Memory Hierarchies

- Key Principles
  - Locality - most programs do not access code or data uniformly
  - Smaller hardware is faster

- Goal
  - Design a memory hierarchy "with cost almost as low as the cheapest level of the hierarchy and speed almost as fast as the fastest level"
  - This implies that we be clever about keeping more likely used data as "close" to the CPU as possible

- Levels provide subsets
  - Anything (data) found in a particular level is also found in the next level below.
  - Each level maps from a slower, larger memory to a smaller but faster memory

Memory and CPU performance gap since 1980

- Processor-Memory Performance Gap (grows 50% / year)
Propagation delay bounds where memory can be placed

Cache

- Cache is the name given to the first level of the memory hierarchy encountered once an address leaves the CPU.
- Takes advantage of the principle of locality
- The term cache is also now applied whenever buffering is employed to reuse commonly occurring items
- Cache hit – CPU finds a requested data item in the cache
- Cache miss – the item in not in the cache at access
- Block – a fixed size collection of data, retrieved from memory and placed into the cache

What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits spatial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers “a cache” on variables - software managed
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?

Caches: Automatic Management of Fast Storage

- CPU
- L1-Cache
- L2-Cache
- Memory
- Disk, Tape, etc.
- Faster
- Bigger

Main Memory

- CPU
- L2-Cache
- L3-Cache
- Memory
- ~256KB
- ~4MB
- ~10 pclk latency
- ~50 pclk latency
- 16~32KB
- 1~2 pclk latency
Memory Hierarchy

- Placing the fastest memory near the CPU can result in increases in performance
- Now we consider the number of cycles the CPU is stalled waiting for a memory access – memory stall cycles

CPU execution time = (CPU clock cycles + Memory stall cycles) * clock cycle time.

Memory stall cycles = number of misses * miss penalty = IC*(memory accesses/instruction)*miss rate* miss penalty

Cache Performance – Simplified Models

- Hit ratio r = number of requests that are hits/total num requests
- Cost of memory access= rC_h + (1-r) C_m
  - C_h is cost/time from cache, C_m is cost/time when miss – fetch from memory
- Extend to multiple levels
  - Hit ratios for level 1, level 2, etc.
  - Access times for level 1, level 2, etc.
  - \( r_1 C_{h1} + r_2 C_{h2} + (1-r_1 -r_2) C_m \)

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- CPI = ideal CPI + average stalls per instruction
  \[ 1.1 \text{(cycles/ins)} + \left[ 0.30 \left( \text{DataMops/ins} \right) \times 0.10 \left( \text{miss/DataMop} \right) \times 50 \left( \text{cycle/miss} \right) \right] + \left[ 1 \left( \text{InstMop/ins} \right) \times 0.01 \left( \text{miss/InstMop} \right) \times 50 \left( \text{cycle/miss} \right) \right] \]
  = (1.1 + 1.5 + 0.5) cycle/ins = 3.1
- 58% of the time the proc is stalled waiting for memory!
- AMAT=(1/1.3)(1+0.01x50)+(0.3/1.3)(1-0.1x50)=2.54
Memory Hierarchy

• Q1: Where can a block be placed in the upper level? (block placement)
• Q2: How is a block found if it is in the upper level? (block identification)
• Q3: Which block should be replaced on a miss? (block replacement)
• Q4: What happens on a write? (Write strategy)

Definitions

• Locating a block requires two attributes:
  – Size of block
  – Organization of blocks within the cache
• Block size (also referred to as line size)
  – Granularity at which cache operates
  – Each block is contiguous series of bytes in memory and begins on a naturally aligned boundary
  – Eg: cache with 16 byte blocks
    • each contains 16 bytes
    • First byte aligned to 16 byte boundaries in address space
      – Low order 4 bits of address of first byte would be 0000
    – Smallest usable block size is the natural word size of the processor
    • Else would require splitting an access across blocks and slows down translation

Where can a block be placed in a cache?-Cache Organization

• If each block has only one place it can appear in the cache, it is said to be "direct mapped" and the mapping is usually (Block address) MOD (Number of blocks in the cache)
• If a block can be placed anywhere in the cache, it is said to be fully associative
• If a block can be placed in a restrictive set of places in the cache, the cache is set associate. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. (Block address) MOD (Number of sets in the cache) if there are n blocks in a set, the cache is called \( n \)-way set associative

Alternatives for 8 block frame cache with a 32 block memory
Cache Memory Structures

