Faculty of Science

Memory Hierarchies & Shared Memory Systems

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

October 2024 PMPH Lecture Slides

Course Organization

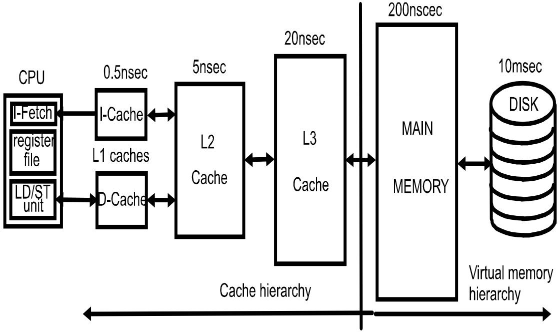
W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends Vector Machine	 ⟵	List HOM (Map-Reduce)	Intro & Simple Map Programming
2	In Order Processor	⟶ ⟵	VLIW Instr Scheduling	Scan & Reduce
3	Cache Coherence		Reasoning About Parallelism	Sparse Vect Matrix Mult
4	Interconnection Networks		Case Studies & Optimizations	Transpose & Matrix Matrix Mult
5	Memory Consistency		Optimising Locality	Sorting & Profiling & Mem Optimizations
6	OoO, Spec Processor		Thread-Level Speculation	Project Work

Three narative threads: the path to complex & good design:

- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

1. The Pyramid of Memory Levels
2. Cache Design
   - Cache Mappings
   - Replacement & Write (Back/Through) Policies
   - The Four Types of Cache Misses
3. Improving Performance: Lockup-Free Cache and Prefetching
4. Cache Coherence in Bus-Based Shared-Memory Multiprocessors
   - Simple Protocol for Write-Through Caches
   - Design Space: MSI, MESI, MOESI Protocols
   - Multiphase Cache Protocols
   - Cache Miss Classification (Updated)

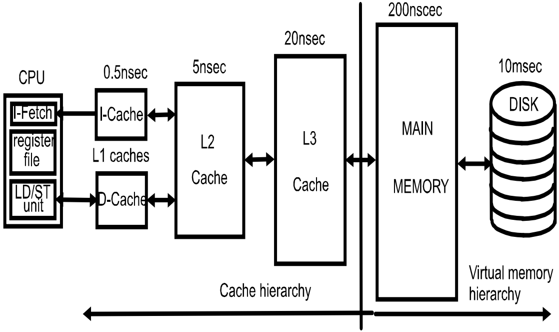
Typical Memory Hierarchy



Memory goes at electronic speed, Disk at mechanical speed.

Locality Principle:

Typical Memory Hierarchy

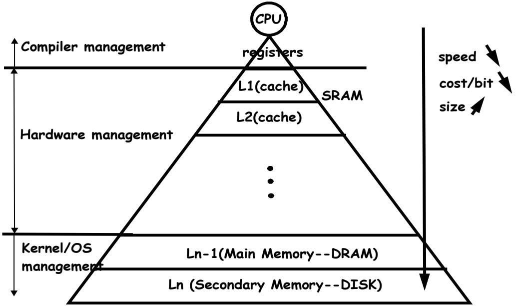


Memory goes at electronic speed, Disk at mechanical speed.

Locality Principle:

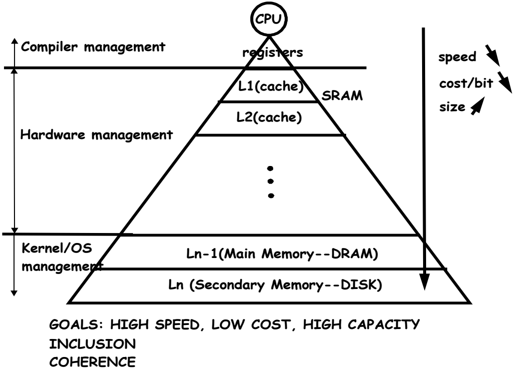
- small set of addresses accessed at a time, named working set, ⇒ low miss rate,
- when program transitions ⇒ abrupt change of working sets ⇒ high miss rate,
- Temporal Locality: a referenced item is likely to be accessed again soon,
- Spatial Locality: items close-by a referenced item likely to be accessed soon,
- Spatial ⇒ Temporal at block/page level.

Typical Memory Hierarchy



GOALS: HIGH SPEED, LOW COST, HIGH CAPACITY
INCLUSION
COHERENCE

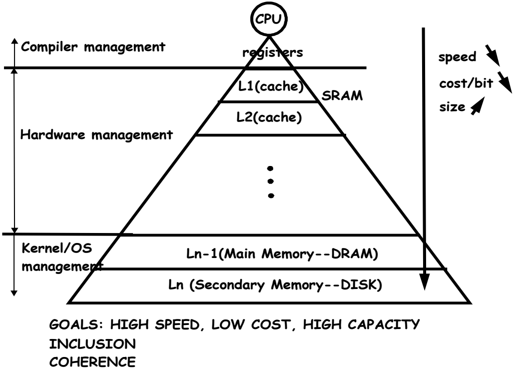
Typical Memory Hierarchy



- Illusion of a monolithic memory of lowest cost, largest capacity & fastest average access time.
- Larger caches are slower because speed dominated by wire delays (do not scale with technology).

- Coherence for single cores: instrs executed out of order & speculatively, but the result is as if instrs executed one at a time in program order & monolithic memory

Typical Memory Hierarchy



- Illusion of a monolithic memory of lowest cost, largest capacity & fastest average access time.
- Larger caches are slower because speed dominated by wire delays (do not scale with technology).

- Coherence for single cores: instrs executed out of order & speculatively, but the result is as if instrs executed one at a time in program order & monolithic memoryor "a load must return the value of the previous store to the same address".
- Inclusion: Cache level `j` includes `i` (`j > i`) ⇒ locations at level `i` are also cached & has same or more restrictive rights than level `j`. (Helps coherence.)

Cache Performance

- Average Memory Access Time (AMAT):
  `AMAT = hit time + miss rate × miss penalty`
- Miss Rate ≡ 1.0 - Hit Rate ≡ % of accesses not satisfied at highest level:
  `Miss Rate ≡ (# misses in L1) / (# processor references)`
- Misses Per Instruction (MPI):
  `MPI = (# misses in L1) / (# instructions)`
  Easier to use than Miss Rate: $\texttt{CPI} = \texttt{CPI}_0 + \texttt{MPI} \times \texttt{MissPenalty}$
- Miss Penalty: average delay per miss caused in the processor:
  If processor blocks on misses ⇒ miss latency (time to bring a block from mem)
  In an OoO processor cannot be measured directly ≠ miss latency
- Miss Rate and Penalty can be defined at every cache level. Normalized to:
  # of processor references or

  # of accesses from the lower level.

Cache Mapping

- Cache behavior mostly dictated by: cache size and
- **The mapping** of memory blocks to cache lines.
  (Each cache line hosts multiple mem blocks at different times.)
- *direct or set-associative or fully-associative mapping*.

Physical Address:

<table>
<tr><td colspan="2">Memory Block Address</td><td>Block Offset</td></tr>
<tr><td>TAG</td><td>Cache Index</td><td>Block Offset</td></tr>
</table>

Cache Acess Has Two Phases:

cache index	use index bits to fetch tags and data from the set,
tag check	check tag to detect hit/miss (and status bits).

