CSC 631: High-Performance Computer Architecture

Spring 2017
Lecture 10: Memory

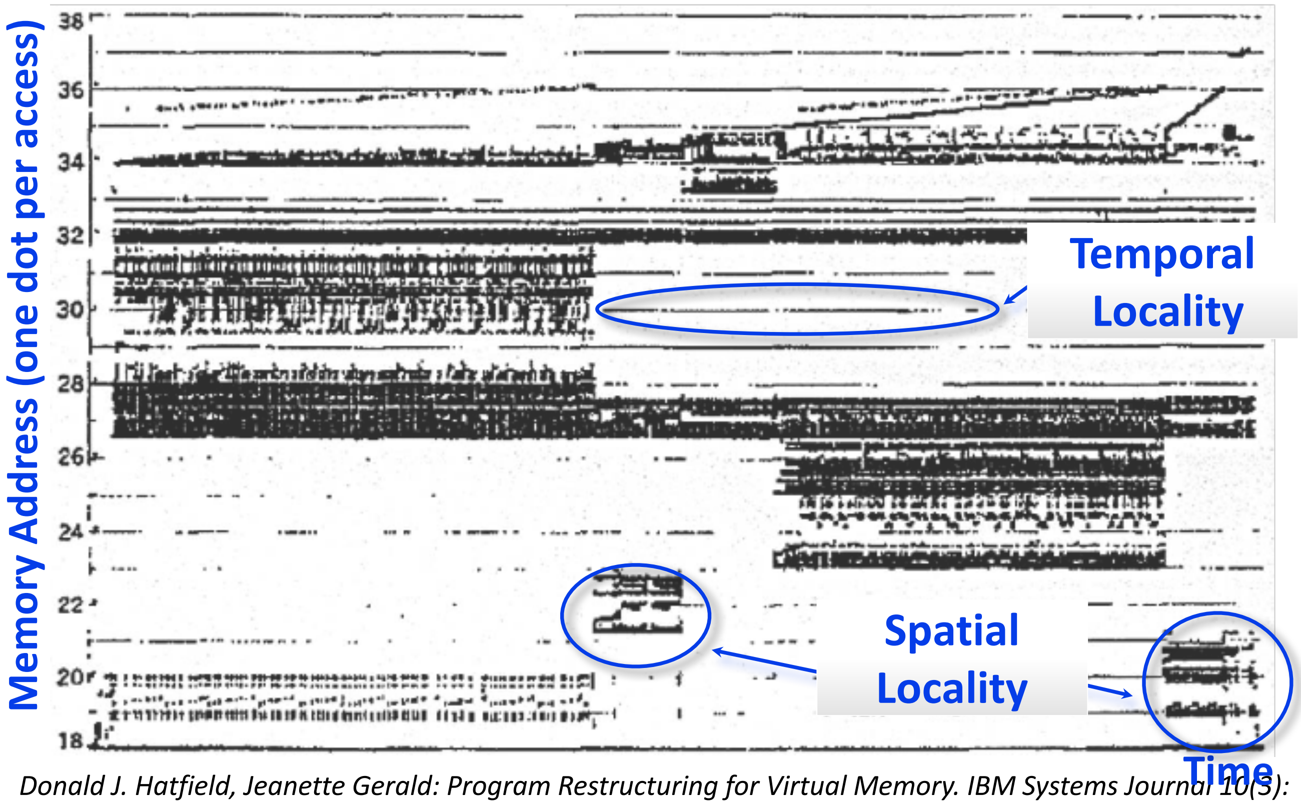
Part II

Two predictable properties of memory references:

- Temporal Locality: If a location is referenced it is likely to be referenced again in the near future.

- Spatial Locality: If a location is referenced it is likely that locations near it will be referenced in the near future.

Memory Reference Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory Hierarchy

- Small, fast memory near processor to buffer accesses to big, slow memory
  - Make combination look like a big, fast memory
- Keep recently accessed data in small fast memory closer to processor to exploit temporal locality
  - Cache replacement policy favors recently accessed data
- Fetch words around requested word to exploit spatial locality
  - Use multiword cache lines, and prefetching

Management of Memory Hierarchy

- Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software's back, e.g., stack management, register renaming
- Larger/slower storage, e.g., main memory
  - Address usually computed from values in register
  - Generally implemented as a hardware-managed cache hierarchy (hardware decides what is kept in fast memory)
    - but software may provide "hints", e.g., don't cache or prefetch

Important Cache Parameters (Review)

- Capacity (in bytes)
- Associativity (from direct-mapped to fully associative)
- Line size (bytes sharing a tag)
- Write-back versus write-through
- Write-allocate versus write no-allocate
- Replacement policy (least recently used, random)

Improving Cache Performance

Average memory access time (AMAT) =
Hit time + Miss rate x Miss penalty

To improve performance:

- reduce the hit time
- reduce the miss rate
- reduce the miss penalty

What is best cache design for 5-stage pipeline?

Biggest cache that doesn't increase hit time past 1 cycle (approx 8-32KB in modern technology)

[design issues more complex with deeper pipelines and/or out-of-order superscalar processors]

Causes of Cache Misses: The 3 C's

- **Compulsory:** first reference to a line (a.k.a. cold start misses)
  - misses that would occur even with infinite cache
- **Capacity:** cache is too small to hold all data needed by the program
  - misses that would occur even under perfect replacement policy
- **Conflict:** misses that occur because of collisions due to line-placement strategy
  - misses that would not occur with ideal full associativity

