Introduction to Caching

COS 316 Lecture 8

Amit Levy
Figure 1: CPU Connected Directly to Memory
How long to run this code?

Characteristics

• CPU Instructions: 0.5ns (2GHz)

• Memory access: 100ns

```c
int arr[1000];
for (i = 0; i < arr.len(); i++) { ++arr[i]; }

loop: ldr r2, [r0, #0]
    add r2, r2, #1
    str r2, [r0, #0]
    subs r0, r0, #4
    bne <loop>
```
How long to run this code?

```
loop: ldr r2, [r0, #0]
     add r2, r2, #1
     str r2, [r0, #0]
     subs r0, r0, #4
     bne <loop>
```

1. \(2.5\mu S \ (2, 500\,nS)\)
2. \(30\mu S \ (30, 000\,nS)\)
3. \(201.5\mu S \ (201, 500\,ns)\)
Why not just make everything fast?

<table>
<thead>
<tr>
<th>Type</th>
<th>Access Time</th>
<th>Typical Size</th>
<th>$/MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>&lt; 0.5 ns</td>
<td>~256 bytes</td>
<td>$1000</td>
</tr>
<tr>
<td>SRAM/&quot;Cache&quot;</td>
<td>5 ns</td>
<td>1-4 MB</td>
<td>$100</td>
</tr>
<tr>
<td>DRAM/&quot;Memory&quot;</td>
<td>50 ns</td>
<td>GBs</td>
<td>$0.01</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>5 ms</td>
<td>TBs</td>
<td>$0.000001</td>
</tr>
</tbody>
</table>

- High cost of fast storage
- Physical limitations
- Not necessarily possible—e.g. accessing a web page across the world
What is caching?

- Keep *all* data in bigger, cheaper, slower storage
- Keep *copies* of “active” data in smaller, more expensive, faster storage

*Figure 2: CPU + Cache + Memory*
What do we cache?

- Data stored verbatim in slower storage
- Previous computations—recomputation is also a kind of slow storage
- Examples:
  - CPU memory hierarchy
  - File system page buffer
  - Content distribution network
  - Web application cache
  - Database cache
  - Memoization
Locality

- Temporal locality: nearness in time
  - Data accessed now probably accessed recently
  - Useful data tends to continue to be useful

- Spatial locality: nearness in name
  - Data accessed now “near” previously accessed data
  - Memory addresses, files in the same directory, frames in a video...
### When is caching effective?

Which of these workloads could we cache effectively?

<table>
<thead>
<tr>
<th>Repeated Access</th>
<th>Random Access</th>
<th>Sequential access</th>
</tr>
</thead>
<tbody>
<tr>
<td>A few popular items</td>
<td>No pattern to accesses</td>
<td>Access items in order</td>
</tr>
<tr>
<td>E.g. most social media</td>
<td>E.g. large hash tables</td>
<td>E.g. streaming a video</td>
</tr>
</tbody>
</table>
Caching Terminology

- **Hit**: when a requested item was in the cache
- **Miss**: when a requested item was *not* in the cache
- **Hit rate** and **Miss rate**: proportion of hits and misses, respectively
- **Hit time** and **Miss time**: time to access item in cache and not in cache, respectively
Effective access time is a function of:

- Hit and miss rates
- Hit and miss times

\[ t_{effective} = (hit\_rate) t_{hit} + (1 - hit\_rate) t_{miss} \]

aka, Average Memory Access Time (AMAT)
• *Effective access time*

• Look-aside vs. Look-through

• Write-through vs. Write-back

• Allocation policy

• Eviction Policy
Who handles misses?

What happens when a requested item is not in the cache?

Figure 3: User requests an item not in the cache
Look-aside

Advantages: easy to implement, flexible

Disadvantages: application handles consistency, can be slower on misses

Figure 4: Look-aside Cache
Look-through Cache

- Advantages: helps maintain consistency, simple to program against
- Disadvantages: harder to implement, less flexible

Figure 5: Look-through Cache
Handling Writes

- Caching creates a replica/copy of the data
- When you write, the data needs to be synchronized *at some point*
  - But when?
Write-through

Write to backing store on every update

• Advantages:
  • Cache and memory are always consistent
  • Eviction is cheap
  • Easy to implement

• Disadvantages:
  • Writes are at least as slow as writes to the backing store
Update only in the cache. Write “back” to the backing store only when evicting item from cache

- Advantages:
  - Writes always at cache speed
  - Multiple writes to same item combined
  - Batch writes of related items

- Disadvantages:
  - More complex to maintain consistency
  - Eviction is more expensive
When writing to items *not* currently in the cache, do we bring them into the cache?

**Yes == Write-Allocate**
- Advantage: Exploits temporal locality: written data likely to be accessed again soon

**No == Write-No-Allocate**
- Advantage: Avoids spurious evictions if data not accessed soon
Eviction policies

Which items to we evict from the cache when we run out of space?

Many possible algorithms:

- Least Recently Used (LRU), Most Recently Used (MRU)
- Least Frequently Used (LFU)
- First-In-First-Out (FIFO), Last-In-First-Out (LIFO)
- ...

Deciding factors include:

- Workload
- Performance
Challenges in Caching

- Speed: making the cache itself fast
- Cache Coherence: dealing with out-of-sync caches
- Performance: maximizing hit rate
- Security: avoiding information leakage through the cache
• Caching in the CPU Memory Hierarchy
• File system page buffer
• Caching in the Web (Prof. Freedman)
• Assignment 3: Implement a look-aside, write-allocate cache