Introduction to Caching

COS 316: Principles of Computer System Design

Amit Levy & Jennifer Rexford

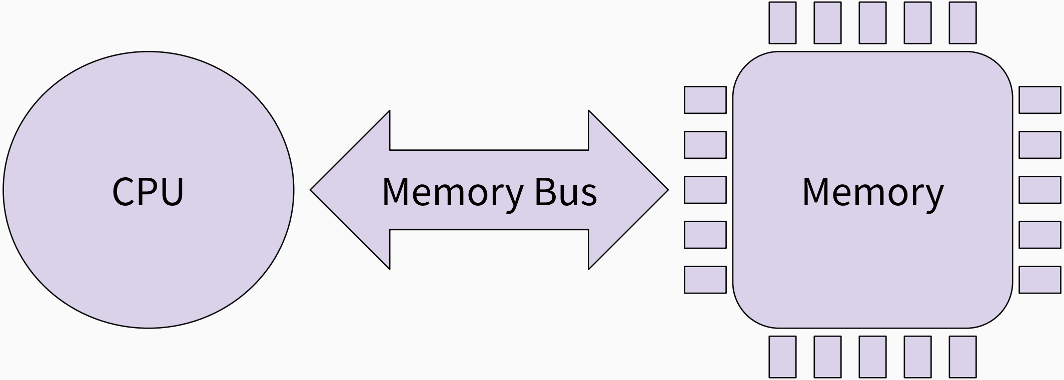


Figure 1: CPU Connected Directly to Memory

How long to run this code?

Characteristics

- CPU Instructions & Register accesses: 0.5ns (2GHz)
- Memory access: 50ns

```
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>
```

CPU instruction	0.5ns
Memory access	50ns

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

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

CPU instruction	0.5ns
Memory access	50ns

Solution

In each loop interation:

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

$$1*0.5+1000*(3*0.5+2*50) = 101,505ns$$

Why not just make everything fast?

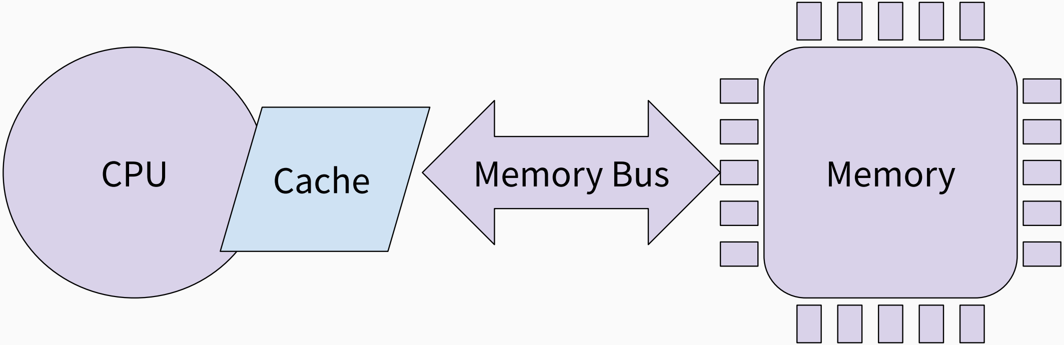
Type	Access Time	Typical Size	$/MB
Registers	$< 0.5ns$	~256 bytes	$\$1000$
SRAM/"Cache"	$5ns$	1-4MB	$\$100$
DRAM/"Memory"	$50ns$	GBs	$\$0.01$
Solid state	$20\mu S$	TBs	$\$0.0001$
Magnetic Disk	$5ms$	10-100s TB	$\$0.000001$

- 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



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
  - Domain Name System (DNS)
  - Content distribution network (CDN)
  - Web application cache
  - Database cache