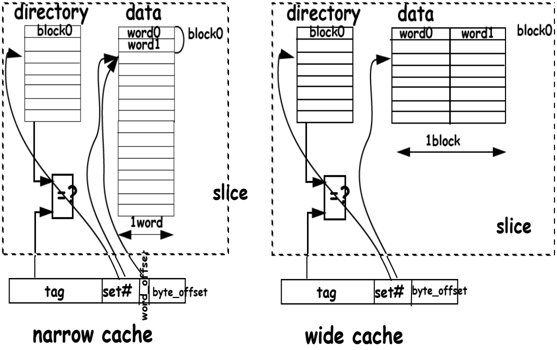
Cache:

- Data Memory, i.e., the cached copy of the memory block +
- Directory Memory, one entry per cache line containing
  `TAG` (`ID`) & status bits: valid, dirty, reference, cache coherence.

Direct-Mapped Cache

Cache Slicing: a memory block is always mapped in the same cache line, e.g., at index:

`(Block Address / Block Size) mod (# of cache lines).`



Two Phases:
Index + Tag Check

Data-Entry Size:

narrow: directory length < data length; takes several cycles to load a memory block;

wide: equal; on a miss, the data is reloaded in one cycle of data memory.

+ fast access time on a hit
- several blocks competing on the same line ⇒ high miss rate

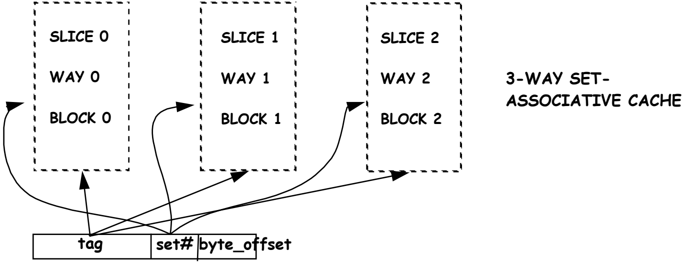
Set-Associative Cache

Cache is partitioned into a set of lines:

- access to each set is directly mapped, but
- a block may reside in any set!

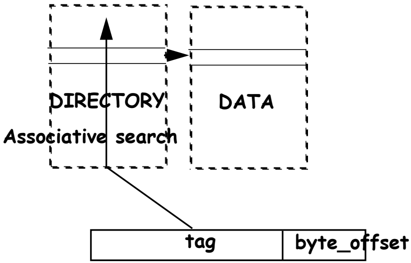
read requires one cycle: all 3 directory and data entries fetched in ||, then the tag is compared in || with the tag bits of each slice. A hit selects the corresponding word, if all miss ⇒ cache miss!

write requires at least two mem cycles (can be pipelined): one to check the hit or miss, and then one to write into data memory.



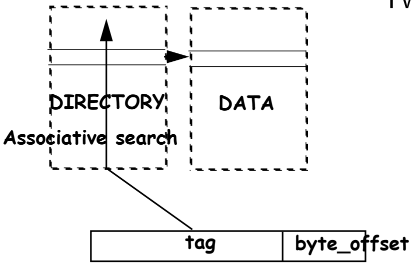
Full-Associative Cache

Very different structure than an all-way set-associative cache:
to find the block all directories must be checked in parallel!



Full-Associative Cache

Very different structure than an all-way set-associative cache: to find the block all directories must be checked in parallel!



Two Steps:

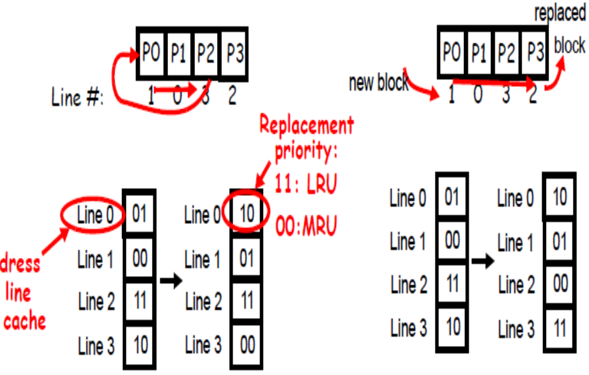
- || tag check ⇒ tag bus lines throughout the directory; comparator associated with each dir entry, then
- on match the row line is activated and data returned.
- load & store requires 2 cycles.

Content-Addressable Memory (CAM) slower & less dense than RAM. (signal propag, comparison, etc.)

Small Caches: intuition says they should be fully associative because potential for conflict in hot sets is damaging to performance (?).

Replacement Policies (Selects a Victim Block)

Random, Least Recently Used (LRU), FIFO, Pseudo-LRU: maintain replacement bits.



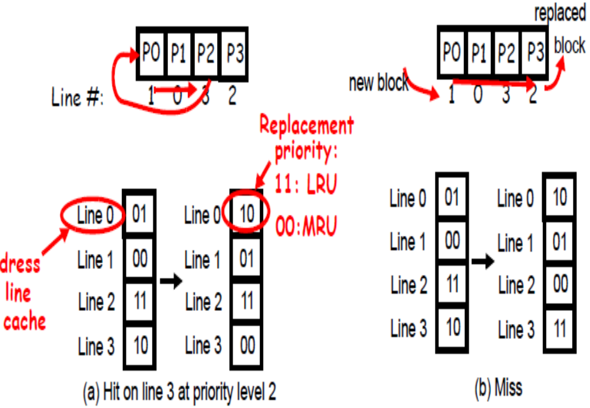
(a) Hit on line 3 at priority level 2

(b) Miss

⇐ (a) LRU Example
⇐ (b) FIFO Example
(history bits updated only on a miss by simple increment)

Replacement Policies (Selects a Victim Block)

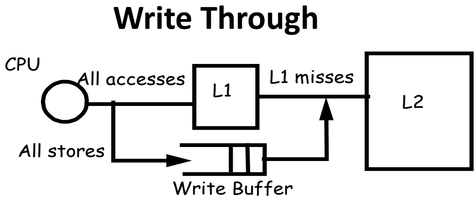
Random, Least Recently Used (LRU), FIFO, Pseudo-LRU:
maintain replacement bits.



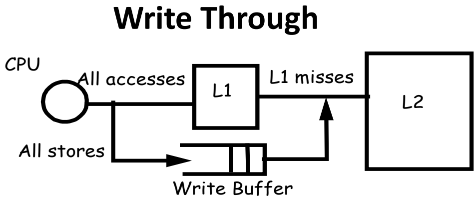
⇐ (a) LRU Example
⇐ (b) FIFO Example
(history bits updated only on a miss by simple increment)

Direct Mapper ⇒ No Need.
Set/Fully Associative ⇒ Per-Set/Cache Replacement.