Effect of Cache Parameters on Performance

- Larger cache size

  + reduces capacity and conflict misses

  - hit time will increase

- Higher associativity

  + reduces conflict misses

  - may increase hit time

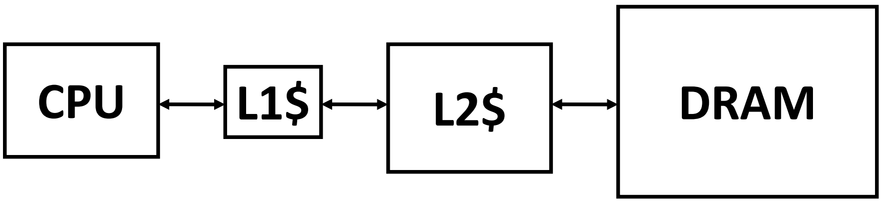
- Larger line size

  + reduces compulsory misses

  - increases conflict misses and miss penalty

Multilevel Caches

- Problem: A memory cannot be large and fast
- Solution: Increasing sizes of cache at each level



CPU L1$ L2$ DRAM

Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions

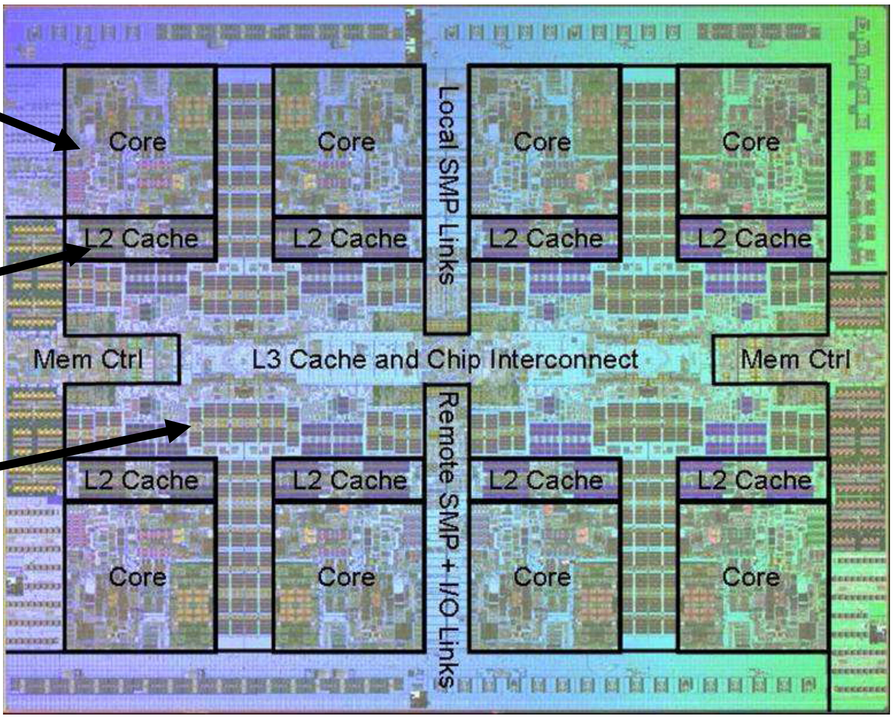
Presence of L2 influences L1 design

- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time
  - Backup L2 reduces L1 miss penalty
  - Reduces average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cache absorbs write traffic, doesn’t go off-chip
  - At most one L1 miss request per L1 access (no dirty victim write back) simplifies pipeline control
  - Simplifies coherence issues
  - Simplifies error recovery in L1 (can use just parity bits in L1 and reload from L2 when parity error detected on L1 read)

Inclusion Policy

- Inclusive multilevel cache:
  - Inner cache can only hold lines also present in outer cache
  - External coherence snoop access need only check outer cache
- Exclusive multilevel caches:
  - Inner cache may hold lines not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache
- Why choose one type or the other?

Power 7 On-Chip Caches [IBM 2009]



32KB L1 I$/core
32KB L1 D$/core
3-cycle latency

256KB Unified L2$/core
8-cycle latency

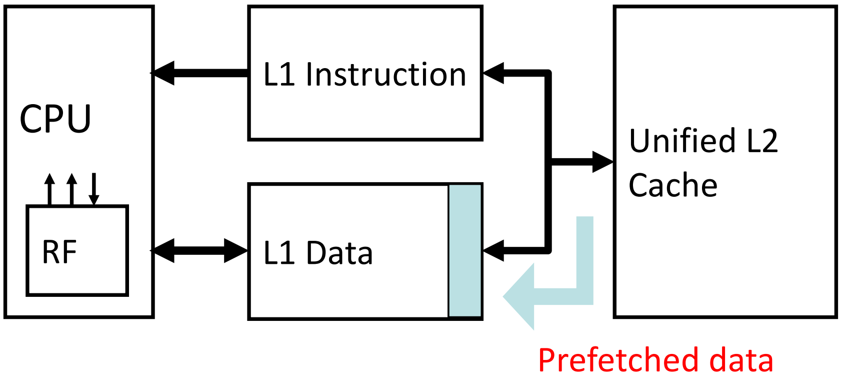
32MB Unified Shared L3$
Embedded DRAM (eDRAM)
25-cycle latency to local slice

Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- What types of misses does prefetching affect?

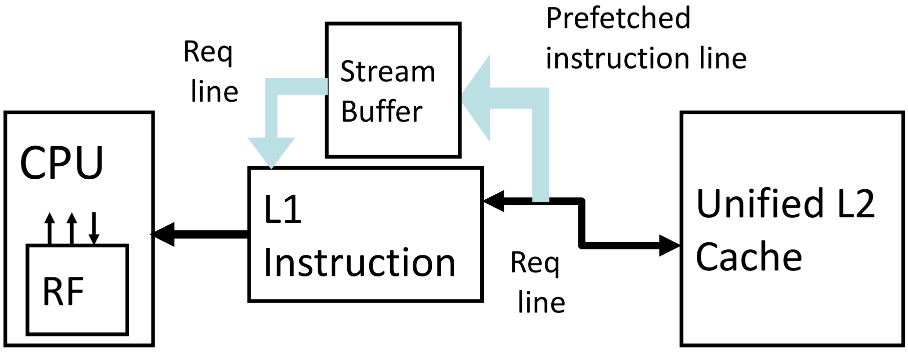
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



Hardware Instruction Prefetching

- Instruction prefetch in Alpha AXP 21064
  - Fetch two lines on a miss; the requested line (i) and the next consecutive line (i+1)
  - Requested line placed in cache, and next line in instruction stream buffer
  - If miss in cache but hit in stream buffer, move stream buffer line into cache and prefetch next line (i+2)



Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch b + 1 upon miss on b
- One-Block Lookahead (OBL) scheme
  - Initiate prefetch for block b + 1 when block b is accessed
  - Why is this different from doubling block size?
  - Can extend to N-block lookahead
- Strided prefetch
  - If observe sequence of accesses to line b, b+N, b+2N, then prefetch b+3N etc.