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>
```

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

1. $2.5\mu S$
2. $11.5\mu S$
3. $101.5\mu S$

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. $11.5\mu S$
3. $101.5\mu S$

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

It's complicated!

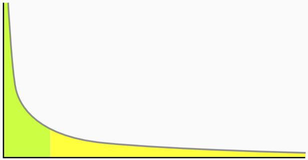
We don't have enough information to answer. Yet!

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

Which of these workloads could we cache effectively?

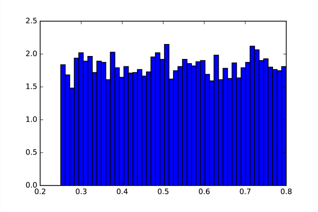
Repeated Access



A few popular items

E.g. most social media

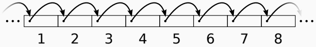
Random Access



No pattern to accesses

E.g. large hash tables

Sequential access



Access items in order

E.g. streaming a video

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

Effective access time is a function of:

- Hit and miss ratio
- Hit and miss times

$$t_{effective} = (hit\_ratio)t_{hit} + (1 - hit\_ratio)t_{miss}$$

aka, Average Memory Access Time (AMAT)

Characterizing a Caching System

Design decisions affect what cache is suitable for, also, how to effectively *use* a cache.

- *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?

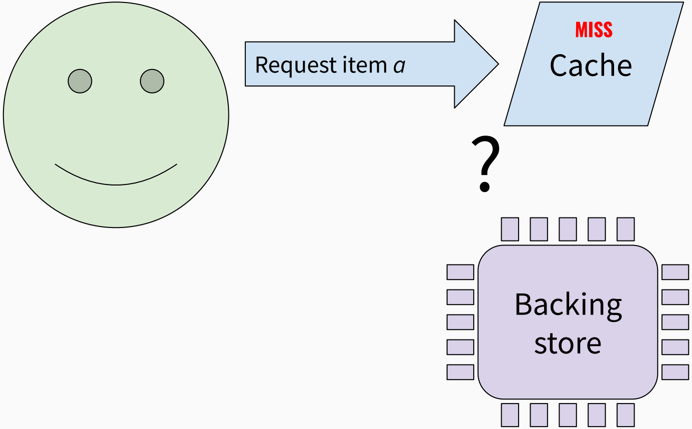


Figure 3: User requests an item not in the cache

Look-aside

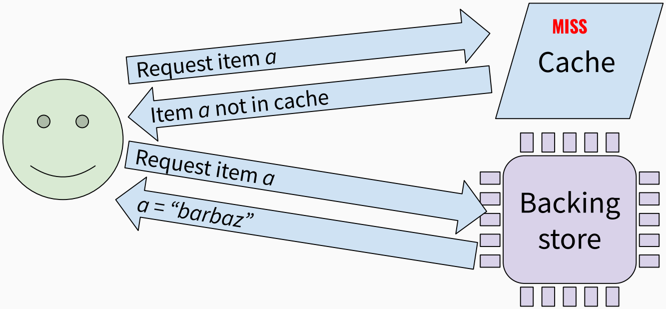


Figure 4: Look-aside Cache

- Advantages: easy to implement, flexible
- Disadvantages: application handles consistency, can be slower on misses

Look-through

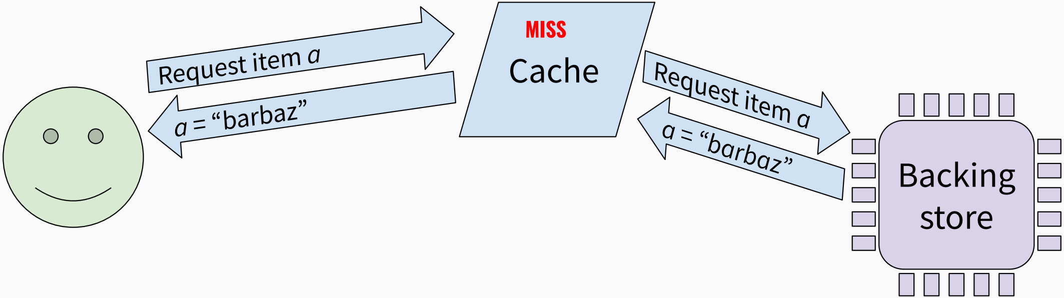


Figure 5: Look-through Cache

- 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

Characterizing a Caching System

- Effective access time
- Look-aside vs. Look-through
- Write-through vs. Write-back
- Write-allocation
- Eviction Policy

Characterizing a Caching System

- Effective access time
- Look-aside vs. Look-through
- Write-through vs. Write-back
- Write-allocation
- Eviction Policy

Affect designers and applications

- Where and how to add a cache to a system
  - Can a cache be effective given expected workload?
  - Maintain consistency?
  - Could applications more effectively cache on their own?
- How to effectively *use* caches from applications
  - How should I organize data?
  - Do I need to worry about consistency?
  - How will performance scale with larger workloads?

Remainder of this Section

- Caching in the CPU Memory Hierarchy
- CDN Caching
- From the research: TBD
- Next assignment: in-memory Web application cache

References