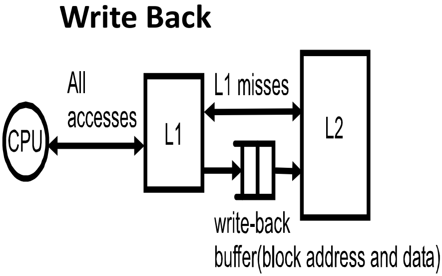
Write Policies



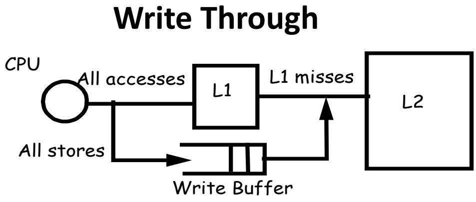
Write Policies



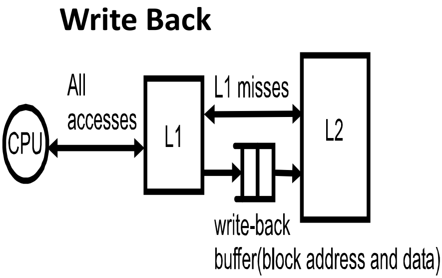
- **write to next level on all writes**
- use a write buffer to avoid stalls; loads must check the buffer first!
- Used for small 1st-level caches:
- simple, no inconsistency on levels
- but large store traffic



Write Policies



- **write to next level on all writes**
- use a write buffer to avoid stalls; loads must check the buffer first!
- Used for small 1st-level caches:
- simple, no inconsistency on levels
- but large store traffic



- **write to next level on replacement**
- uses a dirty bit (db) & write-back buffer block is loaded/modified ⇒ db reset/set block is evicted ⇒ if db set then written.
- write happens only on a miss
- IN BOTH CASES: a load checks the buffer first (consistency)!
- Write Miss: always allocate on write back; design choice in write through!

Classification of Cache Misses

The Four C's:

- **Cold** (Compulsory) misses: first reference of a block,
- **Capacity** misses: insufficient space for data/code,
- **Conflict** misses: two memory blocks map to the same cache line,
- **Coherence** misses, e.g., another thread has modified the needed value.

How to measure them:

Classification of Cache Misses

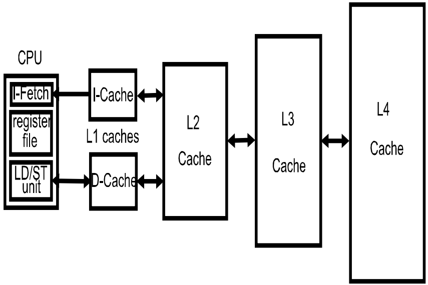
The Four C's:

- **Cold** (Compulsory) misses: first reference of a block,
- **Capacity** misses: insufficient space for data/code,
- **Conflict** misses: two memory blocks map to the same cache line,
- **Coherence** misses, e.g., another thread has modified the needed value.

How to measure them:

- **Cold:** simulate infinite cache size,
- **Capacity:** simulate fully-assoc cache and subtract cold misses
- **Conflict:** simulate cache and subtract cold and capacity misses.

Multi-Level Cache Hierarchies



1st and 2nd levels on chip; 3rd and 4th mostly off chip

We will assume Cache Inclusion. A block

- misses in $L_i \Rightarrow$ must be brought in all $L_j, j > i$.
- is replaced in $L_j \Rightarrow$ must be removed in all $L_i, j > i$.
- replication but good for coherence.

Cache Exclusion. A block:

- is in $L_i \Rightarrow$ then it is not in any other level.
- misses in $L_i \Rightarrow$ all copies are removed from all levels $> i$.
- is replaced in $L_j \Rightarrow$ allocated in $j+1$.
- size is the sum of all caches, but horrible for coherence.

Cache Parameters

Large Caches: slower (wire delays), more complex, less capacity misses.

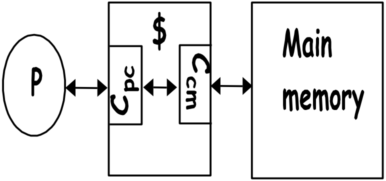
Larger Block Size:

- exploits spatial locality, but
- if too big $\Rightarrow$ increased number of capacity misses
- big blocks increase miss penalty.

Higher Set Associativity (SA):

- reduces the number of conflict misses;
- increases the hit latency;
- 8-16 ways SA as good as fully associative;
- A 2-way SA cache of size N has similar miss rate with a direct mapped of size $2 \times N$.

Lockup-Free Caches



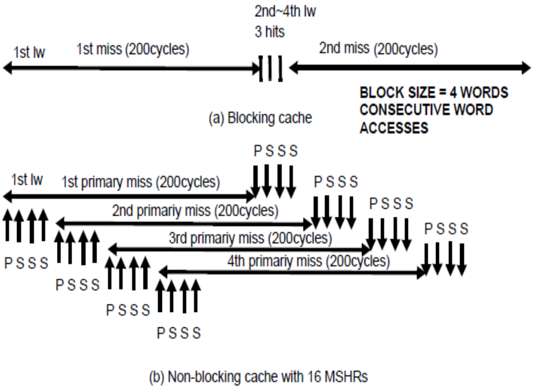
Cache is a Two-Ported Device: Memory & Processor.

C_{pc} : cache to processor interf

C_{cm} : cache to memory interface

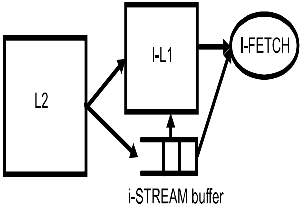
- Needed to support Prefetching & Dynamically-Scheduled OoO Single Proces & Core MultiThreading & Multi Cores
- A Lockup-Free Cache does not block on a miss, but keeps accepting proc requests,
- hence, it allows concurrent processing of multiple hits/misses.
- Cache has to bookkeep all pending misses:
  - Miss Status Handling Register (`MSHR`) contains address of the pending miss + destination block in cache + destination register.
  - `MSHR` used to complete a miss and to avoid sending multiple miss requests per block. # of `MSHR`s limits the # of pending misses (at a time).
- Data dependencies eventually block the processor.

Lockup-Free Caches (Continuation)



- Primary Miss (P) is the first miss to a block
- Secondary Miss (S) next accesses to same block (due to pending P)
  - Many more misses than Blocking Cache, which has only Ps.
  - Needs `MSHRs` for both P and S misses
  - Misses are overlapped with computation and other misses.

Hardware Prefetching

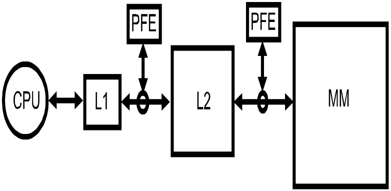


Sequential Prefetching Of Instrs:

I-Fetch Miss ⇒ fetch 2 blocks instead of 1.

2nd block stored in i-STREAM buffer:
(1) If I-STREAM hits ⇒ block moved to L1
(2) Not accessed ⇒ I-STREAM blocks overlaid.
(3) Prefetch Buffer avoids cache pollution.

(4) Applicable to data but less effective.



Hardware Prefetch Engines (PFE):

(1) detect strides in stream of missing addresses by observing the bus then start fetching ahead.
(2) naturally triggered by speculative exec:
(3) prefetch is harmless, i.e., exception ⇒ prefetch dropped.

(4) but might polute caches.

Software Prefetching

- Prefetch instrs: non-blocking & non-binding (load in-cache only)
- E.g., prefetch instructions may be inserted in the loop's body to prefetch data needed by future iterations:

HL Code / **MIPS Code**

```
for(i=1000;i>0;i--)
  A[i]=A[i]+s
```

```
Loop: L.D    F2, 0(R1)
      PREF     -24(R1)
      ADD.D  F4, F2, F0
      S.D    F4, 0(R1)
      SUBI   R1, R1, #8
      SUBI   R2, R2, #1
      BNEZ   R2, Loop
```

`PREF -24(R1)` prefetches the elements of `A` 3 iterations ahead.