Indexed Memory
- k-bit index
- $2^k$ blocks

Associative Memory (CAM)
- no index
- unlimited blocks

N-Way Set-Associative Memory
- k-bit index
- $2^k \cdot N$ blocks

Direct Mapped Caches

Cache Block Size
- Each cache block or (cache line) has only one tag but can hold multiple "chunks" of data
  - reduce tag storage overhead
    - In 32-bit addressing, an 1-MB direct-mapped cache has 12 bits of tags
    - 4-byte cache block $\Rightarrow$ 256K blocks $\Rightarrow$ ~384KB of tag
    - 128-byte cache block $\Rightarrow$ 8K blocks $\Rightarrow$ ~12KB of tag
    - the entire cache block is transferred to and from memory all at once
      good for spatial locality since if you access address i, you will probably want i+1 as well (prefetching effect)
- Block size = $2^b$, Direct Mapped Cache Size = $2^{b+b}$

Fully Associative Cache

N-Way Set Associative Cache

Cache Size = $N \times 2^{B+b}$

Cache Organizations

- Direct Mapped vs Fully Associate
  - Direct mapped is not flexible enough; if $X \equiv Y \pmod{K}$ then $X$ and $Y$ cannot both be located in cache
  - Fully associative allows any mapping, implies all locations must be searched to find the right one – expensive hardware
- Set Associative
  - Compromise between direct mapped and fully associative
  - Allow many-to-few mappings
  - On lookup, subset of address bits used to generate an index
  - BUT index now corresponds to a set of entries which can be searched in parallel – more efficient hardware implementation than fully associative

Large Blocks and Subblocking

- Large cache blocks can take a long time to refill
  - refill cache line critical word first
  - restart cache access before complete refill
- Large cache blocks can waste bus bandwidth if block size is larger than spatial locality
  - divide a block into subblocks
  - associate separate valid bits for each subblock.
How is a block found if it is in the cache

• Since we have many-to-one mappings, need tag
• Caches have an address tag on each block that gives the block address.
  – Eg: if slot zero in cache contains tag K, the value in slot zero corresponds to block zero from area of memory that has tag K
  – Address consists of <tag t, block b, offset o>
    • Examine tag in slot b of cache:
      – if matches t then extract value from slot b in cache
      – Else use memory address to fetch block from memory, place copy in slot b of cache, replace tag with t, use o to select appropriate byte
• The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the CPU – all possible tags are searched in parallel
• A valid bit may be added to the cache to indicate whether or not this entry contains a valid address.

Address in a set associative or direct mapped cache – tag gives the block number, index is used to select the set

<table>
<thead>
<tr>
<th>Block address</th>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2003 Elsevier Science (USA). All rights reserved.

Fully associative caches have no index field.

Which block should be replaced?

• Random – candidate blocks randomly selected
• LRU – accesses to blocks are recorded, the block replaced is the one unused for the longest time
• FIFO – approximates LRU by throwing out oldest block

What happens on write?

• Write through – information written to both the block in the cache and to the block in lower level memory
• Write back – The information is written only to the block in the cache, memory is written only when block is replaced
• A dirty bit is kept in the block description area
Cache performance

- Average memory access time = hit time + miss rate * miss penalty
- CPU time = (CPU execution clock cycles + Memory stall clock cycles) * Clock cycle time
- CPU’s with a low CPI and high clock rates will be significantly impacted by cache rates (example page 409)

Next -- How to Improve Cache Performance?

\[ AMAT = HitTime + MissRate \times MissPenalty \]

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

Improving Performance: Reducing Cache Miss Penalty

- Multilevel caches – second and subsequent level caches can be large enough to capture many accesses that would have gone to main memory, and are faster (therefore less penalty)
- Critical word first and early restart – don’t wait for full block of cache to be loaded, send the critical word first, restart the CPU and continue the load
- Priority to read misses over write misses
- Merging write buffer – if the address of a new entry matches that of one in the write buffer, combine it
- Victim Caches – cache discarded blocks elsewhere

Miss Rates (Multilevel Caches)

- Local miss rate – the number of misses in a cache divided by the total number of memory accesses to the cache
- Global miss rate – the number of misses in the cache divided by the total number of memory accesses generated by the CPU
Miss rates versus cache size for multilevel caches

Relative execution time by second level cache size

Bottom write buffer uses write merging while top does not

Use of victim cache
Reducing Miss Rate – the classical approach to improving cache behavior

• Categories of misses
  – Compulsory - the very first access to a block cannot be in the cache, so the block must be brought into the cache. (Cold start misses)
  – Capacity - if the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
  – Conflict - if the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. Interference or collision misses

Miss Rate Reduction Strategies

• Increase block size – reduce compulsory misses
• Larger caches
• Higher associativity
• Pseudo-associative caches (Way prediction)
• Compiler controlled pre-fetching (faulting/non-faulting)
  – Code reorganization
    • Merging arrays
    • Loop interchange (row column order)
    • Loop fusion (2 loops into 1)
    • Blocking
What else drives up block size?

