Caches

- Why is caching needed?
  - Technological development and Moore’s Law

- Why are caches successful?
  - Principle of locality

- Three basic models and how they work (in detail)
  - Direct mapped, fully associative, set associative

- How they interact with the pipeline

- Performance analysis of caches
  - Average Memory Access Time (AMAT)

- Enhancements to caches

---

Since 1980, CPU has outpaced DRAM ...

---

1977: DRAM actually faster than microprocessors!!

Apple ][ (1977)

CPU: 1000 ns
DRAM: 400 ns

---

How do computer architectures address the gap?

Put smaller, faster "cache" memories between CPU and DRAM. Create a "memory hierarchy".
The Goal: illusion of large, fast, cheap memory

- Fact: Large memories are slow, fast memories are small
- How do we create a memory that is large, cheap and fast (most of the time)?
  • Hierarchy
  • Parallelism

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access.
The Principle of Locality

- Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 15 years, HW relied heavily on locality for speed

Locality is a universal property of programs which is exploited in many aspects of HW and SW design.

Memory Hierarchy Use of Locality

- Temporal Locality (Locality in Time):
  => Keep most recently accessed data items closer to the processor
- Spatial Locality (Locality in Space):
  => Move blocks of contiguous words to the upper levels
Memory Hierarchy: Terminology

° Hit: data appears in some block in the upper level
  • Hit Rate: the fraction of memory access found in the upper level
  • Hit Time: Time to access the upper level which consists of
    Time to determine hit/miss + time to deliver block to processor
° Miss: data needs to be retrieved from a block in the lower level
  • Miss Rate = 1 - (Hit Rate)
  • Miss Time: Time to determine hit/miss + Time to replace a block in
    the upper level + Time to deliver the block to the processor
  • Miss Penalty: Extra time incurred for a miss = Time to replace a
    block in the upper level

° Hit Time << Miss Penalty and Miss Time

Four Organizing Principles for Caches and Memory Hierarchy

° Q1: Where can a block be placed in the upper level? (Block placement)
° Q2: How is a block found if it is in the upper level? (Block identification)
° Q3: Which block should be replaced on a miss? (Block replacement)
° Q4: What happens on a write? (Write strategy)

Q1: Where can a block be placed in the upper level?

° Block 12 placed in 8 block cache:
  • Fully associative, direct mapped, 2-way set associative

Example: 1 KB Direct Mapped Cache with 32 B Blocks

° For a $2^N$ byte cache:
  • The uppermost (32 - N) bits are always the Cache Tag
  • The lowest M bits are the Byte Select (Block Size = $2^M$)
Example: Fully Associative

- Fully Associative Cache
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32 B blocks, we need N 27-bit comparators

```
31 4 0
Cache Tag (27 bits long) Byte Select
```

Q2: How is a block found if it is in the upper level?

- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag

Example: Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<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>
Q4: What happens on a write?

- **Writes occur less frequently than reads:**
  - Under MIPS: 7% of all memory traffic are writes
  - 25% of all data traffic are writes
- Thus, Amdahl’s Law implies that caches should be optimized for reads. However, we cannot ignore writes.
- **Problems with writes:**
  - Must check tag BEFORE writing into the cache
  - Only a portion of the cache block is modified
  - Write stalls - CPU must wait until the write completes

---

Q4: What happens on a write: Design Options

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Is block clean or dirty?
- **Pros and Cons of each?**
  - **WT:** read misses don’t cause writes; easier to implement; copy of data always exists
  - **WB:** write at the speed of the cache; multiple writes to cache before write to memory; less memory BW consumed

---

Write Buffer for Write Through

- **A Write Buffer is needed between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- **Memory system designer’s nightmare:**
  - Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - Write buffer saturation

---

Write Buffer Saturation

- **Store frequency (w.r.t. time) > 1 / DRAM write cycle**
  - If this condition exist for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - The CPU Cycle Time <= DRAM Write Cycle Time
- **Solution for write buffer saturation:**
  - Use a write back cache
  - Install a second level (L2) cache: (does this always work?)
Write Miss Design Options

° Write allocate ("fetch on write") - block is loaded on a write miss, followed by the write.

° No-write allocate ("write around") - block is modified in the lower level, not loaded into the cache.

Write-miss Policy: Write Allocate versus Not Allocate

° Assume: a 16-bit write to memory location 0x0 and causes a miss

  • Do we read in the block?
    - Yes: Write Allocate
    - No: Write Not Allocate

Impact on Cycle Time

Cache Hit Time: directly tied to clock rate
increases with cache size
increases with associativity

Average Memory Access time (AMAT) =
Hit Time + Miss Rate x Miss Penalty

Compute Time = IC x CT x (ideal CPI + memory stalls)

Example: direct map allows miss signal after data

What happens on a Cache miss?

° For in-order pipeline, 2 options:
  - Freeze pipeline in Mem stage (popular early on: Sparc, R4000)
    IF ID EX Mem stall stall stall stall .. stall Mem Wr
    IF ID EX stall stall stall .. stall Mem Ex Wr
  - Use Full/Empty bits in registers + MSHR queue
    - MSHR = "Miss Status/Handler Registers" (Kroft)
      Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.
      - Per cache-line: keep info about memory address.
      - For each word: register (if any) that is waiting for result.
      - Used to "merge" multiple requests to one memory line
    - New load creates MSHR entry and sets destination register to "Empty". Load is "released" from pipeline.
    - Attempt to use register before result returns causes instruction to block in decode stage.
    - Limited "out-of-order" execution with respect to loads.

° Out-of-order pipelines already have this functionality built in... (load queues, etc.).