- Works for both load and stores, but
- data must be prefetched at perfect time:
  not too early (cache polution), not too late (not in cache),
- Instructional overhead & requires non-blocking cache,
- Done for arrays, but also for pointer accesses.

Faster Hit Times

Princeton vs Harvard Cache:

- Princeton: unified instr/data cache $\Rightarrow$ can use whole cache
- Harvard: split instr/data cache $\Rightarrow$ optimized for access type.
- Pipelined Machine: FstLC Harvard & SndLC Princeton.

Pipeline Cache Accesses:

- Especially useful for stores:
- Pipeline Tag Check and Data Store (2 mem cycles)
- Separate Read/Write Ports to cache, optimized for each
- Also useful for I-Caches and Load in D-Caches
- $\uparrow$ pipeline length, but must split cache accesses into stages!

What Should First Level Cache (FLC) Be?

What Should First Level Cache (FLC) Be?

Keep the cache simple and fast:

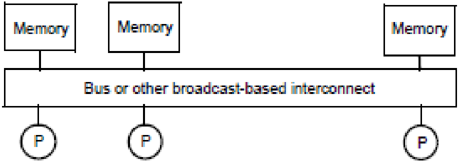
- Favors direct-mapped cache:
  - less multiplexing
  - overlap of tag and use of data.

- Interestingly, the size of FLC tends to decrease and associativity goes up as FLCs try to keep up with CPU.

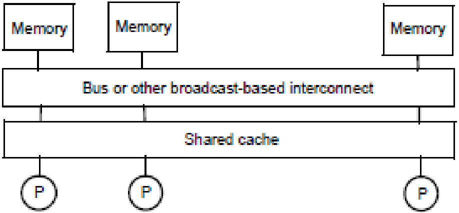
Processor	L1 Data Cache
Alpha 21164	8KB, direct mapped
Alpha 21364	64KB, 2-way
MPC 750	32KB, 8-way, PLRU
PA 8500	1MB, 4-way, PLRU
Classic Pentium	16KB, 4-way, LRU
Pentium-II	16KB, 4-way, PLRU
Pentium-III	16KB, 4-way, PLRU
Pentium-IV	8KB, 4-way, PLRU
MIPS R10K/12K	32KB, 2-way, LRU
UltraSparc-IIi	16KB, direct mapped
UltraSparc-III	64KB, 4-way, random

Organization of Bus-Based Shared-Memory SMPs

Design space of cache coherence for small scale SMPs assuming a broadcast-based interconnect, such as a bus.



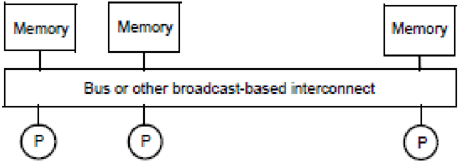
(a) Dance-hall multiprocessor architecture or SMP



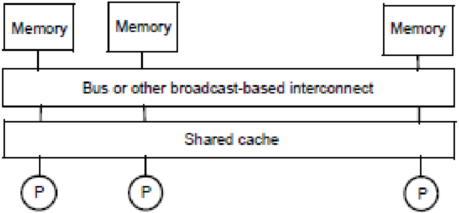
(b) SMP with shared level 1 cache

Organization of Bus-Based Shared-Memory SMPs

Design space of cache coherence for small scale SMPs assuming a broadcast-based interconnect, such as a bus.



(a) Dance-hall multiprocessor architecture or SMP



(b) SMP with shared level 1 cache

(a) Dance-Hall: implicit coherency but not realistic!

- Cache hierarchy vital for SMPs:
- hides memory & interconnect latency,
- saves mem & interconnect bandwidth.

(b) Shared Cache between processor & interconnect:

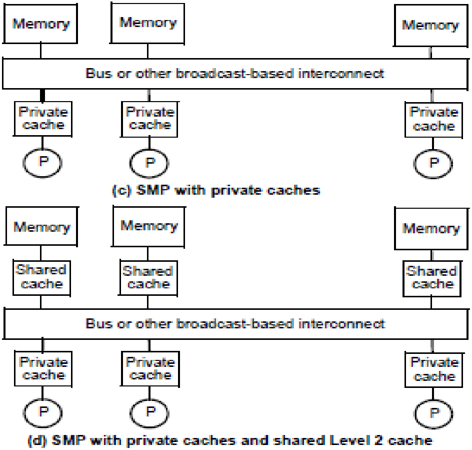
\+ constructive sharing of cache resources.

− interconnect latency added to the critical mem access path ⇒

effective when very few processors.

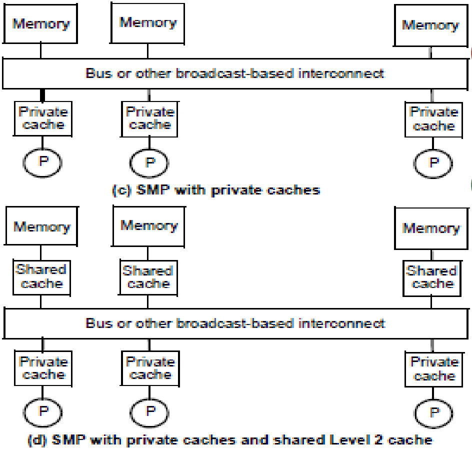
Organization of Bus-Based Shared-Memory SMPs

Design space of cache coherence for small scale SMPs assuming a broadcast-based interconnect, such as a bus.



Organization of Bus-Based Shared-Memory SMPs

Design space of cache coherence for small scale SMPs assuming a broadcast-based interconnect, such as a bus.



(c) Private Cache on Proc Side:

(d) Hybrid Private-Shared Cache:

- Private Cache between processor & interconnect:
- 2nd level cache (SLC) reduces memory access latency gap between 1st level cache (FLC) and memory
- used in most commercially available processors.

Informal Definition of Cache Coherence

Informal Definition of Cache Coherence

Definition (Sequential Cache Coherence)

A load must return the value of the latest store in process order to the same address. (Simple, but check the write buffers.)

Definition (Cache Coherence in Multiprocessors)

A cache system is cache coherent *iff* all processors, at any time, have a consistent view of the last globally written value to each location.

Coherence Problem: pervasive & performance critical

- sharing of data, implicit communication,
- thread migration, software not informed $\Rightarrow$ hardware must solve the problem.

Locking, Barrier, Point-to-Point Synchronization

Barrier and Point-to-Point Synchronization

T_1	T_2	T_1	T_2
`...`	`...`		`A := 1;`
`BAR := BAR + 1`	`BAR := BAR + 1`		`FLAG := 1;`
`while( BAR < 2 ) ;`	`while( BAR < 2 ) ;`	`while( FLAG == 0 ) ;`	
`...`	`...`	`print A`	

Point-to-Point: no need for critical section; producer-consumer sync.

Barrier: all threads have to reach it before executing beyond it.

- need critical section to increment BAR + read/reset BAR fine.
- but even assuming the writes to bar do not interleave, i.e., atomic, with the current cache design:

Locking, Barrier, Point-to-Point Synchronization

Barrier and Point-to-Point Synchronization

T_1	T_2	T_1	T_2

`...`
`BAR := BAR + 1`
`while( BAR < 2 ) ;`
`...`
	`...`
	`BAR := BAR + 1`
	`while( BAR < 2 ) ;`
	`...`
		`while( FLAG == 0 ) ;`
		`print A`
			`A    := 1;`
			`FLAG := 1;`

Point-to-Point: no need for critical section; producer-consumer sync.