- Example: IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
}
```

Software Prefetching Issues

- Timing is the biggest issue, not predictability
  - If you prefetch very close to when the data is required, you might be too late
  - Prefetch too early, cause pollution
  - Estimate how long it will take for the data to come into L1, so we can set P appropriately
  - Why is this hard to do?

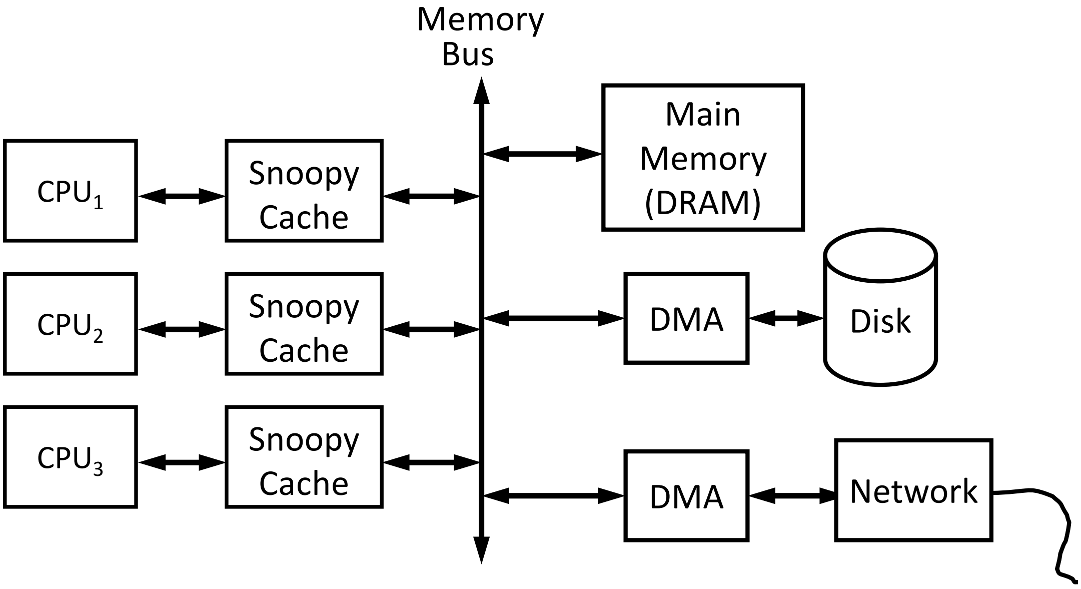
```
for(i=0; i < N; i++) {
   prefetch( &a[i + P] );
   prefetch( &b[i + P] );
   SUM = SUM + a[i] * b[i];
}
```

Must consider cost of prefetch instructions

Compiler Optimizations

- Restructuring code affects the data access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data (“no-allocate” instruction hints or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations

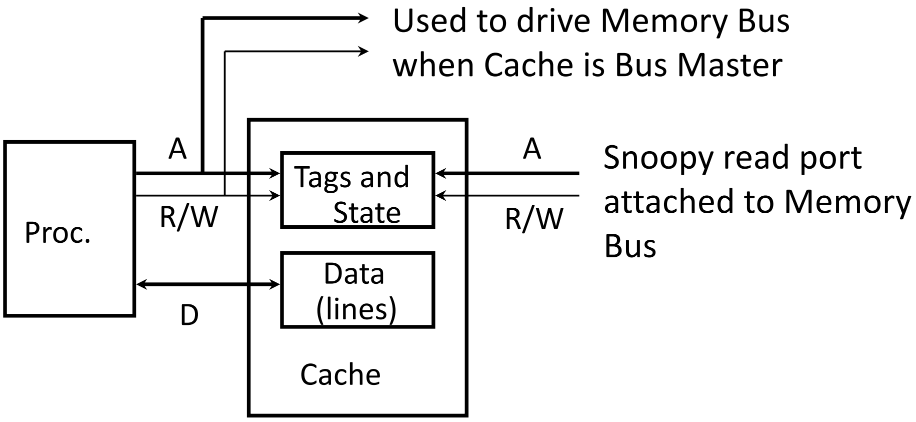
Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
- Snoopy cache tags are dual-ported


Used to drive Memory Bus when Cache is Bus Master
Proc.
A
R/W
D
Tags and State
Data (lines)
Cache
A
R/W
Snoopy read port attached to Memory Bus

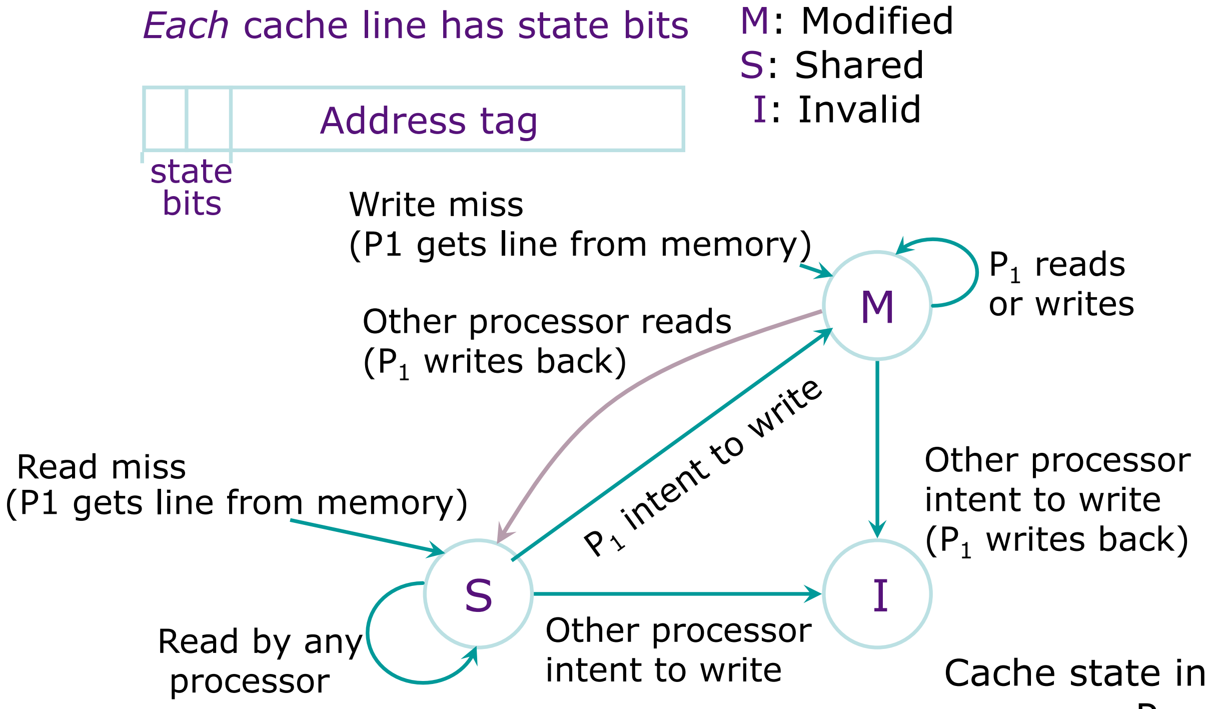
Snoopy Cache Coherence Protocols

- Write miss:
  - the address is invalidated in all other caches before the write is performed
- Read miss:
  - if a dirty copy is found in some cache, a write-back is performed before the memory is read

Cache State Transition Diagram

The MSI protocol

Each cache line has state bits

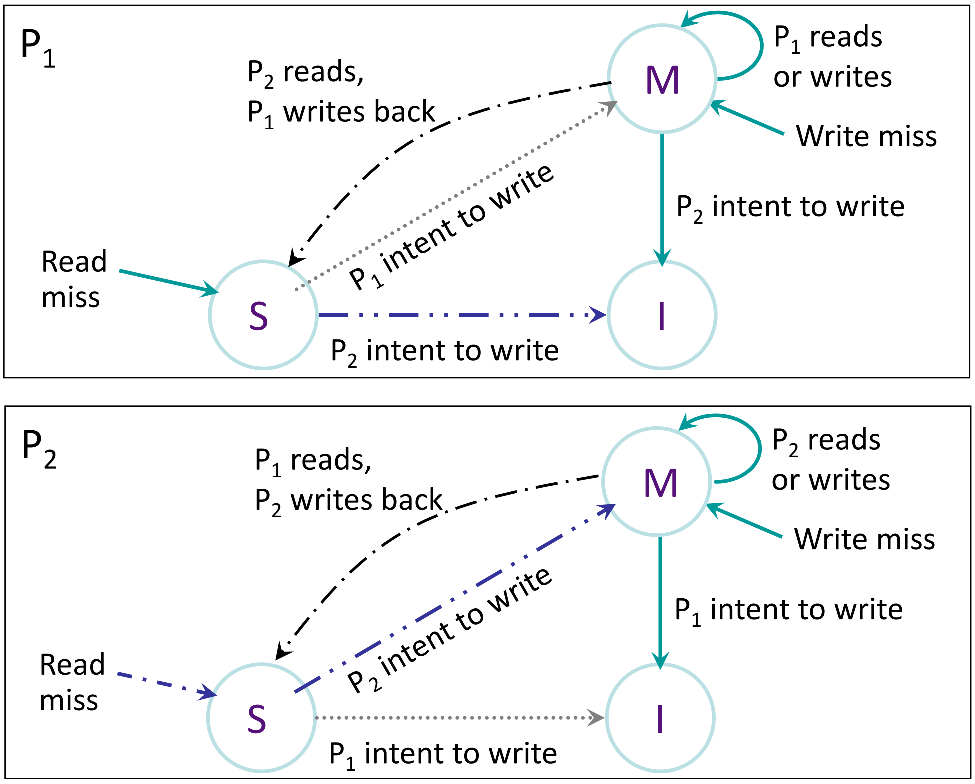

M: Modified
S: Shared
I: Invalid

Address tag
state bits
Write miss (P1 gets line from memory)
M
P_1 reads or writes
Other processor reads (P_1 writes back)
P_1 intent to write
Read miss (P1 gets line from memory)
Other processor intent to write (P_1 writes back)
S
I
Read by any processor
Other processor intent to write

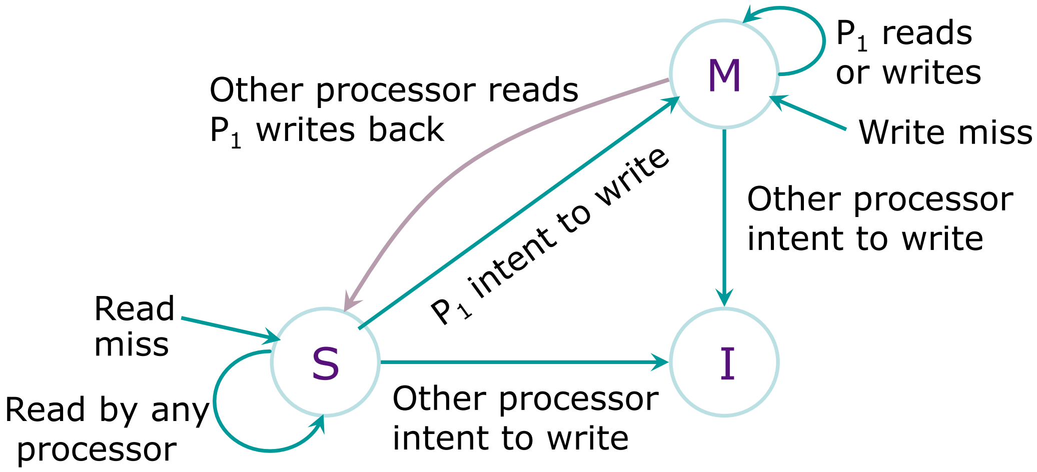
Cache state in processor P_1

Two Processor Example (Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes



Observation

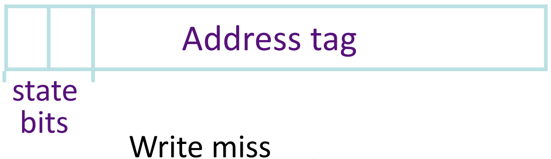


- If a line is in the M state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

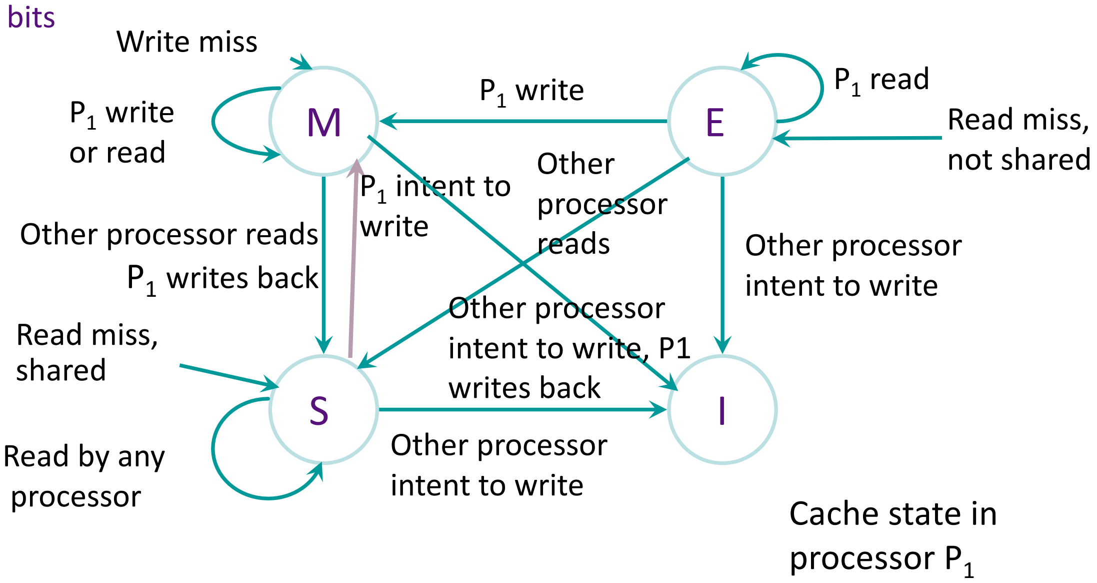
MESI: An Enhanced MSI protocol

increased performance for private data

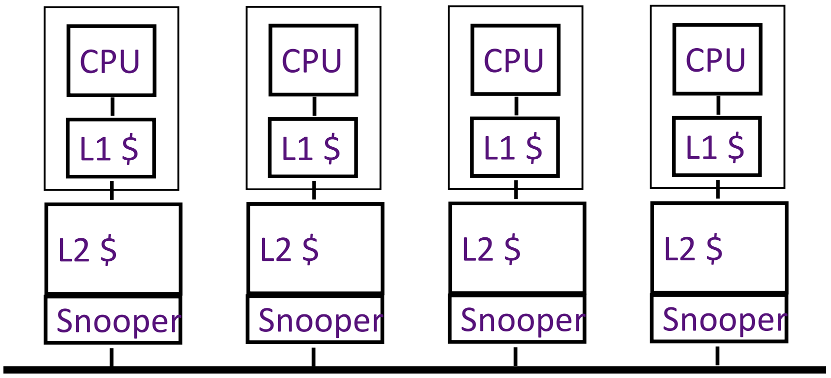
Each cache line has a tag



M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid

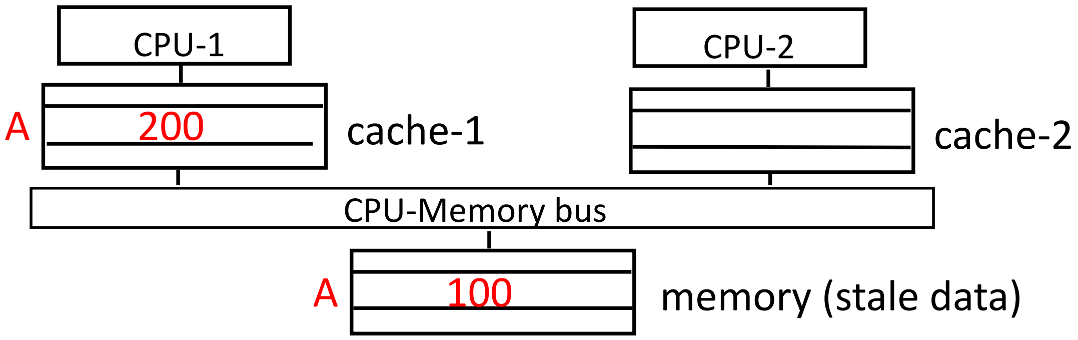


Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 ⇒ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Intervention



- When a read-miss for A occurs in cache-2,
- a read request for A is placed on the bus
  - Cache-1 needs to supply & change its state to shared
