CSE 502 Graduate Computer Architecture
Lec 19 – Directory-Based Shared-Memory Multiprocessors & MP Synchronization

Larry Wittie
Computer Science, StonyBrook University

http://www.cs.sunysb.edu/~cse502 and ~lw

Slides adapted from David Patterson, UC-Berkeley cs252-s06
Review & Assignment

• Caches contain all information on state of cached memory blocks
• Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
• Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
• Reading Assignment: Finish Chap. 4 MPs and start Chap 5 Memory Hierarchies.
Outline

• Review
• Directory-based protocols and examples
• Synchronization
• Relaxed Consistency Models
• Fallacies and Pitfalls
• Cautionary Tale
• Conclusion
A Cache Coherent System Must:

• Provide set of states, state transition diagram, and actions

• Manage coherence protocol
  – (0) Determine when to invoke coherence protocol
  – (a) Find info about state of block in other caches to determine action
    » whether need to communicate with other cached copies
  – (b) Locate the other copies
  – (c) Communicate with those copies (invalidate/update)

• (0) is done the same way on all systems
  – state of the line is maintained in the cache
  – protocol is invoked if an “access fault” occurs on the line

• Different approaches distinguished by (a) to (c)
Bus-based Coherence

• All of (a), (b), (c) done through broadcast on bus
  – faulting processor sends out a “search”
  – others respond to the search probe and take necessary action

• Could do it in scalable network too
  – broadcast to all processors, and let them respond

• Conceptually simple, but broadcast does not scale with $p$
  – on bus, bus bandwidth does not scale
  – on scalable network, every fault leads to at least $p$ network transactions

• Scalable coherence:
  – can have same cache states and state transition diagram
  – different mechanisms to manage protocol
Scalable Approach: Directories

• Every memory block has associated directory information
  – keeps track of copies of cached blocks 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
Basic Operation of Directory

- **k processors (or k snoopy nodes).**
- **With each cache-block in memory:**
  - k presence-bits, 1 dirty-bit
- **With each cache-block in cache:**
  - 1 valid bit, and 1 dirty (owner) bit

• Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON }
  - If dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i}

• Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON }
  - If dirty-bit ON then { recall line from dirty proc (cache state to invalid); update memory; supply recalled data to i; send invalidations to all caches that have the block; leave dirty-bit ON; turn p[i] ON }
Directory Protocol

• Similar to Snoopy Protocol: Three states
  – **Shared**: \( \geq 1 \) processors have data, memory up-to-date
  – **Uncached** only in memory; no processor has it; not valid in any cache
  – **Exclusive**: 1 processor (owner) has data in its cache; or **Dirty** memory copy is out-of-date

• In addition to cache state, directory must track which (multi-)processor( node)s have data when in the shared state (usually bit vector, 1 if processor has copy)

• Keep it simple(r):
  – Writes to non-exclusive data (exclusive changes can be private) \( \Rightarrow \) write miss
  – Processor blocks until access completes
  – Assume messages received and acted upon in order sent
Directory Protocol

• No bus and do not want to broadcast across network:
  – shared interconnect no longer is a 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

• In example messages on next slide:
P = processor number, A = address
## Directory Protocol Messages (Fig 4.22)

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Processor P reads data at address A;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>make P a read sharer and request send data block to P</em></td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Processor P has a write miss at address A;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>make P the exclusive owner and request old data for P</em></td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Invalidate a shared copy at address A</em></td>
</tr>
<tr>
<td>Fetch</td>
<td>Home directory</td>
<td>Remote owner cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Fetch the block at address A and send it to its home directory;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>change the state of A in the remote cache to shared</em></td>
</tr>
<tr>
<td>Fetch/Invalidate</td>
<td>Home directory</td>
<td>Remote owner cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Fetch the block at address A and send it to its home directory;</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>invalidate the block in the remote ex-owner cache</em></td>
</tr>
<tr>
<td>Data value reply</td>
<td>Home directory</td>
<td>Local cache</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Return a data value from the home memory (read miss response)</em></td>
</tr>
<tr>
<td>Data write back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><em>Write back a data value for address A (invalidate or sharer response)</em></td>
</tr>
</tbody>
</table>
Implementing a Directory

