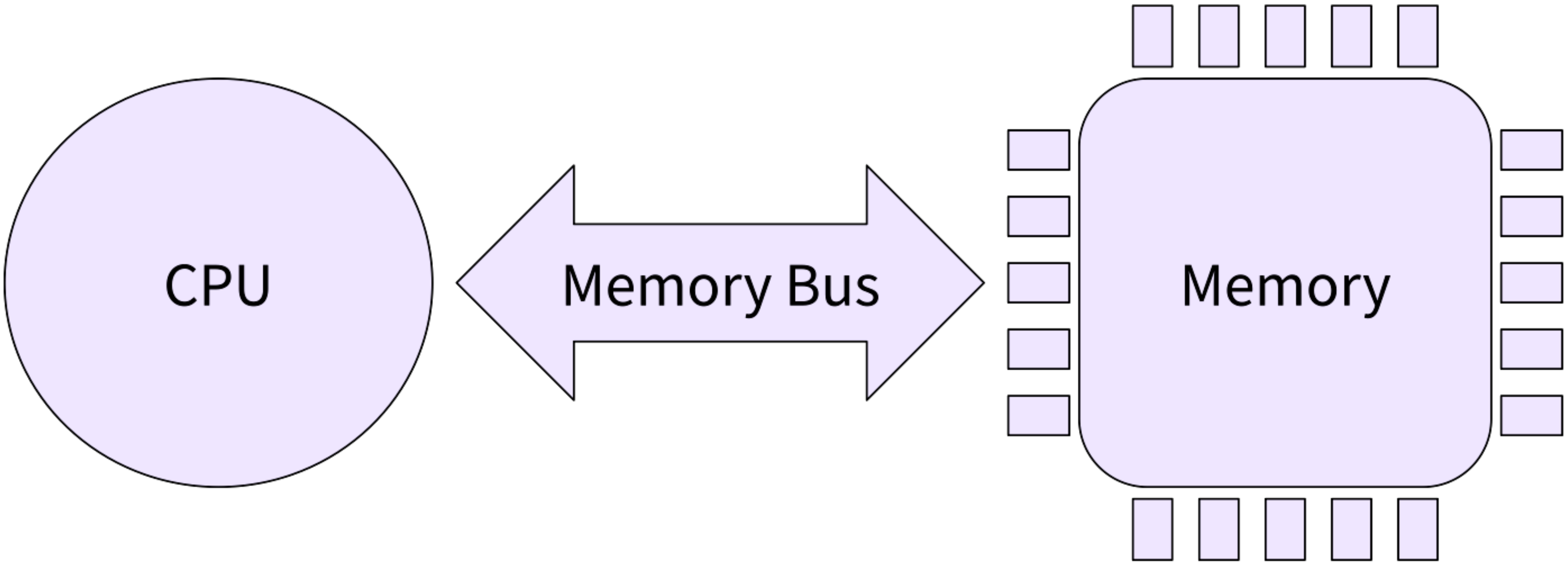
Introduction to Caching

COS 316: Principles of Computer System Design

Lecture 7

Amit Levy & Ravi Netravali



CPU connected directly to memory

How long to run this code?

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

```
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>
```

1. 2.5 microseconds (2,505 ns)
2. 250 microseconds (250,000 ns)
3. 101 microseconds (101,000.5 ns)

How long to run this code?

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

```
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>
```

1. 2.5 microseconds (2,505 ns)
2. 250 microseconds (250,000 ns)
3. 101 microseconds (101,000.5 ns)