Reducing Misses via “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)

- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache, similar in UltraSPARC
Reducing Misses by **Hardware** Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in "stream buffer"
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty

Reducing Misses by **Compiler Optimizations**

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

---

### Merging Arrays Example

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];

Reducing conflicts between val & key; improve spatial locality

### Loop Interchange Example

/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
      x[i][j] = 2 * x[i][j];

/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];

Sequential accesses instead of striding through memory every 100 words; improved spatial locality
**Loop Fusion Example**

// Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    { a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}

2 misses per access to a & c vs. one miss per access; improve spatial locality

**Blocking Example**

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    r = 0;
    for (k = 0; k < N; k = k+1)
      r = r + y[i][k]*z[k][j];
    x[i][j] = r;

/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); j = j+1)
        { r = 0;
          for (k = kk; k < min(kk+B-1,N); k = k+1)
            r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j] + r;}

• B called Blocking Factor
• Capacity Misses from $2N^2$ to $N^2/B+2N^2$
• Conflict Misses Too?

**Summary of Compiler Optimizations to Reduce Cache Misses (by hand)**

- vprina (nasa7)
- gnty (nasa7)
- tomatcv
- btrix (nasa7)
- mxml (nasa7)
- spice
- cholesky
- compress (nasa7)

Performance Improvement

- merged arrays
- loop interchange
- loop fusion
- blocking

1 1.5 2 2.5 3
Main Memory Organizations for Improving Performance

- Wider Main Memory
- Simple Interleaved Memory
- Independent Memory Banks

Main Memory Organizations

- **Simple:**
  - CPU, Cache, Bus, Memory same width (32 or 64 bits)
- **Wide:**
  - CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UltraSPARC: 512)
- **Interleaved:**
  - CPU, Cache, Bus 1 word; Memory N Modules (4 Modules); example is word interleaved

Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses
  - Multiprocessor
  - I/O
  - CPU with Hit under n Misses, Non-blocking Cache
- **Superbank:** all memory active on one block transfer (or Page)
- **Bank:** portion within a superbank that is word interleaved (or Subbank)

Virtual Memory

- Shares a smaller amount of physical memory among processes which can have a logical address space much larger than the physical memory itself
  - Must provide for dynamic relocation of the address space rather than moving portions of memory or readdressing object code
- Must inherently provide a protection scheme to maintain process integrity
Logical Program

Address Translation

Speeding up Address Translation, the Translation Look-aside Buffer (TLB)

The Problem of Multiprocessor Cache Coherency
What Does Coherency Mean?

• Informally:
  – “Any read must return the most recent write”
  – Too strict and too difficult to implement
• Better:
  – “Any write must eventually be seen by a read”
  – All writes are seen in proper order (“serialization”)
• Two rules to ensure this:
  – “If P writes x and P1 reads it, P’s write will be seen by P1 if the read and write are sufficiently far apart”
  – Writes to a single location are serialized: seen in one order
    • Latest write will be seen
    • Otherwise could see writes in illogical order (could see older value after a newer value)

Cache Coherency Solutions

• Snooping Solution (Snoopy Bus)
• Directory-Based Schemes

Potential HW Coherence Solutions

• Snooping Solution (Snoopy Bus):
  – Send all requests for data to all processors
  – Processors snoop to see if they have a copy and respond accordingly
  – Requires broadcast, since caching information is at processors
  – Works well with bus (natural broadcast medium)
  – Dominates for small scale machines (most of the market)
• Directory-Based Schemes (discuss later)
  – Keep track of what is being shared in 1 centralized place (logically)
  – Distributed memory => distributed directory for scalability
    (avoids bottlenecks)
  – Send point-to-point requests to processors via network
  – Scales better than Snooping
  – Actually existed BEFORE Snooping-based schemes