• We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix E)

• Optimizations to lessen network traffic, shorten latencies, or work by (bottleneck) directory PU:
  – For read-miss or write-miss of Exclusive cacheblock: send data directly to requestor from owner instead of owner sending to memory and then memory sending to requestor
  – For read-miss or write-miss of Exclusive cacheblock: let directory send cacheblock owner id to requesting remote node and let requestor send message to owner to lessen work by directory (see next slide)
  – For write-miss of Shared (= non-modified) block: let directory send cacheblock value and list of sharing nodes to requestor and let requestor send invalidate requests to all nodes with a cacheblock copy to lessen work by directory (see next slide)
Example Directory Protocol (1st Read)

P1: pA

Id vA -> rd pA
{pA is page for add A}
Example Directory Protocol (Read Share)

P1: pA
P2: pA

Id vA -> rd pA
Id vA -> rd pA
Example Directory Protocol (Wr to shared)

D for dirty block, modified from memory copy

Read_to_update pA

Invalidate pA

P1: pA EX
P2: pA

reply xD(pA)

st vA -> wr pA

Inv ACK
Example Directory Protocol (Wr to Ex)

D for dirty block, modified from memory copy

P2: pA EX

Read_to_update pA

Write_back pA

Reply xD(pA)

st vA -> wr pA
Basic Directory Transactions To Let Remote CPU Do Much Coherency Work For Directory

1. Read request to directory
2. Directory node for block
3. Reply with owner identity
4a. Data Reply
4b. Revision message to directory

(a) Read miss to a block in dirty state

1. RdEx request to directory
2. Reply with sharers identity
3a. Inval. req. to sharer
3b. Inval. req. to sharer
4a. Inval. ack
4b. Inval. ack

(b) Write miss to a block with two sharers

Requestor

Directory node

Sharer

Node with dirty copy
A Popular Middle Ground

• Two-level “hierarchy”
• Individual nodes are multiprocessors, connected non-hierarchically
  – e.g. mesh of SMPs
• Coherence among nodes is directory-based
  – directory keeps track of nodes, not individual processors
• Coherence within nodes is snooping (or directory)
  – orthogonal, but needs a good interface of functionality
• SMP on a chip: support external directory + snoop internally?
Synchronization (for Shared Memory MPs)

• Why Synchronize? Need to know when it is safe for different processes to use shared data (or code)

• Issues for Synchronization:
  – Need uninterruptable instruction to fetch and update memory (an atomic operation);
  – User level synchronization operation using this primitive;
  – For large scale MPs, synchronization can be a bottleneck; need techniques to reduce system overhead from contention for same lock by several processors and the latency of synchronization
Uninterruptable Instructions to Fetch and Update Memory Values Used as Locks

- **Atomic exchange**: interchange a value in a register for a value in memory
  - 0 ⇒ synchronization variable is free
  - 1 ⇒ synchronization variable is locked and unavailable
    - Set register to 1 & swap
    - New value in register determines success in getting lock
      - 0 if processor (PU) succeeded in setting the lock (PU was first)
      - 1 if another processor had already claimed ownership of lock
    - Key is that exchange operation is indivisible (“atomic”) by other stores

- **Test-and-set**: sets(=>1) a lock value and tests prior lock value to see if PU has control of locked data (or code)
  - 0 ⇒ synchronization variable was free, so now owned by this PU
  - 1 ⇒ synchronization variable is owned (previously set) by another

- **Fetch-and-increment**: returns the prior value of a memory location & atomically increments it in memory
  - Use to give PU unique pointer to job in a task queue
Uninterruptable Instruction Pair LL SC to Fetch and Update Memory Atomically

- Cannot read & write in 1 mips instruction: use 2 instead.
- **Load linked** (or “load locked”) + store conditional
  - Load linked (\(ll\)) returns the initial value & sets an “addr. not stored” flag.
  - Store conditional returns 1 to “new value reg” if it succeeds (no other store to same memory location since preceding \(ll\)) and 0 otherwise.

