Lecture Outline

• Intro to caches review
• Write Policies and Write Buffers
• Cache Performance
• How to improve cache performance?
  – Reducing Cache Miss Penalty
What is a cache?

- Small, fast storage used to improve average access time to slow memory
- Hold subset of the instructions and data used by program
- Exploits spacial and temporal locality

Program locality is why caches work

- Memory hierarchy exploit program locality:
  - Programs tend to reference parts of their address space that are local in time and space
  - Temporal locality: recently referenced addresses are likely to be referenced again (reuse)
  - Spatial locality: If an address is referenced, nearby addresses are likely to be referenced soon
- Programs that don’t exploit locality won’t benefit from caches
Where do misses come from?

• Classifying Misses: 3 Cs

  – **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses. (*Misses in even an Infinite Cache*)

  – **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. (*Misses in Fully Associative Size X Cache*)

  – **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (*Misses in N-way Associative, Size X Cache*)

---

Cache Examples: Cycles 1 – 5

**Spatial Locality!**

```
0,1,2,3,4,5,6,7,8,9,0,0,2,2,2,4,9,1,9,1
```

<table>
<thead>
<tr>
<th>Miss</th>
<th>Hit (1)</th>
<th>Miss</th>
<th>Hit (3)</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>2 3</td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 5</td>
</tr>
</tbody>
</table>
### General View of Caches

- **Cache is made of frames**
  - Frame = data + tag + state bits
  - State bits: Valid (tag/data there), Dirty (wrote into data)

- **Cache Algorithm**
  - Find frame(s)
  - If incoming tag != stored tag then Miss
    - Evict block currently in frame
    - Replace with block from memory (or L2 cache)
  - Return appropriate word within block

---

### Basic Cache Organization

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>state~</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Block Frames organized into **sets**

Number of Frames (**ways**) in each set is **associativity**

- One Frame per set (1 column) = Direct Mapped
### Mapping Addresses to Frames

<table>
<thead>
<tr>
<th>Tag (T)</th>
<th>Index (N)</th>
<th>Offset (O)</th>
</tr>
</thead>
</table>

Divide Address into offset, index, tag

- **Offset**: finds word within a cache block
  - \( O \)-bit offset \( \Leftrightarrow 2^O \)-byte block size
- **Index**: finds set containing block frame
  - \( N \)-bit offset \( \Leftrightarrow 2^N \) sets in cache
  - Direct Mapped Cache: Index finds frame directly
- **Tag**: Remaining bits not implied by block frame, must match

### Direct Mapped Caches

- **Partition Memory Address into three regions**
- \( C \) = Cache Size
- \( M \) = Numbers of bits in memory address
- \( B \) = Block Size

\[
\begin{align*}
M & \log C & \log C/B & \log B \\
\text{Tag} & & \text{Index} & \text{Block Offset} \\
\end{align*}
\]

- **Diagram**: Shows the flow of data and tag memory for a direct mapped cache.
Set Associative Caches

- Partition Memory Address into three regions
  - \( C \) = Cache Size, \( B \) = Block Size, \( A \) = number of members per set
  - \( M \cdot \log \frac{C}{A} \), \( \log \frac{C}{B \cdot A} \), \( \log B \)

<table>
<thead>
<tr>
<th>Tag Memory</th>
<th>Index</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Memory

= Hit/Miss

Data

\[ \text{Tag} \rightarrow \text{Index} \rightarrow \text{Block Offset} \]

Cache Example

- 32-bit machine
- 4KB, 16B Blocks, direct-mapped cache
  - 16B Blocks => 4 Offset Bits
  - 4KB / 16B Blocks => 256 Frames
  - 256 Frames / 1 –way (DM) => 256 Sets => 8 index bits
  - 32-bit address – 4 offset bits – 8 offset bits => 20 tag bits
Another Example

- 32-bit machine
- 64KB, 32B Block, 2-Way Set Associative
- Compute Total Size of Tag Array
  - 64KB / 32B blocks => 2K Blocks
  - 2K Blocks / 2-way set-associative => 1K Sets
  - 32B Blocks => 5 Offset Bits
  - 1K Sets => 10 index bits
  - 32-bit address – 5 offset bits – 10 index bits = 17 tag bits
  - 17 tag bits * 2K Blocks => 34Kb => 4.25KB