Bus Snooping Topology

• Memory: centralized with uniform access time (“una”) and bus interconnect
• Examples: Sun Enterprise 5000, SGI Challenge, Intel System Pro
Basic Snoopy Protocols
(Review slides 5 to 21)

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and invalidate any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and update any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared: block can be read
  - OR Exclusive: cache has only copy, its writeable, and dirty
  - OR Invalid: block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses
Snoopy-Cache State Machine-II

- State machine for bus requests for each cache block
- Appendix E7 gives details of bus requests

Invalid → Write miss for this block → Shared (read-only)
Write miss for this block → Write Back Block (abort memory access)
Read miss for this block → Write Back Block (abort memory access)
Exclusive (read/write)

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>P2: Read A1</td>
<td>P2: Write 20 to A1</td>
<td>P2: Write 40 to A2</td>
</tr>
<tr>
<td>Processor 1</td>
<td>Processor 2</td>
<td>Bus</td>
<td>Memory</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>P2: Read A1</td>
<td>P2: Write 20 to A1</td>
<td>P2: Write 40 to A2</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2

Snoopy-Cache State Machine-III

- State machine for CPU requests for each cache block and for bus requests for each cache block

Invalid → Write miss for this block → CPU Read Miss on bus → CPU Read hit
Write miss for this block → Place Write Miss on bus
Write Back Block: (abort memory access)
Shared (read-only)

CPU Read miss

Example: Step 1

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>P2: Read A1</td>
<td>P2: Write 20 to A1</td>
<td>P2: Write 40 to A2</td>
</tr>
<tr>
<td>Processor 1</td>
<td>Processor 2</td>
<td>Bus</td>
<td>Memory</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>P2: Read A1</td>
<td>P2: Write 20 to A1</td>
<td>P2: Write 40 to A2</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2

Active arrow =
Example: Step 2

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

Example: Step 3

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: Write A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

Example: Step 4

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: Write A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

Example: Step 5

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: Write A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.
Snooping Cache Variations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Exclusive</td>
<td>Owned Exclusive</td>
<td>Private Dirty</td>
<td>Modified (private, ≠ Memory)</td>
</tr>
<tr>
<td>Shared</td>
<td>Owned Shared</td>
<td>Private Clean</td>
<td>Exclusive (private, ≠ Memory)</td>
</tr>
<tr>
<td>Invalid</td>
<td>Shared</td>
<td>Shared</td>
<td>Shared</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Owner can update via bus invalidate operation
Owner must write back when replaced in cache
If read sourced from memory, then Private Clean
If read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

Implementation Complications

- **Write Races:**
  - Cannot update cache until bus is obtained
  - Otherwise, another processor may get bus first, and then write the same cache block.
- **Two step process:**
  - Arbitrate for bus
  - Place miss on bus and complete operation
- If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
- Split transaction bus:
  - Bus transaction is not atomic; can have multiple outstanding transactions for a block
  - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
  - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU cache access:
  - solution 1: duplicate set of tags for L1 caches, just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
  - block size, associativity of L2 affects L1
Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is \( f(\text{memory size}) \) vs. \( f(\text{cache size}) \)
- Prevent directory as bottleneck?
  - distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: \( \geq 1 \) processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data
  - \( \Rightarrow \) write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent
Directory Protocol

- No bus and don’t want to broadcast:
  - Interconnect no longer single arbitration point
  - All messages have explicit responses
- Terms: Typically 3 processors involved
  - Local node: Where a request originates
  - Home node: Where the memory location of an address resides
  - Remote node: Has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  \( P = \) processor number, \( A = \) address

Another MP Issue:
Memory Consistency Models

- What is consistency? What must a processor see the new value? E.g., seems that
  \( P_1: A = 0; P_2: B = 0; \ldots A = 1; B = 1; \ldots \)
- What if write invalidate is delayed & processor continues?
- Memory consistency models:
  - What are the rules for such cases?
  - Sequential consistency: Result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: Delay all memory accesses until all invalidates done

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations
    \( \text{write}(x) \)
    \( \ldots \)
    \( \text{release}(s) [/unlock] \)
    \( \ldots \)
    \( \text{acquire}(s) [/lock] \)
    \( \ldots \)
    \( \text{read}(x) \)
- Only those programs willing to be nondeterministic are not synchronized: “data race”; outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards RAR, WAR, RAW, WAW to different addresses

Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => Scalable shared address multiprocessor => Cache coherent, non uniform memory access