Barrier: all threads have to reach it before executing beyond it.

- need critical section to increment BAR + read/reset BAR fine.
- but even assuming the writes to bar do not interleave, i.e., atomic, with the current cache design:

write back: P1 and P2 write M, then barrier, then both read M ⇒ if cache line not evicted, both read their private-cache value.

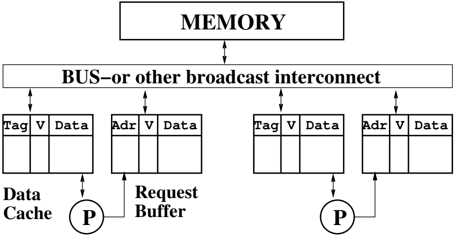
write through: P1 writes M (mem updated), then P2 writes M, barrier ⇒ P1 will read an inconsistent value next from its private cache.

Need Coherence!

Simple Protocol For Write-Through Caches

Simplifying Assumptions:

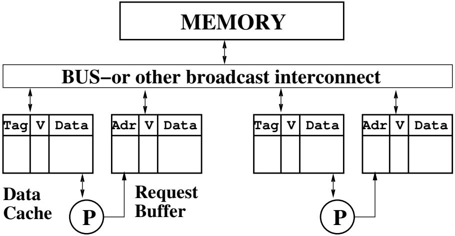
- blocking, write-through, write-allocate, single-level private cache as in (c)
- when granted access, cache controller owns the bus until transaction completes.



Simple Protocol For Write-Through Caches

Simplifying Assumptions:

- blocking, write-through, write-allocate, single-level private cache as in (c)
- when granted access, cache controller owns the bus until transaction completes.



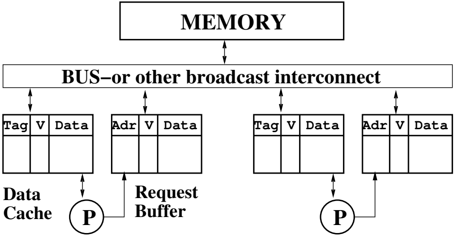
Local cache updated last: "last globally written value" cannot be released until all other procs can see the new value.

Rd Miss inserted in (request) buffer, V(alid) bit set. When bus acquired ⇒ BusRd request placed on bus & returns copy of mem block.

Simple Protocol For Write-Through Caches

Simplifying Assumptions:

- blocking, write-through, write-allocate, single-level private cache as in (c)
- when granted access, cache controller owns the bus until transaction completes.



Local cache updated last: "last globally written value" cannot be released until all other procs can see the new value.

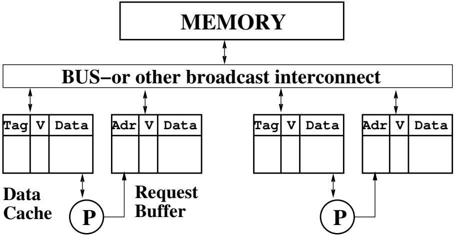
Rd Miss inserted in (request) buffer, V(alid) bit set. When bus acquired $\Rightarrow$ BusRd request placed on bus & returns copy of mem block.

Wr Hit: value and address inserted in buffer.

Simple Protocol For Write-Through Caches

Simplifying Assumptions:

- blocking, write-through, write-allocate, single-level private cache as in (c)
- when granted access, cache controller owns the bus until transaction completes.



Local cache updated last: "last globally written value" cannot be released until all other procs can see the new value.

Rd Miss inserted in (request) buffer, V(alid) bit set. When bus acquired ⇒ `BusRd` request placed on bus & returns copy of mem block.

Wr Hit: value and address inserted in buffer. When acquired, `BusWrite` request on bus ⇒ updates memory AND invalidates all remote copies (clears V). Local cache updated just before releasing bus.

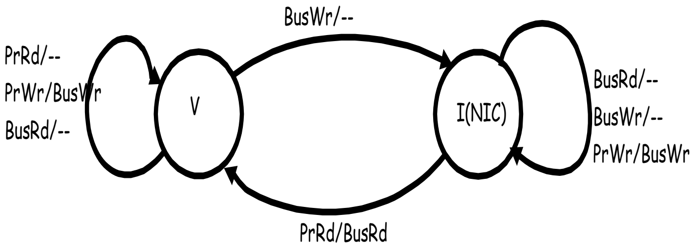
Wr Miss like Wr Hit, but `BusRdX` on bus (also brings block from mem). For no-write-allocate: like WR Hit, but local cache not updated.

Specify Behavior via Finite State Machine (FSM)

Each memory block in a private cache is represented by a FST:

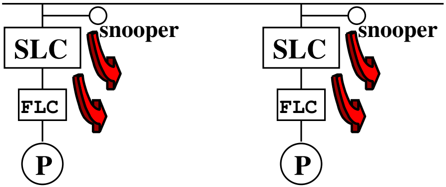
- Imagine P identical FSM working together (one per cache line),
- Actually, FSM shows the cache behavior w.r.t. a memory block.
- 2 states: Valid or Invalid (**not in cache**), requires 1 bit (V)

For example a write (hit) in valid state remains in valid, but triggers a `BusWrite` which may cause other caches to transition to invalid.
Figure below assumes a no-write-allocate policy. Why?



Copy Invalidation By Bus Snooping

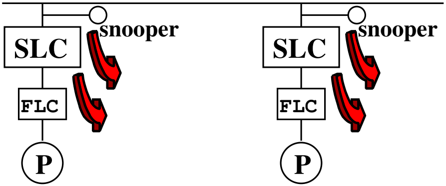
- Bus interface can monitor (snoop) traffic &
- if tag matches ⇒ invalidate (local) cache entry (clear V).



Copy Invalidation By Bus Snooping

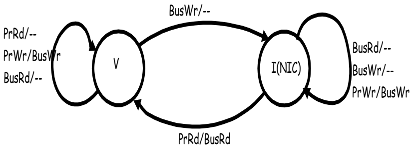
- Bus interface can monitor (snoop) traffic &
- if tag matches ⇒ invalidate (local) cache entry (clear V).

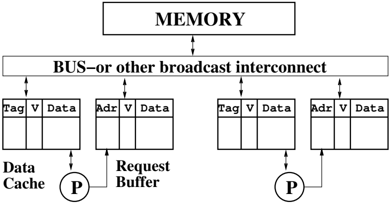
Dual Directory (DD) is a copy of cache directory, kept consistent on updates (rare). DD filters out bus requests to avoid conflicts with CPU.



Inclusion ⇒ SLC contains bit indicating whether block is in FLC, and is used to filter out transactions from FLC (since SLC far less busy than FLC)

Example of Subtle Issue

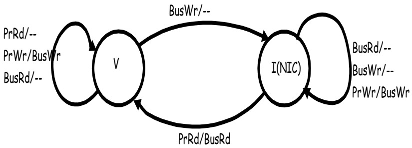


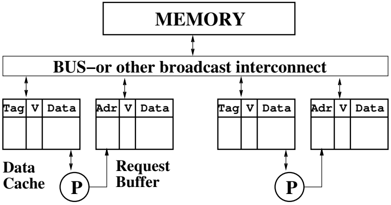


FST specifies the high-level behavior of cache as a whole.
The use of request buffer is not safe, and will be fixed later.

