CMSC 611: Advanced Computer Architecture

Cache
Temporal Locality (Locality in Time):
⇒ Keep most recently accessed data items closer to the processor

Spatial Locality (Locality in Space):
⇒ Move blocks consists of contiguous words to the faster levels

Speed: Fastest
Size: Smallest
Cost: Highest

Slowest
Biggest
Lowest
Memory Hierarchy Terminology

- **Hit**: data appears in some block in the faster level (example: Block X)
  - Hit Rate: the fraction of memory access found in the faster level
  - Hit Time: Time to access the faster level which consists of
    - Memory access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the slower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- **Hit Time << Miss Penalty**

![Diagram of Memory Hierarchy]

- Slower Level Memory
- Faster Level Memory
- Block X
- Block Y

To Processor

From Processor
Memory Hierarchy Design Issues

- **Block identification**
  - How is a block found if it is in the upper (faster) level?
    - Tag/Block

- **Block placement**
  - Where can a block be placed in the upper (faster) level?
    - Fully Associative, Set Associative, Direct Mapped

- **Block replacement**
  - Which block should be replaced on a miss?
    - Random, LRU

- **Write strategy**
  - What happens on a write?
    - Write Back or Write Through (with Write Buffer)
The Basics of Cache

- Cache: level of hierarchy closest to processor
- Caches first appeared in research machines in early 1960s
- Virtually every general-purpose computer produced today includes cache

**Issues:**
- How do we know that a data item is in cache?
- If so, How to find it?

Requesting $X_n$ generates a miss and the word $X_n$ will be brought from main memory to cache.

<table>
<thead>
<tr>
<th>X4</th>
<th></th>
<th>X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td></td>
<td>X1</td>
</tr>
<tr>
<td>$X_{n-2}$</td>
<td></td>
<td>$X_{n-2}$</td>
</tr>
<tr>
<td>$X_{n-1}$</td>
<td></td>
<td>$X_{n-1}$</td>
</tr>
<tr>
<td>X2</td>
<td></td>
<td>X2</td>
</tr>
<tr>
<td>X3</td>
<td></td>
<td>X3</td>
</tr>
</tbody>
</table>
Worst case is to keep replacing a block followed by a miss for it: Ping Pong Effect

To reduces misses:
- make the cache size bigger
- multiple entries for the same Cache Index

Cache block address = (Block address) modulo (Number of cache blocks)
Accessing Cache

- Cache Size depends on:
  - # cache blocks
  - # address bits
  - Word size

- Example:
  - For n-bit address, 4-byte word & 1024 cache blocks:
    - cache size = 1024 \[(n-10 -2) + 1 + 32\] bit
Cache with Multi-Word/Block

- Takes advantage of spatial locality to improve performance
- Cache block address = (Block address) modulo (Number of cache blocks)
- Block address = (byte address) / (bytes per block)
Determining Block Size

- Larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- Average Access Time =
  Hit Time \( \times \) (1 - Miss Rate) + Miss Penalty \( \times \) Miss Rate

![Graphs showing relationships between block size, miss penalty, miss rate, and average access time.](Image)

- Miss Penalty rises with block size.
- Miss Rate decreases with block size, exploiting spatial locality.
- Average Access Time decreases, with increased Miss Penalty and Miss Rate.

Exploits Spatial Locality

Fewer blocks: compromises temporal locality
## Block Placement

**Hardware Complexity**

**Cache utilization**

### Direct mapped

<table>
<thead>
<tr>
<th>Block #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Set associative

<table>
<thead>
<tr>
<th>Set #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fully associative

<table>
<thead>
<tr>
<th>Data</th>
<th>Tag</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>↑</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>↑</td>
</tr>
</tbody>
</table>

- Set number = (Block number) modulo (Number of sets in the cache)
- Increased flexibility of block placement reduces probability of cache misses
N-way Set Associative Cache

- N entries for each Cache Index
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result
Locating a Block in Associative Cache

Tag size increases with higher level of associativity
Fully Associative Cache

- Forget about the Cache Index
- Compare the Cache Tags of all cache entries in parallel
- Example: Block Size = 32 Byte blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache
Handling Cache Misses

- Read misses bring blocks from memory
- Write access requires careful maintenance of consistency between cache and main memory
- Two write strategies:
  - Write through: write to both cache and memory
    - Read misses cannot result in writes
    - No allocation of a cache block is needed
    - Always combined with write buffers so that don’t wait for slow memory
  - Write back: write cache only; write to memory when cache block is replaced
    - Is block clean or dirty?
    - No writes to slow memory for repeated write accesses
    - Requires allocation of a cache block
- Processor writes data into the cache and the write buffer
- Memory controller writes contents of the buffer to memory
- Increased write frequency can cause saturation of write buffer
- If CPU cycle time too fast and/or too many store instructions in a row:
  - Store buffer will overflow no matter how big you make it
  - The CPU Cycle Time get closer to DRAM Write Cycle Time
- Write buffer saturation can be handled by installing a second level (L2) cache
Empirical results indicates less significance of replacement strategy with increased cache sizes

- Straight forward for Direct Mapped since every block has only one location
- Set Associative or Fully Associative:
  - Random: pick any block
  - LRU (Least Recently Used)
    - requires tracking block reference
    - for two-way set associative cache, reference bit attached to every block
    - more complex hardware is needed for higher level of cache associativity

<table>
<thead>
<tr>
<th>Associativity</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

- Empirical results indicates less significance of replacement strategy with increased cache sizes