Lecture 26: Multiprocessors







Memory Hierarchy Properties

- A virtual memory page can be placed anywhere in physical memory (fully-associative)
- Replacement is usually LRU (since the miss penalty is huge, we can invest some effort to minimize misses)
- A page table (indexed by virtual page number) is used for translating virtual to physical page number
- The page table is itself in memory

- Since the number of pages is very high, the page table capacity is too large to fit on chip
- A translation lookaside buffer (TLB) caches the virtual to physical page number translation for recent accesses
- A TLB miss requires us to access the page table, which may not even be found in the cache – two expensive memory look-ups to access one word of data!
- A large page size can increase the coverage of the TLB and reduce the capacity of the page table, but also increases memory waste



TLB and Cache

- Is the cache indexed with virtual or physical address?
 - ➤ To index with a physical address, we will have to first look up the TLB, then the cache → longer access time
 - Multiple virtual addresses can map to the same physical address – must ensure that these different virtual addresses will map to the same location in cache – else, there will be two different copies of the same physical memory word
- Does the tag array store virtual or physical addresses?
 Since multiple virtual addresses can map to the same physical address, a virtual tag comparison can flag a miss even if the correct physical memory word is present

Cache and TLB Pipeline



Virtually Indexed; Physically Tagged Cache

Bad Events

- Consider the longest latency possible for a load instruction:
 - TLB miss: must look up page table to find translation for v.page P
 - Calculate the virtual memory address for the page table entry that has the translation for page P – let's say, this is v.page Q
 - TLB miss for v.page Q: will require navigation of a hierarchical page table (let's ignore this case for now and assume we have succeeded in finding the physical memory location (R) for page Q)
 - Access memory location R (find this either in L1, L2, or memory)
 - We now have the translation for v.page P put this into the TLB
 - We now have a TLB hit and know the physical page number this allows us to do tag comparison and check the L1 cache for a hit
 - If there's a miss in L1, check L2 if that misses, check in memory
 - At any point, if the page table entry claims that the page is on disk, flag a page fault – the OS then copies the page from disk to memory and the hardware resumes what it was doing before the page fault ... phew!

Multiprocessor Taxonomy

• SISD: single instruction and single data stream: uniprocessor

- MISD: no commercial multiprocessor: imagine data going through a pipeline of execution engines
- SIMD: vector architectures: lower flexibility
- MIMD: most multiprocessors today: easy to construct with off-the-shelf computers, most flexibility

Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)
- Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)
- Shared-memory because all processors can access the entire memory address space
- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors



States: Shored: (Read-only perms) Excl/Modified: Wr and Rd perms Invalid: Nr and Rd Perms Example

- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state RL Miss
- <u>P2 reads X</u>: not found in cache-2, request sent on bus, everyone snoops this request, cache-1does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state (NO



- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops
 - and realizes it has the only valid copy, so it up downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated

13

Example

Request	Cache Hit/Miss	Request on the bus	Who responds	State in Cache 1	State in Cache 2	State in Cache 3	State in Cache 4
				Inv	Inv	Inv	Inv
P1: Rd X	Rd Miss	Rd X	Memory	S	Inv	Inv	Inv
P2: Rd X	Rd Miss	Rd X	Memory	S	539	Inv	Inv
P2: Wr X	Perms Miss	Upgrade X	No response. Other caches invalidate.	Inv	M	7 Inv	Inv
P3: Wr X	Wr Miss	Wr X	P2 responds) Inv	Înv	УМ 59	Inv
P3: Rd X	Rd Hit			Inv	Inv	Μ	Inv
P4: Rd X	Rd Miss	Rd X	P3 responds. Mem wrtbk	Inv	Inv	S	S

granularity st a block G4B

157

59

Cache Coherence Protocols

- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory
- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary
- Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
- Write-update: when a processor writes, it updates other shared copies of that block

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data
- A lock surrounding the data/code ensures that only one program can be in a critical section at a time
- The hardware must provide some basic primitives that allow us to construct locks with different properties



- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write
- Atomic exchange: swap contents of register and memory
- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory (if memory has 0, lock is free)
- lock: t&s register, location
 bnz register, lock
 CS
 - st location, #0

When multiple parallel threads execute this code, only one will be able to enter CS

Coherence Vs. Consistency

- Coherence guarantees (i) write propagation

 (a write will eventually be seen by other processors), and
 (ii) write serialization (all processors see writes to the same location in the same order)
- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Consistency Example

Consider a multiprocessor with bus-based snooping cache coherence

Initially
$$A = B = 0$$
P1P2 $A \leftarrow 1$ $B \leftarrow 1$if $(B == 0)$ if $(A == 0)$ Crit.SectionCrit.Section

Consistency Example

 Consider a multiprocessor with bus-based snooping cache coherence

Initially
$$A = B = 0$$
P1P2 $A \leftarrow 1$ $B \leftarrow 1$if $(B == 0)$ if $(A == 0)$ Crit.SectionCrit.Section

The programmer expected the above code to implement a lock – because of ooo, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion
- The multiprocessor in the previous example is not sequentially consistent
- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

- Sequential consistency is very slow
- The programming complications/surprises are caused when the program has race conditions (two threads dealing with same data and at least one of the threads is modifying the data)
- If programmers are disciplined and enforce mutual exclusion when dealing with shared data, we can allow some re-orderings and higher performance
- This is effective at balancing performance & programming effort