**Example doing atomic swap (“exch”) with LL & SC:**

```assembly
try:   mov   R3,R4 ; put new exchange value in R3 (sc destroys R3)
  ll    R2,0(R1) ; load linked value from lock=>R2
  sc    R3,0(R1) ; store conditional if same, R3=>lock, 1=>R3
  beqz  R3,try ; retry if sc not store R3 value (so just 0=>R3)
  mov   R4,R2 ; put loaded prior lock value into R4
```

**Example doing fetch & increment with LL & SC:**

```assembly
try:   ll    R2,0(R1) ; load linked value from lock ctr==>R2
  addi  R2,R2,1 ; increment by 1 (OK, since fast if reg–reg)
  sc    R2,0(R1) ; store conditional: if same, ctr+1==>ctr, 1=>R3
  beqz  R2,try ; retry if store failed (not store ctr+1, 0=>R2)
```
User-Level Synchronization-Operation Using An Atomic Exchange Primitive

- **Spin locks**: processor continuously tries to acquire lock, spinning around a loop trying to find the lock free (=0)
  - `test&set`
  - `li R2,#1`
  - `lockit: exch R2,0(R1) ;atomic exchange`
  - `bnez R2,lockit ;already locked?`

- **What about MP (multiprocessor) with cache coherency?**
  - To avoid latency of accessing main memory, should spin on cache copy
  - Processors are likely to get cache hits for often used lock variables

- **Problem**: exchange includes a write, which invalidates all other copies and generates considerable bus traffic

- **Solution**: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):
  - `try: li R2,#1`
  - `lockit: lw R3,0(R1) ;load var`
  - `bnez R3,lockit ;≠ 0 ⇒ not free ⇒ spin`
  - `exch R2,0(R1) ;atomic exchange`
  - `bnez R2,try ;already locked?`
Fallacy: Scalability is almost free

- “build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small number of processors to a large number” False, all systems have bottlenecks.

- Cray T3E scales to 2048 CPUs vs. 4 CPU Alpha
  - At 128 CPUs, it delivers a peak bisection BW of 38.4 GB/s, or 300 MB/s per CPU (uses Alpha microprocessor)
  - Compaq Alphaserver ES40 up to 4 CPUs and has 5.6 GB/s of interconnect BW, or 1400 MB/s per CPU

- Building apps that scale requires significantly more attention to load balance, locality, potential contention, and serial (or partly parallel) portions of program. Speedup of 10X is very hard to achieve.
Pitfall: Not developing SW to take advantage (or optimize for) multiprocessor architecture

- SGI OS protects the page table data structure with a single lock, assuming that page allocation is infrequent
- Suppose a program uses a large number of pages that are initialized at start-up
- Program parallelized so that multiple processes allocate the pages
- But page allocation requires lock of page table data structure, so even an OS kernel that allows multiple threads will be serialized at initialization (even if separate processes)
Cautionary Tale

• Key to success of birth and development of ILP in 1980s and 1990s was software in the form of optimizing compilers that could exploit ILP

• Similarly, successful exploitation of TLP will depend as much on the development of suitable software systems as it will on the contributions of computer architects

• Given the slow progress on parallel software in the past 30+ years, it is likely that exploiting TLP broadly will remain challenging for years to come
And in Conclusion …

• Snooping and Directory Protocols are 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

• MPs are highly effective for multiprogrammed workloads

• MPs proved effective for CPU-intensive commercial workloads, such as OLTP (OnLine Transaction Processing, assuming enough I/O to be CPU-limited), DSS applications (Data Storage Server, where query optimization is critical), and large-scale, web searching applications
Unused slides Spring 2010
Another MP Issue: Memory Consistency Models

