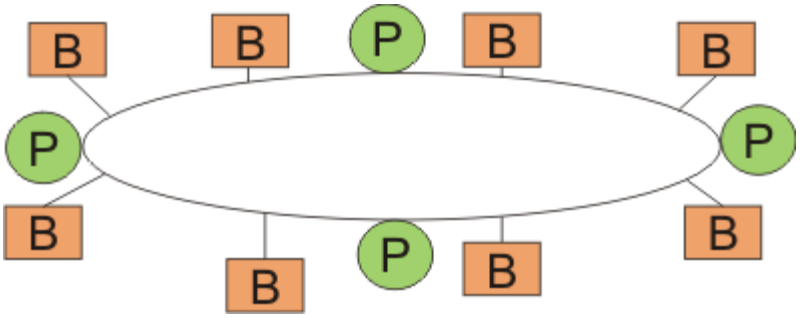


Previous Next

Module 9: "Introduction to Shared Memory Multiprocessors"
Lecture 16: "Multiprocessor Organizations and Cache Coherence"

Shared vs. private in CMPs

- Shared caches are often very large in the CMPs
  - They are banked to avoid worst-case wire delay
  - The banks are usually distributed across the floor of the chip on an interconnect



- In shared caches, getting a block from a remote bank takes time proportional to the physical distance between the requester and the bank
  - Non-uniform cache architecture (NUCA)
- This is same for private caches, if the data resides in a remote cache
- Shared cache may have higher average hit latency than the private cache
  - Hopefully most hits in the latter will be local
- Shared caches are most likely to have less misses than private caches
  - Latter wastes space due to replication

Cache coherence

- Nothing unique to multiprocessors
  - Even uniprocessor computers need to worry about cache coherence
  - For sequential programs we expect a memory location to return the latest value written
  - For concurrent programs running on multiple threads or processes on a single processor we expect the same model to hold because all threads see the same cache hierarchy (same as shared L1 cache)
  - For multiprocessors there remains a danger of using a stale value: hardware must ensure that cached values are coherent across the system and they satisfy programmers' intuitive memory model

Cache coherence: Example

- Assume a write-through cache
  - P0: reads x from memory, puts it in its cache, and gets the value 5
  - P1: reads x from memory, puts it in its cache, and gets the value 5
  - P1: writes x=7, updates its cached value and memory value
  - P0: reads x from its cache and gets the value 5
  - P2: reads x from memory, puts it in its cache, and gets the value 7 (now the system is completely incoherent)
  - P2: writes x=10, updates its cached value and memory value
- Consider the same example with a writeback cache
  - P0 has a cached value 5, P1 has 7, P2 has 10, memory has 5 (since caches are not write through)

- The state of the line in P1 and P2 is M while the line in P0 is clean
- Eviction of the line from P1 and P2 will issue writebacks while eviction of the line from P0 will not issue a writeback (clean lines do not need writeback)
- Suppose P2 evicts the line first, and then P1
- Final memory value is 7: we lost the store x=10 from P2

Previous Next

What went wrong?

- For write through cache
  - The memory value may be correct if the writes are correctly ordered
  - But the system allowed a store to proceed when there is already a cached copy
  - Lesson learned: must invalidate all cached copies before allowing a store to proceed
- Writeback cache
  - Problem is even more complicated: stores are no longer visible to memory immediately
  - Writeback order is important
  - Lesson learned: do not allow more than one copy of a cache line in M state

Implementations

- Must invalidate all cached copies before allowing a store to proceed
  - Need to know where the cached copies are
  - Solution1: Never mind! Just tell everyone that you are going to do a store
    - Leads to broadcast snoopy protocols
    - Popular with small-scale bus-based CMPs and SMPs
    - AMD Opteron implements it on a distributed network (the Hammer protocol)
    - The biggest reason why quotidian Windows fans would buy small-scale multiprocessors and multi-core today
  - Solution2: Keep track of the sharers and invalidate them when needed
    - Where and how is this information stored?
    - Leads to directory-based scalable protocols
- Directory-based protocols
  - Maintain one directory entry per memory block
  - Each directory entry contains a sharer bitvector and state bits
  - Concept of home node in distributed shared memory multiprocessors
  - Concept of sparse directory for on-chip coherence in CMPs
- Do not allow more than one copy of a cache line in M state
  - Need some form of access control mechanism
  - Before a processor does a store it must take “permission” from the current “owner” (if any)
  - Need to know who the current owner is
    - Either a processor or main memory
  - Solution1 and Solution2 apply here also
- Latest value must be propagated to the requester
  - Notion of “latest” is very fuzzy
  - Once we know the owner, this is easy
  - Solution1 and Solution2 apply here also
- Invariant: if a cache block is not in M state in any processor, memory must provide the block to the requester
  - Memory must be updated when a block transitions from M state to S state
  - Note that a transition from M to I always updates memory in systems with writeback caches (these are normal writeback operations)
- Most of the implementations of a coherence protocol deals with uncommon cases and races