1*0.5 + 1000*(2*50 + 2*0.5) = 101,000.5 ns

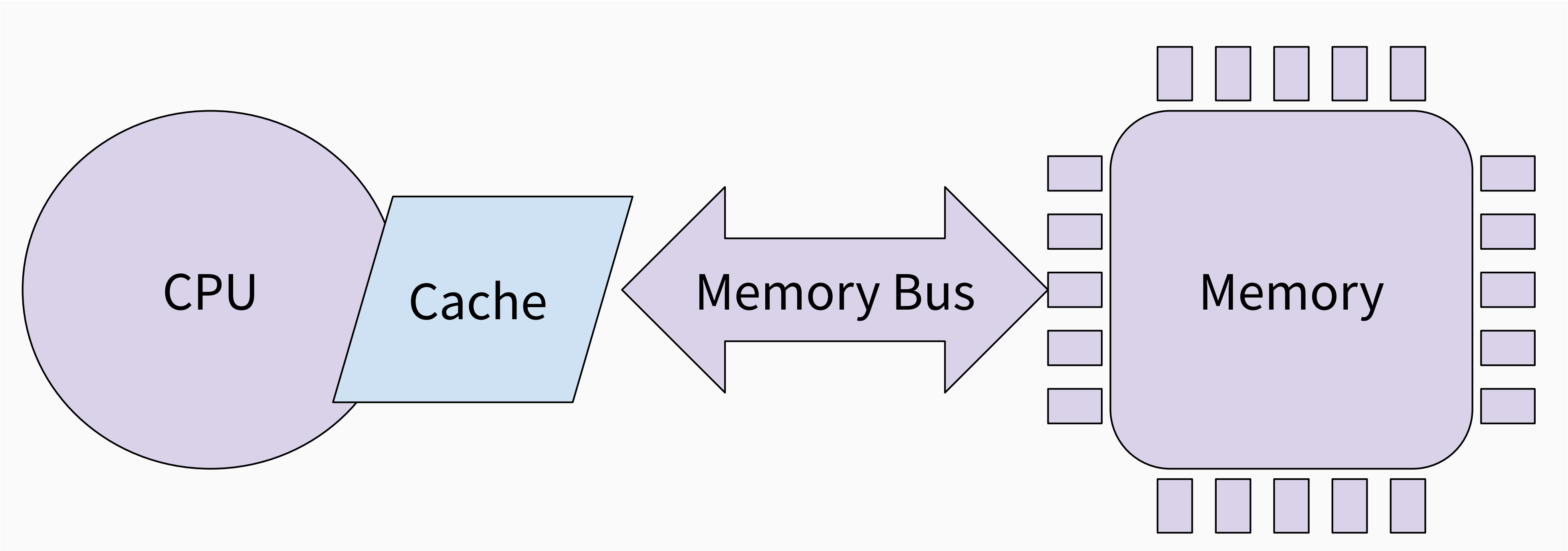
Why not just make everything fast?

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 for fast storage (inverse relationship between cost and performance)!

A Solution: 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 – recomputations are a kind of `slow storage’
- Examples
  - CPU memory hierarchy
  - File system page buffer
  - Domain Name System (DNS)
  - Content Distribution Networks (CDN)
  - Web browser caches
  - Database caches

How long to run this code?

- Characteristics
  - CPU Instructions & Register accesses: 0.5 ns (2 GHz)
  - **CPU cache accesses: 5 ns**
  - Memory accesses: 50 ns

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

It's complicated -- not enough info to answer this yet!

Evaluating cache effectiveness

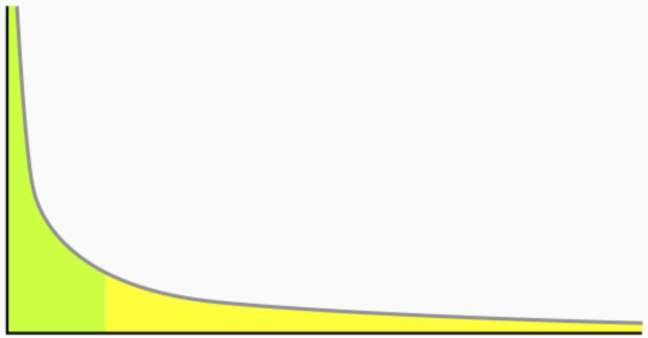
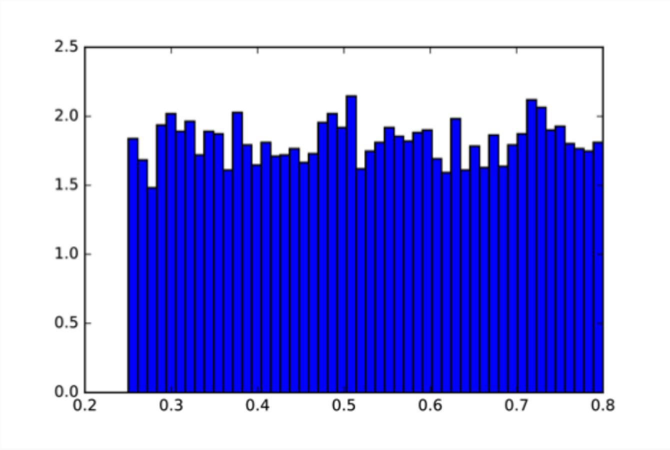
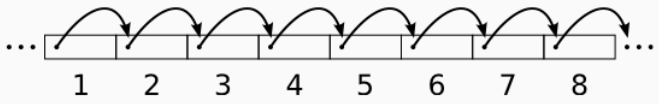
- **Hit**: when a requested item was in the cache
- **Miss**: when a requested item was *not* in the cache