  - The memory may respond to the request also!
- *Does memory know it has stale data?*
- Cache-1 needs to intervene through memory controller to supply correct data to cache-2

False Sharing

state	line addr	data0	data1	...	dataN

- A cache line contains more than one word
- Cache-coherence is done at the line-level and not word-level
- Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and
- both words have the same line address.
- *What can happen?*

Performance of Symmetric Multiprocessors (SMPs)

- Cache performance is combination of:
- Uniprocessor cache miss traffic
- Traffic caused by communication
  - Results in invalidations and subsequent cache misses
- Coherence misses
  - Sometimes called a Communication miss
  - 4th C of cache misses along with Compulsory, Capacity, & Conflict.

Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
  - Invalidates due to 1st write to shared line
  - Reads by another CPU of modified line in different cache
  - Miss would still occur if line size were 1 word
- False sharing misses when a line is invalidated because some word in the line, other than the one being read, is written into
  - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
  - Line is shared, but no word in line is actually shared ⇒ miss would not occur if line size were 1 word

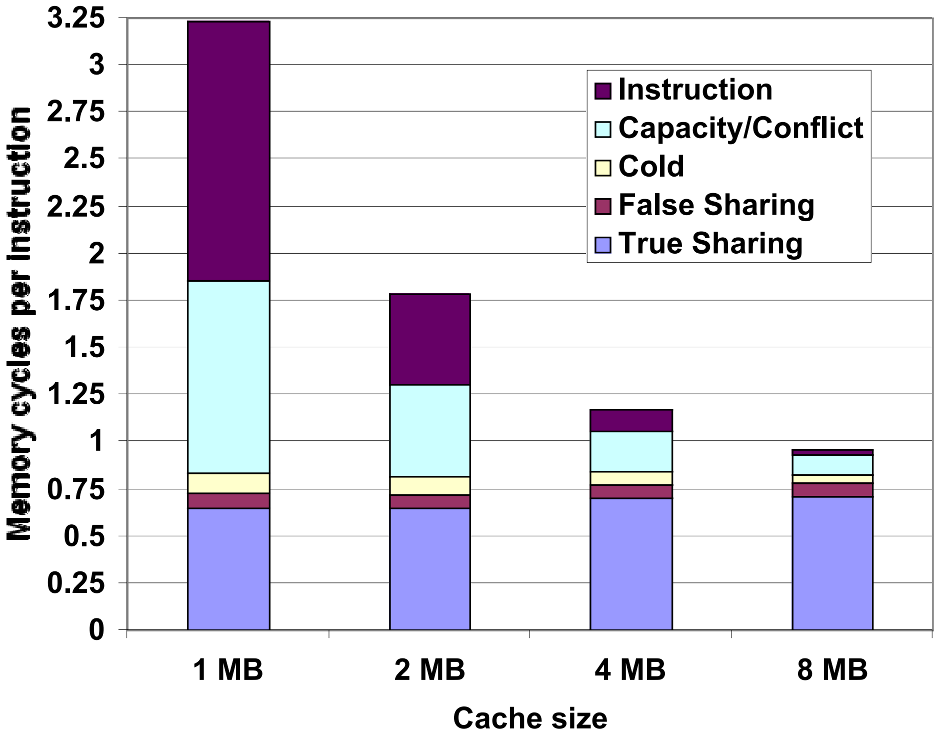
Example: True v. False Sharing v. Hit?

- Assume x1 and x2 in same cache line.
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	True miss; x2 not writeable
5	Read x2		True miss; invalidate x2 in P1

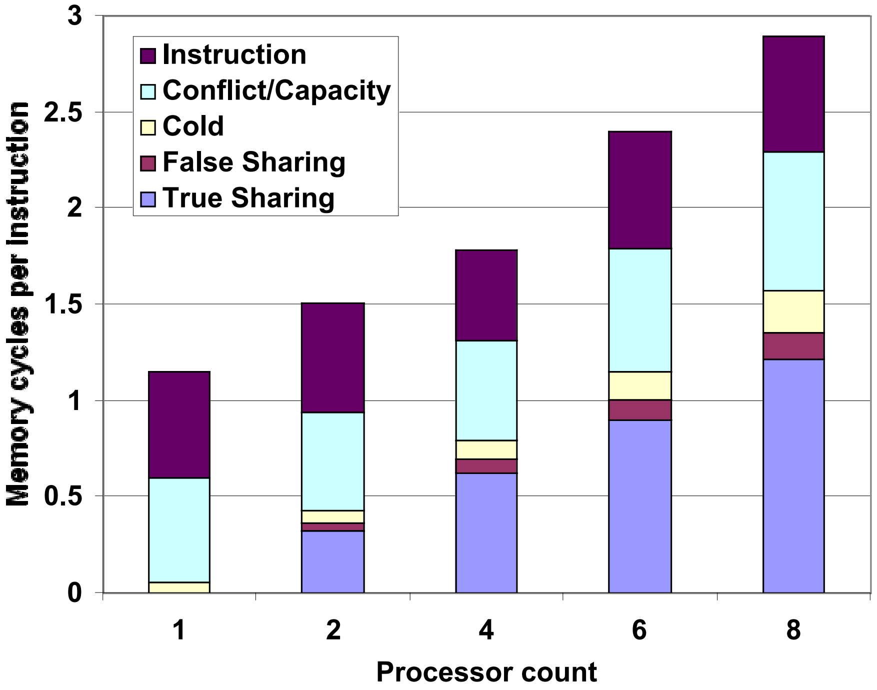
MP Performance 4-Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



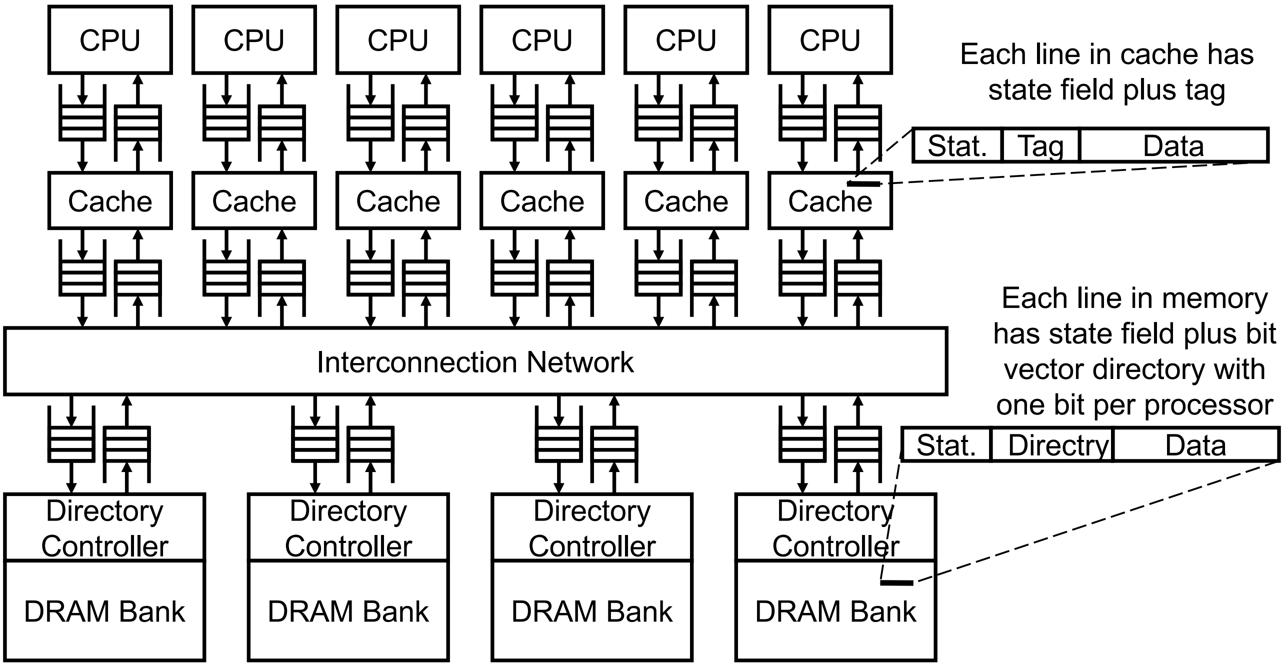
Scaling Snoopy/Broadcast Coherence

- When any processor gets a miss, must probe every other cache
- Scaling up to more processors limited by:
  - Communication bandwidth over bus
  - Snoop bandwidth into tags
- Can improve bandwidth by using multiple interleaved buses with interleaved tag banks
  - E.g, two bits of address pick which of four buses and four tag banks to use – (e.g., bits 7:6 of address pick bus/tag bank, bits 5:0 pick byte in 64-byte line)
- Buses don’t scale to large number of connections, so can use point-to-point network for larger number of nodes, but then limited by tag bandwidth when broadcasting snoop requests.
- Insight: Most snoops fail to find a match!

Scalable Approach: Directories

- Every memory line has associated directory information
  - keeps track of copies of cached lines and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Directory Cache Protocol



Each line in cache has state field plus tag