- P1 and P2 issue write hits to block A.
- Assume P1 acquire bus while P2 waits in buffer. What happens?

Example of Subtle Issue





FST specifies the high-level behavior of cache as a whole.
The use of request buffer is not safe, and will be fixed later.

- P1 and P2 issue write hits to block A.
- Assume P1 acquire bus while P2 waits in buffer. What happens?
- P1 issues a `BusWrite`, which invalidates the P2's cache ⇒
- When P2 acquires bus, it has to check V bit in cache, and send a `BusRdX`, rather than `BusWrite` request.

MSI Protocol for Write Back Caches

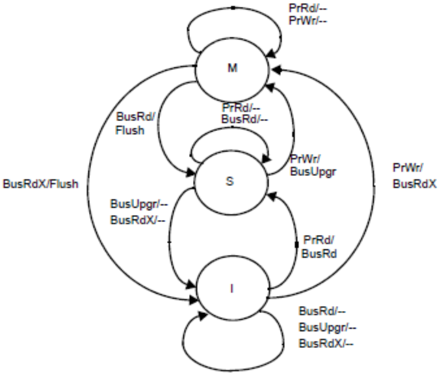
Simple Protocol suffers performance bottlenecks:

- All writes launch bus transactions.

MSI Protocol for Write Back Caches

Simple Protocol suffers performance bottlenecks:

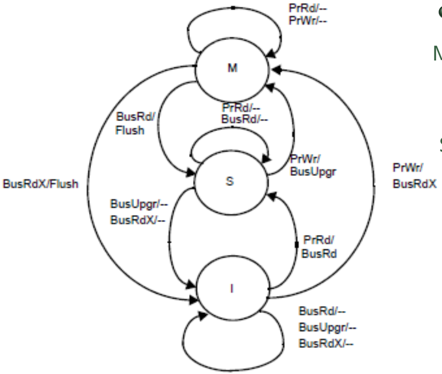
- All writes launch bus transactions.
- Key Insight: most blocks accessed exclusively by one processor ⇒ accesses to non-shared block cannot interfere with other caches!



MSI Protocol for Write Back Caches

Simple Protocol suffers performance bottlenecks:

- All writes launch bus transactions.
- Key Insight: most blocks accessed exclusively by one processor ⇒ accesses to non-shared block cannot interfere with other caches!



- State V split into M(odified) & S(hared):

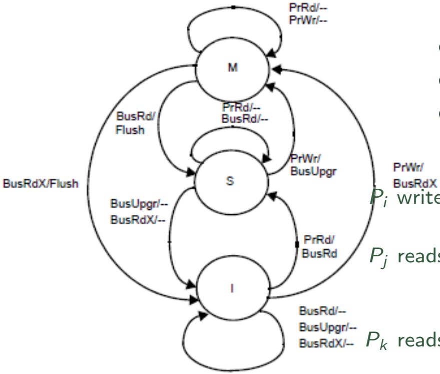
M local copy is the only up-to-date one,
read & writes performed locally!
(Extra state bit M for write-back cache.)

S several remote copies available & all copies in S and memory are up-to-date!
Reads operate locally, but a write must invalidate all remote copies (via `BusUpgr`).

I local copy invalid or not in cache.
Who provides the value on a read miss?
If exists, **necessarily** by a remote copy in M;
Operation named `Flush`: forward block copy to requester & also update memory.
Otherwise, either by a copy in S or mem.

MSI Protocol: Examples

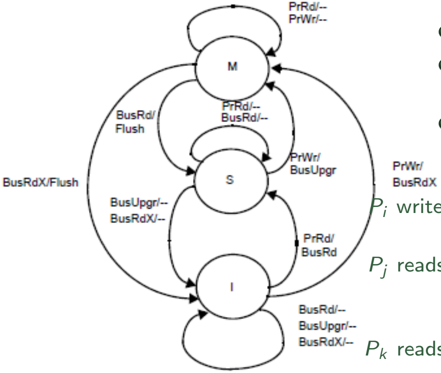
- BusRd requests a copy with no intent to modify.
- BusRdX requests a copy with intent to modify (and invalidates remote copies).
- BusUpgr invalidate remote copies.
- Flush forwards copy to requester & update memory.



1. Assume block A not in any cache:
   - P reads A:
   - P writes A:
   - Next reads & writes of P:
2. Assume only P_i and P_j have S copies.
   - P_i writes
   - P_j reads.
   - P_k reads.

MSI Protocol: Examples

- `BusRd(X)` requests copy with (no) intent to modify.
- `BusUpgr` invalidate remote copies.
- `Flush` forward copy to requester & update memory.



1 Assume block A not in any cache:

- P reads A: I $\rightarrow$ S
- P writes A: S $\rightarrow$ M and launches `BusUpgrd` to invalidate remote copies,
- Next reads & writes of P: execute locally.

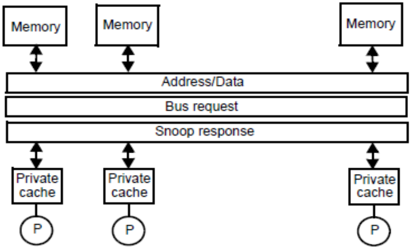
2 Assume only P_i and P_j have S copies:

P_i writes P_i: S $\rightarrow$ M and launches `BusUpgr`, on which P_j: S $\rightarrow$ I

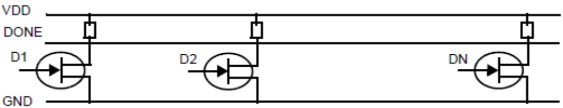
P_j reads. P_j: I $\rightarrow$ S and launches `BusRd`, on which P_i: M $\rightarrow$ S and flushes its (only up-to-date) copy to P_j and memory.

P_k reads. P_k: I $\rightarrow$ S and launches `BusRd`, and the value is brought from memory. All copies are in state S.

MSI: Hardware Structures



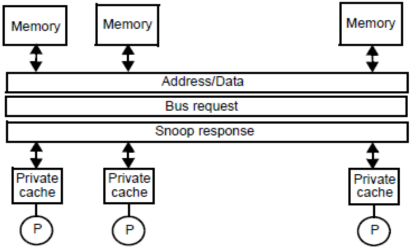
(a) Structure of a snooping bus



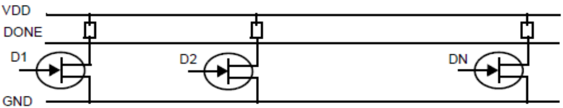
(b) Wired-NOR bus used in handshakes

- Transaction starts by supplying address/data and a request.
- and triggers a snoop action, e.g., invalidate tag, and reply, e.g., when have all completed it?
- Synchronous reply:
- Asynchronous by handshake (b): `NOR(D1,..,Dn)`$\Rightarrow$ `DONE`, i.e.,
- Fetch from Remote or Memory?

