CS152 Announcements

- **Midterm 2** Thursday April 7- in lecture slot (306 Soda)
  - covers lectures 10-16, plus associated problem sets, labs, and readings
  - Topics: OoO, Branch prediction, VLIW, Multi-threading, Vectors, GPUs
  - One handwritten cheat-sheet
Uniprocessor Performance (SPECint)

Hennessey & Patterson [2018]
Parallel Processing: Déjà vu all over again?

- “... today’s processors ... are nearing an impasse as technologies approach the speed of light..”

- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years

- “We are dedicating all of our future product development to multicore designs. ... This is a sea change in computing”
  - Paul Otellini, President, Intel (2005)

- All microprocessor companies switch to MP (2+ CPUs/2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs

- Even handheld systems moved to multicore
  - Nintendo 3DS, iPhone6 had two cores each (plus additional specialized cores), Android Qualcomm Snapdragon 805 had four cores. Playstation Portable Vita had four cores.
Symmetric Multiprocessors (SMPs)

**symmetric**
- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

*Local caches at processors makes it practical!*
Modern Multi-processor System

DOI: 10.1145/3030207.3030223
Synchronization

The need for synchronization arises whenever there are concurrent processes in a system cooperating on some task

*(even in a uniprocessor system)*

Two classes of synchronization:

**Producer-Consumer:** A consumer process must wait until the producer process has produced data

**Mutual Exclusion:** Ensure that only one process uses a resource at a given time
Need for Mutual Exclusion

- Example (wikipedia): shared linked list management
- Two nodes, $i$ and $i + 1$, being removed simultaneously results in node $i + 1$ not being removed!!
Memory Coherence in SMPs

Suppose CPU-1 updates A to 200.
- write-back: memory and cache-2 have stale values
- write-through: cache-2 has a stale value

Do these stale values matter?
What is the view of shared memory for programming?
Maintaining Cache Coherence

- A cache coherence protocol ensures that all writes by one processor are eventually visible to the other processors
  - i.e., updates are not lost

- Hardware support is required such that
  - only one processor at a time has write permission for a location
  - no processor can load a stale copy of the location after a write
⇒ cache coherence protocols
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
Use snoopy mechanism to keep all processors’ view of memory coherent
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 by that cache before the memory is read
The MSI protocol

Each cache line has state bits

<table>
<thead>
<tr>
<th>Address tag</th>
<th>state bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>M: Modified</td>
<td></td>
</tr>
<tr>
<td>S: Shared</td>
<td></td>
</tr>
<tr>
<td>I: Invalid</td>
<td></td>
</tr>
</tbody>
</table>

- **Modified**: The block has been modified in the cache. The data in the cache is then inconsistent with the backing store (e.g. memory). A cache with a block in the "M" state has the responsibility to write the block to the backing store when it is evicted. A block in the Modified state is exclusive (it can’t be in any other cache).

- **Shared**: This block is unmodified and exists in read-only state in at least one cache. The cache can evict the data without writing it to the backing store.

- **Invalid**: This block is either not present in the current cache or has been invalidated by a bus request, and must be fetched from memory if the block is to be stored in this cache.
The MSI protocol

Each cache line has state bits

<table>
<thead>
<tr>
<th>State</th>
<th>Address tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>M: Modified</td>
<td></td>
</tr>
<tr>
<td>S: Shared</td>
<td></td>
</tr>
<tr>
<td>I: Invalid</td>
<td></td>
</tr>
</tbody>
</table>

- A **read miss** to a block in a cache, C1, generates a bus transaction – if another cache, C2, has the block in M state ("exclusively"), it has to write back the block before memory supplies it. C1 gets the data from the bus and the block becomes "shared" in both caches.
- A **write hit to a shared block** in C1 forces a write back – all other caches that have the block should invalidate that block – the block becomes M "exclusive" in C1.
- A **write hit to a modified (exclusive) block** does not generate a write back or change of state.
- A **write miss (to an invalid block)** in C1 generates a bus transaction
  - If a cache, C2, has the block as "shared", C2 invalidates it’s copy
  - If a cache, C2, has the block in “modified (exclusive)”, it writes back the block and changes it state in C2 to “invalid”.
  - If no cache supplies the block, the memory will supply it.
  - When C1 gets the block, it sets its state to “modified (exclusive)”
Each cache line has state bits