- **Hit ratio** and **Miss ratio**: proportion of hits and misses, respectively

- **Hit time** and **Miss time**: time to access item in the cache and not in the cache, respectively

When is caching effective?

- Which of these workloads could we cache effectively?

Repeated Access	Random Access	Sequential access
		
A few popular items	No pattern to accesses	Access items in order
E.g. most social media	E.g. large hash tables	E.g. streaming a video

What influences cache effectiveness?

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

- **Spatial locality**: nearness in name
  - Data accessed now is “near” previously accessed data
  - Memory addresses, files in the same directory, frames in a video, etc.

Effective access time

- Effective access time is a function of:
  - Hit and Miss ratio
  - Hit and Miss times

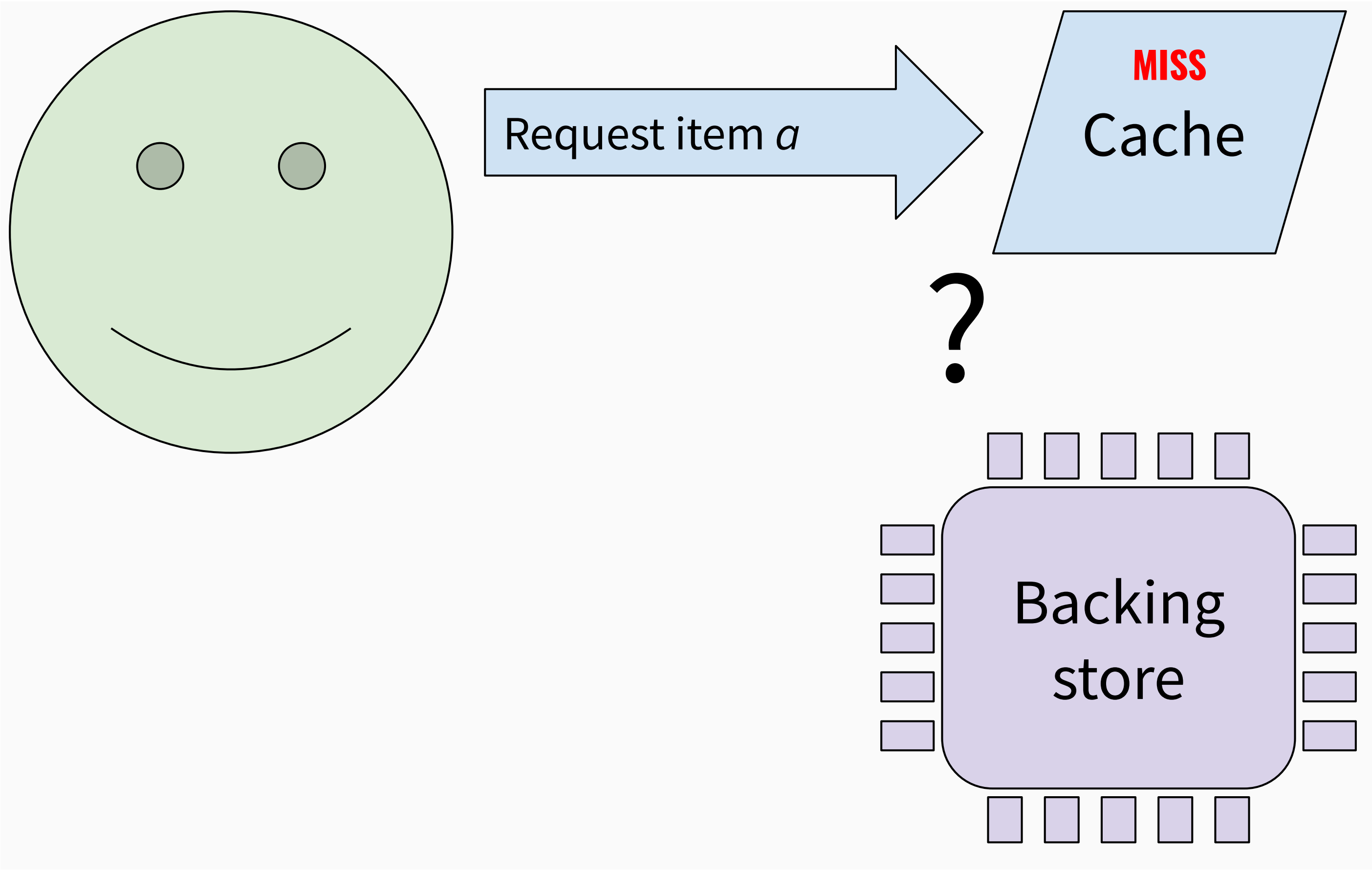
- $t_{effective}$ = (hit_ratio)*t_{hit} + (1– hit_ratio) * t_{miss}
  - Also referred to as AMAT (Average Memory Access Time)