• What is consistency? **When** must a processor see the new value? e.g., the results of this code seem clear, but

  P1: \[ A = 0; \]
  \[ B1 = B; \]
  \[ \ldots \]
  \[ A = 1; \]
  \[ L1: \text{if (B == 0)} \ldots \]

  P2: \[ B = 0; \]
  \[ A2 = A; \]
  \[ \ldots \]
  \[ B = 1; \]
  \[ L2: \text{if (A == 0)} \ldots \]

• Is it impossible for both L1 & L2 *if* conditions to be true?
  – What if the write invalidate for A=1 on P1 is delayed in reaching P2, but both P1 & P2 continue on to execute their *if* statements L1 & L2?

• Memory consistency models: What are rules if accesses to different shared values (e.g., A & B) can cause errors?

• *(Safe) sequential consistency (SC)*: the result of any execution is the same as if all memory (read and write) accesses of each processor were kept in order and the accesses among different processors were interleaved in some order \(\Rightarrow\) all assignments done before the *ifs* above.
  – SC: delay all memory accesses until all caches complete all invalidates.
Relaxed Memory Consistency Models

• Relaxed schemes run faster than always-safe sequential consistency
• Not an issue for most parallel programs; they are synchronized.
  – A program is synchronized if all accesses to shared data are ordered by (slow) synchronization (locking, mutual exclusion) operations.

```plaintext
acquire (s) {lock}
...  
write (x)
...
release (s) {unlock}
...

acquire (s) {lock}
...
read(x)
...
release (s) {unlock}
```

• Only those fast programs willing to be nondeterministic \{outcome = f(processors’ speeds)\} are not synchronized ==> “data races”
• There are several Relaxed Models for Memory Consistency since most parallel programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses
Relaxed Consistency Models: The Basics

• **Key idea**: allow most reads and writes to complete out of order, but add synchronization operations to enforce ordering for critical accesses to distinct shared variables, so the partially synchronized program behaves as if its processors were sequentially consistent
  – By relaxing orderings, may obtain performance advantages (codes run faster).
  – Also specifies range of legal compiler optimizations on shared data
  – Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read/write pairs for two shared data items (A&B) because re-ordering (rwA,rwB=>rwB,rwA) might affect the results of the program

• There are three major sets of (from less to more) relaxed orderings:
  1. Relax W→R ordering (=> not all writes completed before next read)
     • Because it retains ordering among writes, many programs that assume sequential consistency operate well under this model, without additional synchronization. Called **processor consistency or Total Store Order**
  2. Relax W → W ordering (not all writes completed before next write)
  3. Relax R → W and R → R orders (many models with different ordering restrictions & rules for synchronization to enforce critical ordering)
     • Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when each processor can see the values that it has written.
Observation By Mark Hill

- Instead, can use speculation to avoid long access latencies of strict consistency models
  - If processor receives an invalidation for a memory reference before code involving it is committed, the processor uses speculation recovery to back out of its computation and restart with the invalidated memory reference (i.e., fetch the new value and recalculate).

1. Aggressive implementation of SC (sequential consistency) or PC (processor consist.) has most advantages of more relaxed models

2. Optimistic SC implementation adds little to the hardware costs of a speculative processor

3. Speculation allows the programmer to build fast codes using the more easily understood, but normally slower SC & PC models
Cross Cutting Issues: Performance Measurement of Parallel Processors

• Performance: how well scale as increase # Procs
• Speedup fixed as well as scaleup of problem
  – Assume benchmark of size $n_p$ on $p$ processors makes sense: how scale benchmark to $n_{mp}$ to run on $m \times p$ processors?
  – Memory-constrained scaling: keeping the amount of memory used per processor constant
  – Time-constrained scaling: keeping total execution time, assuming perfect speedup, constant

• Example: if 1 hour on 10 P, time $\sim O(n^3)$, what if 100 P?
  – Time-constrained scaling: 1 hour $\Rightarrow 10^{1/3}n \Rightarrow 2.15n$ scale up
  – Memory-constrained scaling: 10n size $\Rightarrow 10^3/10 \Rightarrow 100X$ or 100 hours!
    10X processors for 100X longer???
  – Need to know application well to scale: # iterations, error tolerance
Fallacy: Amdahl’s Law does not apply to parallel computers

• Since some part linear, cannot go above ~100X?

