CAS CS 210: Computer Systems

Rich West

Memory
(Chapter 6)
Exploiting Memory Hierarchy

Users want large and fast memories! As of 2015:

**SRAM** access times are 1.3ns at cost of $25/MB
- Used for CPU register files, caches, device buffers (e.g., LCD displays)
- Static (doesn’t need refresh) RAM built from flip-flops

**DRAM** access times are 20ns at cost of $0.02/MB
- Dynamic (needs refresh), used for main memory
- Stores individual bits in a capacitive circuit

**Disk** access times are 3 million ns at cost of $0.03/GB
Memory Hierarchy

- Try and give impression of one large & fast memory
  - although a memory hierarchy

Levels in the memory hierarchy

Size of the memory at each level

Increasing distance from the CPU in access time
Locality

- A principle that makes having a memory hierarchy a good idea
- If an item is referenced,

**temporal locality:** it will tend to be referenced again soon

**spatial locality:** nearby items will tend to be referenced soon.

*Why does code have locality?*
Locality

- Our initial focus: two levels (upper, lower)
  - block*: minimum unit of data
  - hit: data requested is in the upper level
  - miss: data requested is not in the upper level

- *NB: For caches, a block is also known as a “line”. A cache line (block) is a single unit of transfer between memory and the cache
Cache

- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?

- Cache management strategy:
  - "direct mapped"
  - Others include fully/set-associative
Direct Mapped Cache

- For each item of data at the lower level, there is exactly one location in the cache where it might be.
  e.g., lots of items at the lower level share same location in the upper level

- Mapping: address modulo the number of blocks in the cache
Direct Mapped Cache

BLOCK
TAG
CACHE
INDEX

00001 00101 01001 01101 10001 10101 11001 11101
Cache Addressing

- A cache entry is:
  
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data block</th>
<th>Flag bit(s)</th>
</tr>
</thead>
</table>

- Flag = (in)valid (i-cache), (in)valid + dirty/clean (d-cache)

- Address to cache breaks down to:
  
<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
</table>

- Index identifies cache line
- Block offset identifies the byte or word within the cache block
Example
(Cache Line Accesses)

- Cache C1 direct-mapped, 16 one-word blocks
- Cache C2 direct-mapped, 4 four-word blocks
- Assume empty initially, obtain number of misses for reference string: 0, 4, 8, 11 (word addresses)
Example

- Cache C1 direct-mapped, 16 one-word blocks
- Cache C2 direct-mapped, 4 four-word blocks
- Assume empty initially, obtain number of misses for reference string: 0, 4, 8, 11 (word addresses)

- **Answer:** C1: 4 misses; C2: 3 misses
- “cold” misses happen initially when cache is empty
Writing Cache Friendly Code

**Example:** assume cold cache, 4-byte words, 4-word cache blocks, cache size=N/4 blocks, N multiple of 4. Compute miss ratio?

```c
int sumarrayrows(int a[N][N])
{
    int i, j, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sumarraycols(int a[N][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            sum += a[i][j];
    return sum;
}
```
Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- **Example:** assume cold cache, 4-byte words, 4-word cache blocks, cache size=N/4 blocks, N multiple of 4.

```c
int sumarrayrows(int a[N][N])
{
    int i, j, sum = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

int sumarraycols(int a[N][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            sum += a[i][j];
    return sum;
}
```

Miss ratio = \(1/4 = 25\%\)

Miss ratio = 100%

Initially cold misses, then conflict misses
Hits vs. Misses

- **Read hits**
  - this is what we want!

- **Read misses**
  - stall the CPU, read block from memory into cache, restart
Hits vs. Misses

- **Write hits**
  - can replace data in cache and memory (**write-through**)
  - write the data only into the cache (**write-back** the cache later when it is replaced)

- **Write misses**
  - stall the CPU, read block from memory into cache, restart

*Why might a load cause memory system to write?*

*Why might a store cause memory system to read?*
Performance

- Simplified model:

  \[
  \text{program execution time} = (\text{execution cycles} + \text{stall cycles}) \times \text{cycle time}
  \]

  \[
  \text{stall cycles} = \# \text{ of instructions} \times \text{miss ratio} \times \text{miss penalty}
  \]

- execution cycle includes detecting cache hit
- miss penalty includes detecting the cache miss
Performance

- One way of improving performance:
  - decreasing the miss ratio

What happens if we increase block size?
Example

- Cache C1 direct-mapped, 16 one-word blocks
  - C1 miss penalty = 8 clock cycles
- Cache C2 direct-mapped, 4 four-word blocks
  - C2 miss penalty = 11 clock cycles
- Assume empty initially, obtain number of misses and miss cycles for reference string: 0, 4, 8, 11 (word addresses)

- **Answer:** C1: 4 misses, 32 miss cycles; C2: 3, 33
Example

- 500Mhz machine
- 5% miss ratio
- 200ns miss penalty
- 1 cycle per instruction if any memory reference hits in cache
- How long does a program of “I” instructions take?

**Answer:**

I instructions = I x 1 CPI = I (execution) cycles;
Stall cycles = I x 0.05 x Miss Penalty;
Miss Penalty = 200ns = 500x10^6 x 200x10^{-9} = 100000x10^{-3} = 100 cycles
=> Total = I + I x 0.05 x 100 cycles
=> Total = 6I x 2ns // 500x10^6 = 2ns/cycle
Direct Mapped Cache for 32-bit byte (physical) address

TAG
15 bits

<table>
<thead>
<tr>
<th>cache index</th>
<th>word offset</th>
<th>byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>3 bits</td>
</tr>
</tbody>
</table>

V D TAG
four 8-byte words = 32 bytes = 256 bits

4K entries

byte

HIT
Cache Size vs. Implementation Cost

- Cache size in previous example = 128KB
- Number of bits to implement it includes not only data, but also tags, dirty and valid bits
- For previous example, total number of bits is $4K \times (1 + 1 + 15 + 256) = 1,092Kb = 136.5KB > 128KB$
  - 17 bits overhead per cache line
Cache Size vs. Implementation Cost

- Cache size in previous example = 128KB
- 128KB = 128x1024/32 lines = 128x32=4096 lines
- 4096 lines requires 12 bits for index ($2^{12}=4096$)
- 5 bits of 32-bit address are used for byte offset in cache line
- Therefore, tag bits = 32 – (12 + 5) = 15 bits
- Total cache size = 4096 lines x 1 (valid bit) x 1 (dirty bit) x 15 (tag bits) x 256 (block bits)