MSI: Hardware Structures



(a) Structure of a snooping bus



(b) Wired-NOR bus used in handshakes

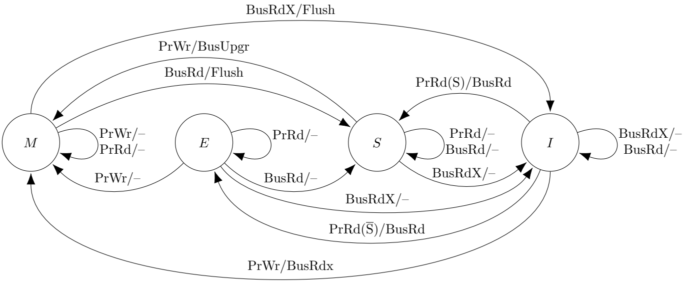
- Transaction starts by supplying address/data and a request.
- and triggers a snoop action, e.g., invalidate tag, and reply, e.g., when have all completed it?
- Synchronous reply: establishes an upper-bound latency that factors in conflicts ⇒ use DualDirectory.
- Asynchronous by handshake (b): `DONE = NOR(D1,..,Dn)`, i.e., `If` any `Di` is 1 `Then DONE` driven to ground 0 `Else` to supply volt 1.
- Fetch from Remote or Memory? $\overline{\texttt{REMOTE}}$`=NOR(M1,..,Mn)`, where `Mi` is 1 when block in cache `i` is in state M. Similar handshake.

- Initiate memory access in PARALLEL with snoop action, but memory responds only after $\overline{\texttt{REMOTE}}$ is known as 1.
- To reduce miss latency when it triggers replacement of M block ⇒ move block to victim buffer, which also supports snooping.

MESI Protocol for Write Back Caches

MSI: read miss followed by write require TWO bus accesses.

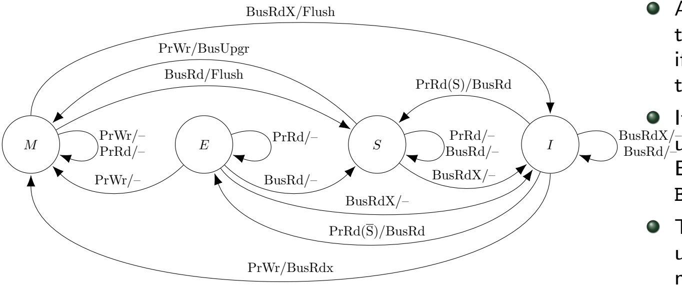
E(xclusive) State entered on a read miss, when block is only in mem. Uses a S(hared) bus line to detect whether the copy will be unique.



MESI Protocol for Write Back Caches

MSI: read miss followed by write require TWO bus accesses.

E(xclusive) State entered on a read miss, when block is only in mem.
Uses a S(hared) bus line to detect whether the copy will be unique.

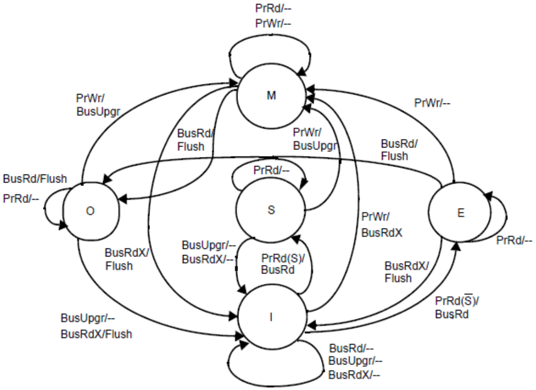


- A read miss transitions from I to S if shared line is 1, and to E otherwise.
- If a block in E is updated (`PrWr`) then E→M without `BusUpgr`.
- Transition S → M uses `BusUpgr`, because memory block is already in the cache.

MOESI: A General Class of Protocols

MOESI adds a notion of ownership:

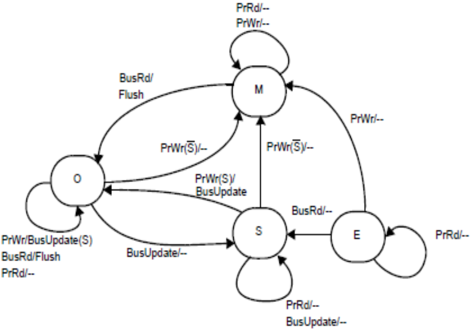
- memory is eventually updated by owner (not at every write)
- allows cache-to-cache transfers between owner and requester (even when access was not exclusive).



Ownership transfered to another cache or memory when block is invalidated or replaced!

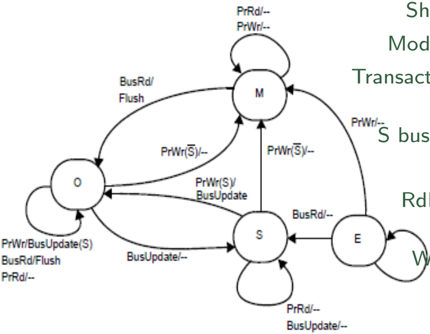
Write Back: M(O)ESI Update Protocol

Dragon Multiprocessor (Xerox PARC 1980):
Same states as MOESI, but Invalid omitted to simplify.



Write Back: M(O)ESI Update Protocol

Dragon Multiprocessor (Xerox PARC 1980):
Same states as MOESI, but Invalid omitted to simplify.



Shared	multiple copies, memory is clean.
Modified	one copy, memory is stale, etc.
Transactions	BusRd requests a copy, BusUpdate updates remote copies.
S bus line	indicates whether remote copies exist.
RdMiss	If no other cached copies, block loaded from mem in E, Otherwise in S.
WrHit	If no other copies E→M without using bus. Else (Shared line 1) all copies are updated via `BusUpdate`, and the update is propagated from owner.

Under which program behavior is invalidation or update protocol best?

Comparison Invalidate vs Update Protocol

Write-Run of an access sequence to the same block is the set of consecutive writes of the same processor before encountering a read/write of another processor.

Example: Write Run Length of $R_1, W_1, R_1, W_1, W_2, R_2$ is 2.

Bandwidth (B) for a Write-Run of length N:

INVALIDATE `B(UPGRADE) + B(READ MISS)`

UPDATE $N \times$ `B(UPDATE)`

Assuming `B(UPGRADE)` $\equiv$ `B(UPDATE)` then

Update outperforms Invalidate (i.e., uses less bandwidth) when

Comparison Invalidate vs Update Protocol

Write-Run of an access sequence to the same block is the set of consecutive writes of the same processor before encountering a read/write of another processor.

Example: Write Run Length of $R_1, W_1, R_1, W_1,\ W_2, R_2$ is 2.

Bandwidth (B) for a Write-Run of length N:

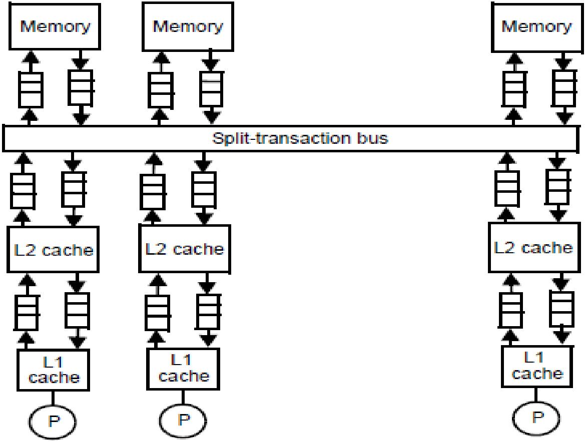
INVALIDATE `B(UPGRADE) + B(READ MISS)`

UPDATE `N × B(UPDATE)`

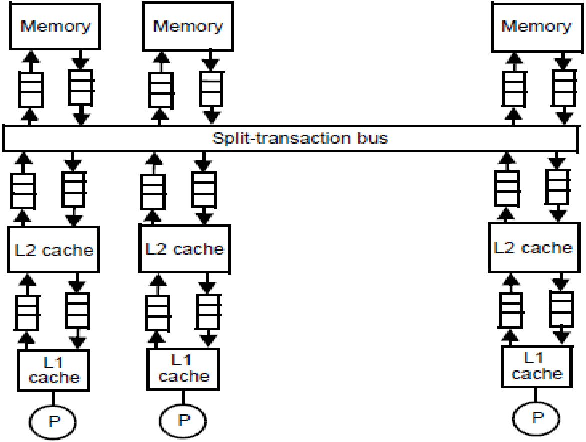
Assuming `B(UPGRADE)` $\equiv$ `B(UPDATE)` then
Update outperforms Invalidate (i.e., uses less bandwidth) when
`N < 1 + B(READ MISS)/B(UPDATE)`

This becomes: $N < 1 + s$, where s is the # of words in a cache line, (because the update protocol updates only one word.)