• 1987 claim to break it, since 1000X speedup for 1000p
  – researchers scaled the benchmark to have a data set size that was
    1000 times larger and compared the uniprocessor and parallel
    execution times for the scaled benchmark. For this particular
    algorithm the sequential portion of the program was constant
    independent of the size of the input, and the rest was fully parallel—
    hence, linear speedup with 1000 processors

• True speedup contests (the Gordon Bell prize) do not
  increase the data size as number of processors (PUs)
  increases; they also include data input times (time to
  distribute data from single disk to all PU’s memories).
Fallacy: Linear speedups are needed to make multiprocessors cost-effective

- Mark Hill & David Wood 1995 study
- Compare costs of SGI uniprocessor and MP systems
- Uniprocessor = $38,400 + $100 * MB
- MP = $81,600 + $20,000 * P + $100 * MB
- 1 GB RAM => Uni = $138k vs. MP = ($181k/P + $20k) * P
- What speedup for better MP cost performance? (if P>2)
  - 8 proc: $341k; $341k/$138k = 2.5X cost, 31% linear speedup
  - 16 proc ⇒ need only 3.6X cost, or 23% linear speedup
- Even if need some more memory for MP, memory size does not need to increase linearly with P
Answers to 1995 Questions about Parallelism

• In the 1995 edition of this text, we concluded the chapter with a discussion of two then current controversial issues.

1. What architecture would very large scale, microprocessor-based multiprocessors use?

2. What was the role for multiprocessing in the future of microprocessor architecture?

Answer 1. Large scale multiprocessors did not become a major and growing market ⇒ clusters of single microprocessors or moderate SMPs

Answer 2. Astonishingly clear. For at least for the next 5 years, future MPU performance comes from the exploitation of TLP through multicore processors vs. exploiting more ILP
Unused Slides 2009
State Transition Diagram for One Cache Block in Directory Based System

- States identical to snoopy case; transactions very similar
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss message to home directory
- Write misses that were broadcast on the bus for snooping ⇒ explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block
CPU - Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if only in memory or other remote cache(s)

Invalid

CPU Read hit

Invalid state if only in memory or other remote cache(s)

Shared (read/only)

CPU Read hit

Fetch/Invalidate
Send Data Write Back message to home directory

CPU Write:
Send Write Miss message to home directory

CPU Read:
Send Read Miss message

CPU Write:
Send Write Miss message to home directory

Fetch:
send Data Write Back message to home directory

CPU read miss => replace:
send Data Write Back message and Read miss to home directory

CPU write miss => replace:
send Data Write Back message and Write Miss to home directory

CPU read hit
CPU write hit

Shared (read/write)

CPU Read hit

Invalid state if only in memory or other remote cache(s)

Fetch/Invalidate
Send Data Write Back message to home directory

CPU Write:
Send Write Miss message to home directory

CPU Read:
Send Read Miss message

CPU Write:
Send Write Miss message to home directory

Fetch:
send Data Write Back message to home directory

CPU read miss => replace:
send Data Write Back message and Read miss to home directory

CPU write miss => replace:
send Data Write Back message and Write Miss to home directory

CPU read hit
CPU write hit
State Transition Diagram for Directory

- Same states & structure as the transition diagram for an individual cache
- **Two actions**: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, **Sharers**, as well as sending a message
**Directory State Machine**

- **State machine for** *Directory* **requests for each memory block**
- **Uncached state if in memory**