Each line in memory has state field plus bit vector directory with one bit per processor

- Assumptions: Reliable network, FIFO message delivery between any given source-destination pair

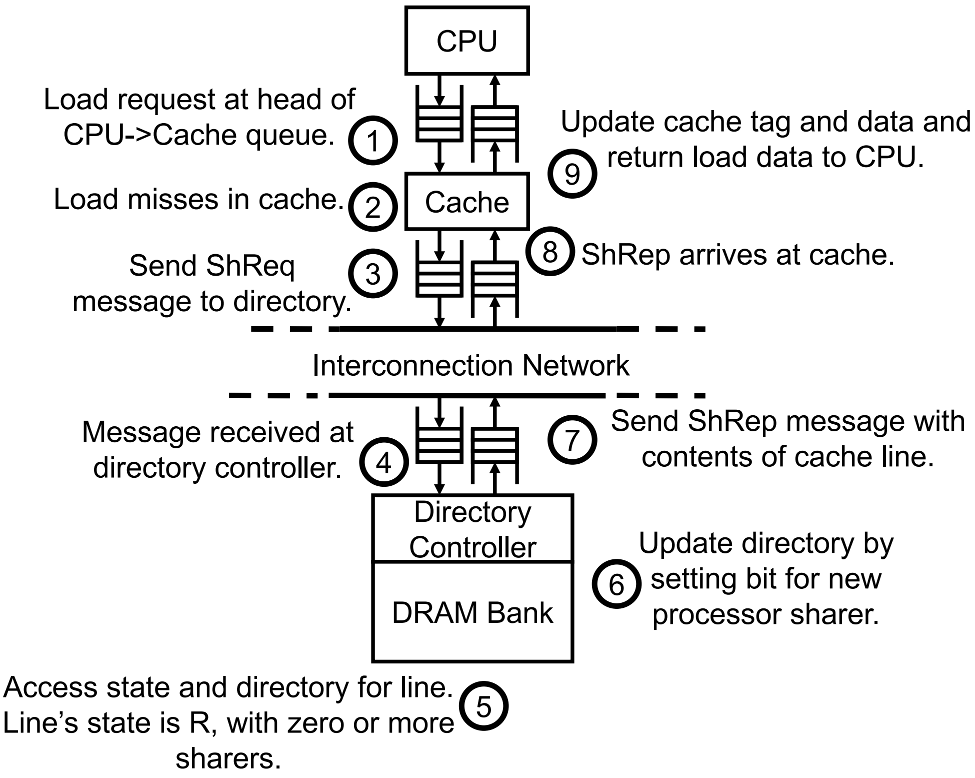
Cache States

- For each cache line, there are 4 possible states:
  - C-invalid (= Nothing): The accessed data is not resident in the cache.
  - C-shared (= Sh): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
  - C-modified (= Ex): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
  - C-transient (= Pending): The accessed data is in a transient state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states

- For each memory line, there are 4 possible states:
  - R(dir): The memory line is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., dir = ε), the memory line is not cached by any site.
  - W(id): The memory line is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.
  - TR(dir): The memory line is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
  - TW(id): The memory line is in a transient state waiting for a line exclusively cached at site id (i.e., in C-modified state) to make the memory line at the home site up-to-date.

Read miss, to uncached or shared line



1 Load request at head of CPU->Cache queue.

2 Load misses in cache.