Multi-Phase Snoopy Cache Protocols



Multi-Phase Snoopy Cache Protocols



So far we have assumed:

- single level of private cache,
- and atomic pipelined buses.

A More Realistic Model:

- multi-level private cache hierarchy
- a split transaction (pipelined) bus *request & response phases*

Different caches/memory cannot consume requests at the same rate. FIFO requests buffers smooth out differences, and have a profound impact on the protocol design!

Atomic Transaction Disadvantages

Example: bus clocked at 100MHz can transfer 3 parallel segments in one cycle: (1) a request, (2) an address and (3) 256-bits of data. Assume no caches and that memory is banked and can supply a 32-byte cache block in 200 ns. What fraction of the time will the atomic bus be idle?

Atomic Transaction Disadvantages

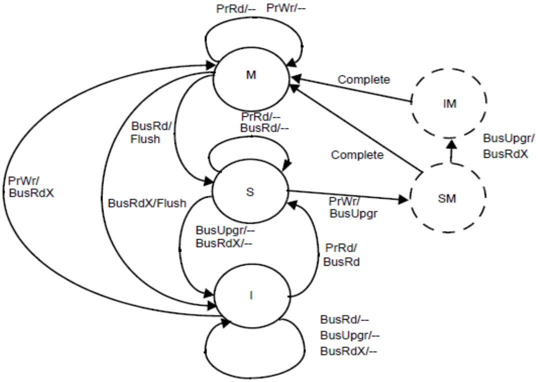
Example: bus clocked at 100MHz can transfer 3 parallel segments in one cycle: (1) a request, (2) an address and (3) 256-bits of data. Assume no caches and that memory is banked and can supply a 32-byte cache block in 200 ns. What fraction of the time will the atomic bus be idle?

1 clock cycle: `1sec / freq = 10ns`.
Bus is idle: $200/220 = 91\%$ of time.

Transient Non-Atomic Cache States for MSI

Need to improve the FSMs to cope with non-atomic transactions!

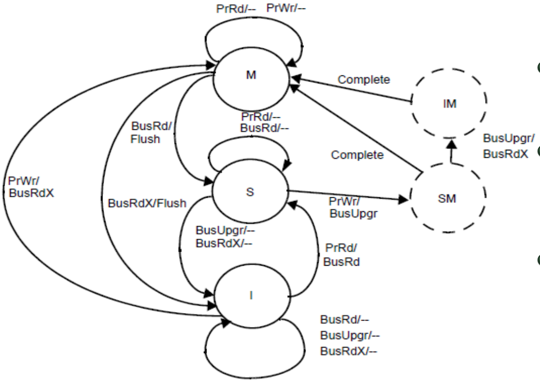
For example, we need to add transient states SM and IM.



Transient Non-Atomic Cache States for MSI

Need to improve the FSMs to cope with non-atomic transactions!

For example, we need to add transient states SM and IM.



- On a P1 write hit a block in S transitions to SM and enqueues `BusUpgr`.
- If a `BusUpgr` received from P2 before P1 transact completed, Then P1 goes to IM and `BusRdX` replaces `BusUpgr` in buffer.
- Eventually, when transition completed, i.e., request sent and reply received, P1 goes to M.

Split-Transaction Bus

Pipelines a sequence of phases in a bus transaction, e.g., arbitration, transfer, response.

Dividing a transaction into subtransactions $\Rightarrow$ Tradeoff between

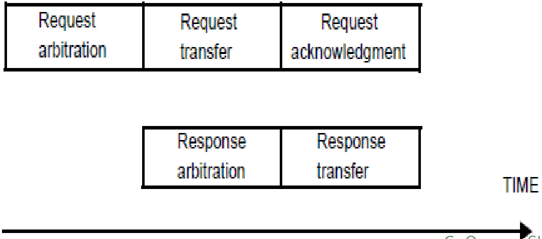
Split-Transaction Bus

Pipelines a sequence of phases in a bus transaction, e.g., arbitration, transfer, response.

Dividing a transaction into subtransactions ⇒ Tradeoff between additional latency (repeated bus arbitration) and better bandwidth.

Pipeline stages must be balanced to maximize throughput.

For example, if both request and response transfer use the address bus, they can only be pipelined:



Multi-Level Cache Issues

Adding another level of private cache offers benefits:

- shorter miss penalty to next level,
- filters out snoop actions to first level ⇒
- less proc-bus conflicts on L1 cache & less snoop latency!
- especially if cache inclusion is maintained (e.g., it can be forced by evicting an L1 block when is evicted from L2.)

Write Policy is important to reduce snoop overhead:

- If L1 is write-back Then L2's copy is inconsistent and dirty miss requests must be serviced by L1.

Multi-Level Cache Issues

Adding another level of private cache offers benefits:

- shorter miss penalty to next level,
- filters out snoop actions to first level ⇒
- less proc-bus conflicts on L1 cache & less snoop latency!
- especially if cache inclusion is maintained (e.g., it can be forced by evicting an L1 block when is evicted from L2.)

Write Policy is important to reduce snoop overhead:

- If L1 is write-back Then L2's copy is inconsistent and dirty miss requests must be serviced by L1.
- If L1 is write-through and inclusion is maintained ⇒ L2 is consistent and can service all miss requests from other processors,
- which can significantly improve performance.

True vs False Sharing

True Sharing Communicates Values (Essential):

- Two processors access the same word. Remember:
- Update protocol better for Fine-Grained Sharing (short Write Runs)
- Invalidate better for Coarse-Grain Sharing: `N` $>$ `1+b`, where `b` is # of words in a cache line, `N` is the write-run length.

False Sharing Does Not Communicate Values (pure overhead)

- P1 and P2 access two different words in the same block.
- Write Invalidate causes false sharing misses, e.g., P1 write `W1` then P2 reads `W2`, where `W1` and `W2` are distinct words in the same block.
- Write Update causes false sharing updates to dead copies.

Essential vs Non-Essential Misses

Assume A, B, C belong to same block B1, and D to another block

Time	Proc1	Proc2	Proc3	Miss Type
1	R_A			Cold
2		R_B		Cold
3			R_C	Cold
4			R_D(evict B1)	Cold
5	W_A			
6		R_A		True Sharing
7	W_B			
8		R_A		False Sharing
9			R_C	Replacement

Cold, True Sharing (coherence), and replacement (conflict or capacity) misses are Essential; False Sharing misses are Non Essential.

Same reasoning can be applied to memory traffic.

Classification of Misses