Summary of Set Associativity

- Direct Mapped
  - One place in cache, One Comparator, No Muxes
- Set Associative Caches
  - Restricted set of places
  - N-way set associativity
  - Number of comparators = number of blocks per set
  - N:1 mux
- Fully Associative
  - Anywhere in cache
  - Number of comparators = number of blocks in cache
  - N:1 mux needed
More Detailed Questions

- Block placement policy?
  - Where does a block go when it is fetched?
- Block identification policy?
  - How do we find a block in the cache?
- Block replacement policy?
  - When fetching a block into a full cache, how do we decide what other block gets kicked out?
- Write strategy?
  - Does any of this differ for reads vs. writes?

Block Placement + ID

- Placement
  - Invariant: block always goes in exactly one set
  - Fully-Associative: Cache is one set, block goes anywhere
  - Direct-Mapped: Block goes in exactly one frame
  - Set-Associative: Block goes in one of a few frames
- Identification
  - Find Set
  - Search ways in parallel (compare tags, check valid bits)
**Block Replacement**

- Cache miss requires a replacement
- No decision needed in direct mapped cache
- More than one place for memory blocks in set-associative
- Replacement Strategies
  - Optimal
    - Replace Block used furthest ahead in time (oracle)
  - Least Recently Used (LRU)
    - Optimized for temporal locality
  - (Pseudo) Random
    - Nearly as good as LRU, simpler

**Write Policies**

- Writes are only about 21% of data cache traffic
- Optimize cache for reads, do writes “on the side”
  - Reads can do tag check/data read in parallel
  - Writes must be sure we are updating the correct data and the correct amount of data (1-8 byte writes)
  - Serial process $\Rightarrow$ slow
- What to do on a write hit?
- What to do on a write miss?
### Write Hit Policies

- **Q1:** When to propagate new values to memory?
- **Write back** – Information is only written to the cache.
  - Next lower level only updated when it is evicted (dirty bits say when data has been modified)
  - Can write at speed of cache
  - Caches become temporarily inconsistent with lower-levels of hierarchy.
  - Uses less memory bandwidth/power (multiple consecutive writes may require only 1 final write)
  - Multiple writes within a block can be merged into one write
  - Evictions are longer latency now (must write back)

---

### Write Hit Policies

- **Q1:** When to propagate new values to memory?
- **Write through** – Information is written to cache and to the lower-level memory
  - Main memory is always “consistent/coherent”
  - Easier to implement – no dirty bits
  - Reads never result in writes to lower levels (cheaper)
  - Higher bandwidth needed
  - Write buffers used to avoid **write stalls**
Write buffers

• Write buffers are essential for write-through caches

- Small chunks of memory to buffer outgoing writes
- Processor can continue when data written to buffer
- Allows overlap of processor execution with memory update

Write buffers

• Writes can now be pipelined (rather than serial)
  • Check tag + Write store data into Write Buffer
  • Write data from Write buffer to L2 cache (tags ok)
• Loads must check write buffer for pending stores to same address
• Loads Check:
  • Write Buffer
  • Cache
  • Subsequent Levels of Memory
**Write Merging**

**Non-merging Buffer**
- Except for multi-word write operations, extra slots are unused

**Merging Write Buffer**
- More efficient writes
- Reduces buffer-full stalls

---

**Write buffer policies:**
**Performance/Complexity Tradeoffs**

- Allow merging of multiple stores? (“coalescing”)
- “Flush Policy” – How to do flushing of entries?
- “Load Servicing Policy” – What happens when a load occurs to data currently in write buffer?
Write Buffer Flush Policies

- When to flush?
  - Aggressive flushing => Reduce chance of stall cycles due to full write buffer
  - Conservative flushing => Write Merging more likely (entries stay around longer) => reduces memory traffic
  - On-chip L2’s => More aggressive flushing
- What to flush?
  - Selective flushing of particular entries?
  - Flush everything below a particular entry
  - Flush everything

Write Buffer Load Service Policies

- Load op’s address matches something in write buffer
- Possible policies:
  - Flush entire write buffer, service load from L2
  - Flush write buffer up to and including relevant address, service from L2
  - Flush only the relevant address from write buffer, service from L2
  - Service load from write buffer, don’t flush
- What if a Read miss doesn’t hit in the Write buffer?
  - Give priority for the Read L2 accesses over the Write L2 Accesses
Write misses?