**States:**
- **Uncached**
- **Shared** (read only)
- **Dirty Exclusive** (read/write)

**Transitions:**
- **Read miss:** Sharers += {P}; send Data Value Reply
- **Write miss:** Sharers = {P}; send Data Value Reply msg
- **Write Miss:** Sharers = {P}; send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
- **Data Write Back:** Sharers = {} (Write back block)
- **Read miss:** Sharers += {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache
- **Write Miss:** Sharers = {P}; send Fetch; send Data Value Reply msg to remote cache (Write back block)
Example Directory Protocol

• Message sent to directory causes two actions:
  – Update the directory
  – More messages to satisfy request

• Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  – Read miss: requesting processor sent data from memory & requestor is made only sharing node; state of block made Shared.
  – Write miss: requesting processor is sent the value & becomes the only Sharing node. The block is made Exclusive to indicate that the only valid copy is in the remote cache. Sharers indicates the identity of the owner.

• Block is Shared ⇒ the memory value is up-to-date:
  – Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  – Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and Sharers vector is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol

- Block is **Exclusive**: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - **Read miss**: owner processor sent data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - **Data write-back**: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - **Write miss**: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.
### Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
### Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
<td><strong>Bus</strong></td>
<td><strong>Directory</strong></td>
<td><strong>Memor</strong></td>
</tr>
<tr>
<td><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
<td>P1</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
<td>P1</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
### Example

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 State</th>
<th>P1 Addr</th>
<th>P1 Value</th>
<th>P2 State</th>
<th>P2 Addr</th>
<th>P2 Value</th>
<th>Bus Action</th>
<th>Proc. State</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>DaRp</td>
<td>P1</td>
<td>A1</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrBk</td>
<td>P1</td>
<td>A1</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DaRp</td>
<td>P2</td>
<td>A1</td>
</tr>
</tbody>
</table>

Write Back since node(s) with shared cacheblock cannot pick one cache to send block copy to a new cache Read-Missing the block, so memory must send.

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
### Example

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Directory</th>
<th>Memor</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>State</td>
<td>Addr</td>
<td>Value</td>
<td>State</td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>Ftch</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv.</td>
<td></td>
<td></td>
<td>Inval.</td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
Example

**Processor 1**  **Processor 2**  **Interconnect**  **Directory**  **Memory**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td></td>
<td></td>
<td>DaRp</td>
<td>P1 A1</td>
<td>0</td>
<td>WrMs</td>
<td>P1 A1</td>
<td></td>
<td></td>
<td></td>
<td>Ex</td>
<td>{P1}</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td>Shar. A1 10</td>
<td>RdMs</td>
<td>P2 A1</td>
<td></td>
<td></td>
<td>Ftch</td>
<td>P1 A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl. A1 20</td>
<td></td>
<td>WrMs</td>
<td>P2 A1</td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Inval</td>
<td>P1 A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>Excl. {P2}</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Excl. A2 40</td>
<td>DaRp</td>
<td>P2 A2</td>
<td>A1 20</td>
<td></td>
<td>Wrbk</td>
<td>P2 A2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block position (but different memory block addresses: A1 ≠ A2)
Computers in the News

• “Core” new microarchitecture; last Pentium 4 (2000)
  – Wide Dynamic Execution: 4 issue + Combine 2 simple instructions into 1 powerful (“macrofusion”)
  – Advanced Digital Media Boost: All SSE instructions 1 clock cycle
  – Smart Memory Access: lets one core control the whole cache when the other core is idle, and governs how the same data can be shared by both cores
  – Intelligent Power Capability: shut down unneeded portions of chip

• 80% more performance, 40% less power

• 4 core chips in 2007 (2 copies of dual core?)
  – CTO: "Intel is taking a conservative approach that focuses on single-thread performance. You won't see mediocre thread performance just for the sake of getting multiple cores on a die."

• CTO urged software companies to support multicore designs with software that can efficiently divide tasks among multiple execution threads. "It's really time to get onboard the multithreaded train"