Introduction to Caching

COS 316: Principles of Computer System Design

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

Characteristics

• CPU Instructions & Register accesses: 0.5ns (2GHz)

• Memory access: 50ns

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

    mov r3, #1000

loop: ldr r1, [r0]
    subs r3, r3, #1
    add r1, r1, #1
    str r1, [r0], #4
    bne <loop>
```
How long to run this code?

```
mov r3, #1000
loop: ldr r1, [r0]
    subs r3, r3, #1
    add r1, r1, #1
    str r1, [r0], #4
    bne <loop>
```

<table>
<thead>
<tr>
<th>CPU instruction</th>
<th>0.5ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access</td>
<td>50ns</td>
</tr>
</tbody>
</table>

1. $2.5\mu S \text{ (2,505\,nS)}$
2. $250\mu S \text{ (250,000\,nS)}$
3. $101.5\mu S \text{ (201,505\,ns)}$
How long to run this code?

```
mov r3, #1000
loop: ldr r1, [r0]
    subs r3, r3, #1
    add r1, r1, #1
    str r1, [r0], #4
    bne <loop>
```

1. $2.5 \mu S$ ($2,505 nS$)
2. $250 \mu S$ ($250,000 nS$)
3. $101.5 \mu S$ ($201,505 nS$)

<table>
<thead>
<tr>
<th>CPU instruction</th>
<th>0.5ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access</td>
<td>50ns</td>
</tr>
</tbody>
</table>

**Solution**

In each loop iteration:

- 2 instructions manipulate registers ($0.5 nS$)
- 3 instructions manipulate memory ($100 nS$)

$$1 \times 0.5 + 1000 \times (3 \times 0.5 + 2 \times 50) = 101,505 nS$$
## 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>Solid state</td>
<td>20 μs</td>
<td>TBs</td>
<td>$0.0001</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>5 ms</td>
<td>10-100s TB</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
A Solution: Caching

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
How long to run this code?

```
 mov r3, #1000
 loop: ldr r1, [r0]
 subs r3, r3, #1
 add r1, r1, #1
 str r1, [r0], #4
 bne <loop>
```

1. 2.5μS
2. 11.5μS
3. 101.5μS

---

CPU instruction 0.5ns
CPU cache access 5ns
Memory access 50ns

It's complicated!
We don't have enough information to answer. Yet!
How long to run this code?

```assembly
mov r3, #1000
loop: ldr r1, [r0]
        subs r3, r3, #1
        add r1, r1, #1
        str r1, [r0], #4
        bne <loop>
```

<table>
<thead>
<tr>
<th>CPU instruction</th>
<th>0.5ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cache access</td>
<td>5ns</td>
</tr>
<tr>
<td>Memory access</td>
<td>50ns</td>
</tr>
</tbody>
</table>

It’s complicated!

1. 2.5μS
2. 11.5μS
3. 101.5μS

We don’t have enough information to answer. Yet!
Caching Effectiveness

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

- Hit and miss ratio
- Hit and miss times

\[ t_{\text{effective}} = (hit_{ratio})t_{hit} + (1 - hit_{ratio})t_{miss} \]

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

• Look-aside vs. Look-through

• Write-through vs. Write-back

• Write-allocation

• 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
Look-through

Advantages: helps maintain consistency, simple to program against

Disadvantages: harder to implement, less flexible
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
Write-back

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
Write-allocate vs. Write-no-allocate

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 is 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 ratio
- Security: avoiding information leakage through the cache
• Caching in the CPU Memory Hierarchy
• CDN Caching
• From the research: Learning Relaxed Belady
• Next assignment: in-memory Web application cache