- **M**: Modified
- **S**: Shared
- **I**: Invalid

### Address tag

State bits

#### State transition diagram:

- **M**: Modified
  - Read miss (P₁ gets line from memory)
  - Other processor reads (P₁ writes back)
  - P₁ reads or writes

- **S**: Shared
  - Read miss (P₁ gets line from memory)
  - Read by any processor
  - Other processor intent to write

- **I**: Invalid
  - Other processor intent to write (P₁ writes back)
  - P₁ intent to write
  - Cache state in processor P₁
Two Processor Example
(Reading and writing the same cache line)

P₁ reads
P₁ writes
P₂ reads
P₂ writes
P₁ reads
P₁ writes
P₂ writes
P₁ writes

P₁

P₂

M

S

I

Read miss
Read miss
Write miss
P₂ reads, P₁ writes back
P₁ reads, P₂ writes back
P₂ reads, P₁ writes back

P₁ reads or writes
P₂ reads or writes
P₂ reads or writes

P₂ intent to write
P₁ intent to write
P₁ intent to write
P₂ intent to write
P₂ intent to write

P₂ writes
P₂ writes
P₂ writes
P₂ writes

P₂ reads
P₂ reads
P₂ reads
P₂ reads

P₁ writes
P₁ writes
P₁ writes
P₁ writes

P₁ returns
P₁ returns
P₁ returns
P₁ returns
Observations

- 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
- A write to a line in the $S$ state causes a writeback (even if no other cache has a copy!)
**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

- **Write to a Exclusive line doesn’t cause a writeback.**

---

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Modified Exclusive</td>
</tr>
<tr>
<td>E</td>
<td>Exclusive but unmodified</td>
</tr>
<tr>
<td>S</td>
<td>Shared</td>
</tr>
<tr>
<td>I</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

- **Write miss**
  - P₁ write or read
  - Other processor reads
  - P₁ writes back
  - Read miss, shared
  - Read by any processor

- **P₁ write**
  - Other processor intent to write
  - Other processor reads
  - Other processor intent to write, P₁ writes back

- **P₁ read**
  - Read miss, not shared
  - Other processor intent to write

---

CS152, Spring 2022
Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)

Inclusion property: entries in L1 must be in L2
  - Miss in L2 $\Rightarrow$ Not present in L1
  - Only if invalidation hits in L2 $\Rightarrow$ probe and invalidate in L1

Snooping on L2 does not affect CPU-L1 bandwidth
When a read-miss for A occurs in cache-2, Cache-2 initiates a read request for A 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
A cache line contains more than one word.

Cache-coherence is done at the line-level and not word-level.

Suppose $M_1$ writes $\text{word}_i$ and $M_2$ writes $\text{word}_k$ and $i \neq k$ but 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
    $\Rightarrow$ miss would not occur if line size were 1 word
Example: True v. False Sharing v. Hit?

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

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
<th>True, False, Hit? Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write x1</td>
<td></td>
<td>True miss; invalidate x1 in P2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read x2</td>
<td>False miss; x1 irrelevant to P2</td>
</tr>
<tr>
<td>3</td>
<td>Write x1</td>
<td></td>
<td>False miss; x1 irrelevant to P2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write x2</td>
<td>True miss; x2 not writeable</td>
</tr>
<tr>
<td>5</td>
<td>Read x2</td>
<td></td>
<td>True miss; x2 invalid in P1</td>
</tr>
</tbody>
</table>
• Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)

• True sharing and false sharing unchanged going from 1 MiB to 8 MiB (L3 cache)
MP Performance 2MiB 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
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. InvRep arrives at cache.
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!
Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252