Characterizing a caching system

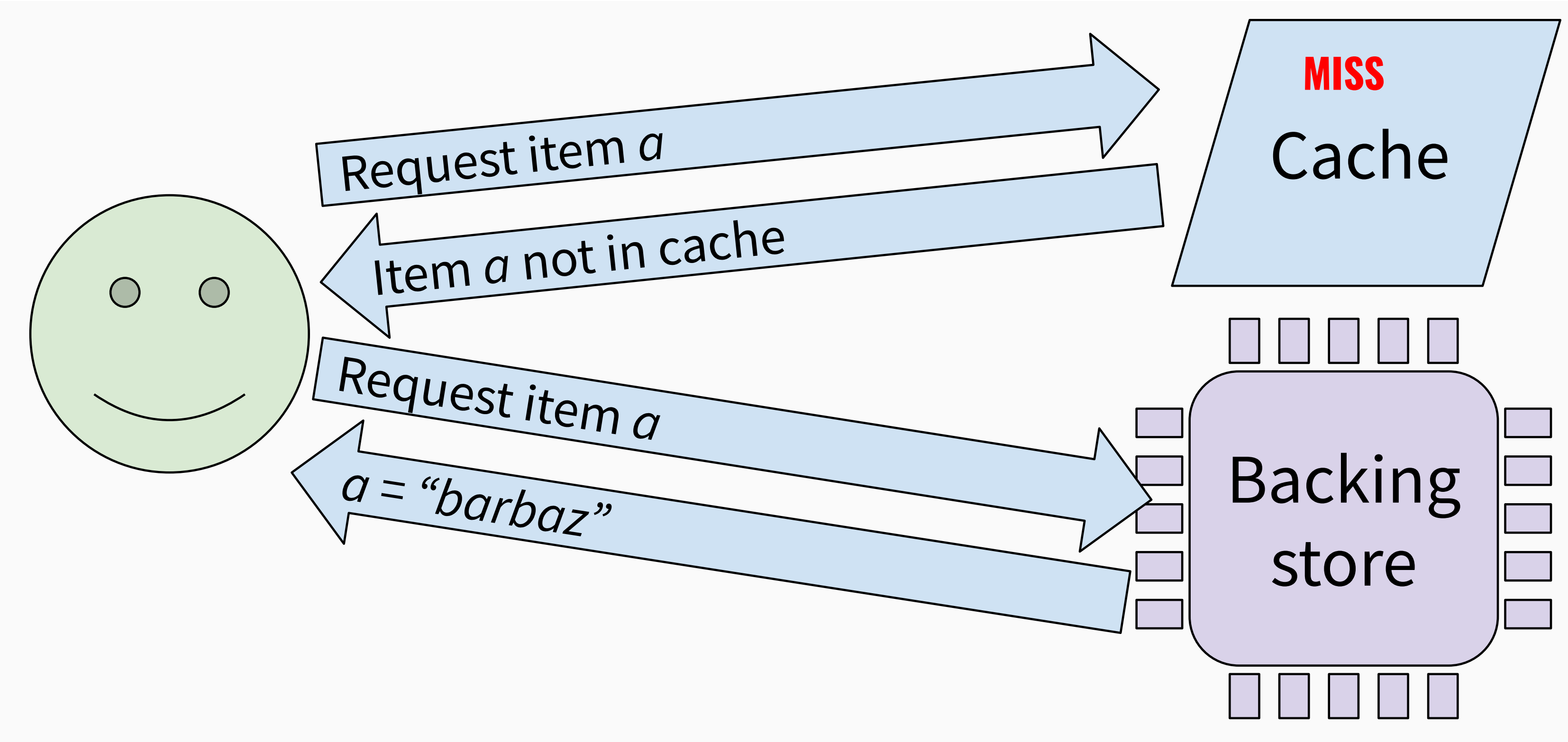
- Properties that affect what cache is suitable for *and* how to effectively use a cache
  - Effective access time
  - Look-aside vs. Look-through
  - Write-through vs. Write-back
  - Write-allocation
  - Eviction policy