- **Write Allocate**
  - Block is allocated on a write miss
  - Standard write hit actions follow the block allocation
  - Write misses = Read Misses
  - Goes well with write-back
- **No-write Allocate**
  - Write misses do not allocate a block
  - Only update lower-level memory
  - Blocks only allocate on Read misses!
  - Goes well with write-through

### Summary of Write Policies

<table>
<thead>
<tr>
<th>Write Policy</th>
<th>Hit/Miss</th>
<th>Writes to</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteBack/Allocate</td>
<td>Both</td>
<td>L1 Cache</td>
</tr>
<tr>
<td>WriteBack/NoAllocate</td>
<td>Hit</td>
<td>L1 Cache</td>
</tr>
<tr>
<td>WriteBack/NoAllocate</td>
<td>Miss</td>
<td>L2 Cache</td>
</tr>
<tr>
<td>WriteThrough/Allocate</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>WriteThrough/NoAllocate</td>
<td>Hit</td>
<td>Both</td>
</tr>
<tr>
<td>WriteThrough/NoAllocate</td>
<td>Miss</td>
<td>L2 Cache</td>
</tr>
</tbody>
</table>
Cache Performance

CPU time = (CPU execution cycles + Memory Stall Cycles)*Clock Cycle Time

AMAT = Hit Time + Miss Rate * Miss Penalty

- Reducing these three parameters can have a big impact on performance
- Out-of-order processors can hide some of the miss penalty

Reducing Miss Penalty

- Have already seen two examples of techniques to reduce miss penalty
  - Write buffers give priority to read misses over writes
  - Merging write buffers
    - Multiword writes are faster than many single word writes
- Now we consider several more
  - Victim Caches
  - Critical Word First/Early Restart
  - Multilevel caches
Reducing Miss Penalty: Victim Caches

- Direct mapped caches => many conflict misses
- Solution 1: More associativity (expensive)
- Solution 2: Victim Cache
- Victim Cache
  - Small (4 to 8-entry), fully-associative cache between L1 cache and refill path
  - Holds blocks discarded from cache because of evictions
  - Checked on a miss before going to L2 cache
  - Hit in victim cache => swap victim block with cache block

Reducing Miss Penalty: Victim Caches

- Even one entry helps some benchmarks!
- Helps more for smaller caches, larger block sizes
Reducing Miss Penalty: Critical Word First/Early Restart

- CPU normally just needs one word at a time
- Large cache blocks have long transfer times
- Don’t wait for the full block to be loaded before sending requested data word to the CPU

- Critical Word First
  - Request the missed word first from memory and send it to the CPU and continue execution

- Early Restart
  - Fetch in order, but as soon as the requested block arrives send it to the CPU and continue execution

Reducing Miss Penalty: Multilevel Caches

- Should the L1 cache be faster to keep up with the CPU or larger to overcome processor-memory gap?
- Both
  - L1 cache is small and fast
  - L2 cache is large to capture many main memory accesses

\[
AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}
\]
\[
\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} + \text{Miss Penalty}_{L2}
\]
\[
AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} + \text{Miss Penalty}_{L2})
\]
L2 Cache Performance

- Local miss rate = \# misses / \#refs to cache
- Global miss rate = \#misses / \#refs of CPU
- Local miss rate of L2 cache is usually not very good because most locality has been filtered out by the L1 Cache

Multilevel Caches

- For L2 Caches
  - Low latency, high bandwidth is less important
  - Low miss rate is very important
  - Why?
- L2 Caches design for
  - Unified (I+D)
  - Larger Size (4-8MB) at the expense of latency
  - Larger block sizes (128Byte lines!)
  - High associativity: 4, 8, 16 at the expense of latency
Multilevel Inclusion

- Inclusion is said to hold between level i and level i+1 if all data in level i is also in level i+1
- Desirable because for I/O and multiprocessors only have to keep 2\textsuperscript{nd} level consistent
- Inclusion must be maintained by flushing blocks in 1\textsuperscript{st} level that are mapped to a particular 2\textsuperscript{nd} level line when it is replaced
- Difficult when different block sizes at various levels

Next Time

- More Cache Performance
- Reducing Miss Rate
- Reducing Hit Time
- Reducing Miss Penalty/\textsuperscript{Rate} via parallelism