3 Send ShReq message to directory.

4 Message received at directory controller.

5 Access state and directory for line. Line’s state is R, with zero or more sharers.

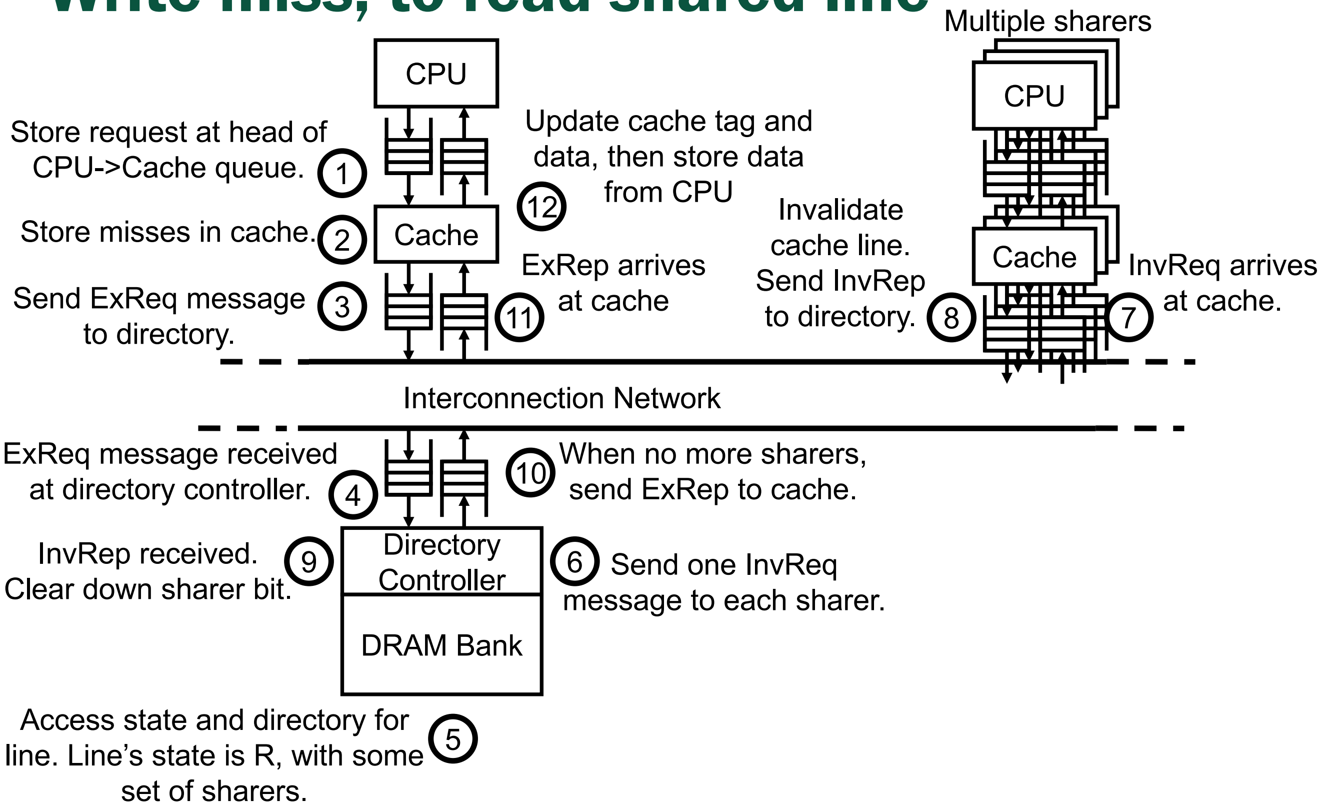
6 Update directory by setting bit for new processor sharer.

7 Send ShRep message with contents of cache line.

8 ShRep arrives at cache.

9 Update cache tag and data and return load data to CPU.

Write miss, to read shared line



1 Store request at head of CPU->Cache queue.

2 Store misses in cache.

3 Send ExReq message to directory.

4 ExReq message received at directory controller.

5 Access state and directory for line. Line’s state is R, with some set of sharers.

6 Send one InvReq message to each sharer.

7 InvReq arrives at cache.

8 Invalidate cache line. Send InvRep to directory.

9 InvRep received. Clear down sharer bit.

10 When no more sharers, send ExRep to cache.

11 ExRep arrives at cache

12 Update cache tag and data, then store data from CPU

Concurrency Management

- Protocol would be easy to design if only one transaction in flight across entire system
- But, want greater throughput and don't want to have to coordinate across entire system
- Great complexity in managing multiple outstanding concurrent transactions to cache lines
  - Can have multiple requests in flight to same cache line!