Who handles misses?

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

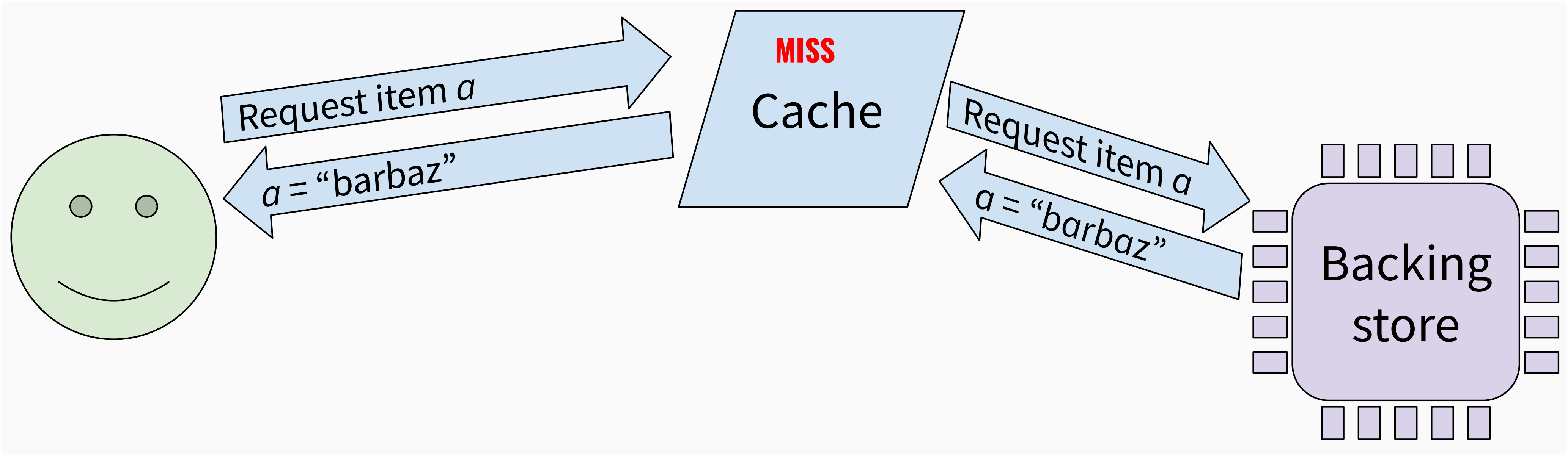


Look-aside



- 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 to 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 since written data is 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 evict from cache when we run out of space?

- Many 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: workload and performance requirements

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-allocate vs. Write-no-allocate
- Eviction policy

Useful for designers of caches and application developers (using caches)!

Remainder of this section

- Caching in the CPU memory hierarchy
- CDN (Web) Caching
- Research: cache optimizations in mobile apps (compute and network)
- Next assignment: in-memory web application cache