

# The memory hierarchy: Cache memories

Yipeng Huang

Rutgers University

April 10, 2023

# Table of contents

## Announcements

## Locality: How to create illusion of fast access to capacious data

- Spatial locality

- Temporal locality

## Caches: motivation

- Hardware caches supports software locality

- Software locality exploits hardware caches

## Cache placement policy (how to find data at address for read and write hit)

- Fully associative cache

- Direct-mapped cache

- Set-associative cache

## PA5: Simulating a cache and optimizing programs for caches

# Announcements

## Class session plan

- ▶ Monday, 4/10: Cache Memories (Book chapter 6.4)
- ▶ Thursday, 4/13: Cache-Friendly Code–cache blocking (Book chapters 6.5 and 6.6)
- ▶ Monday, 4/17: Cache-Friendly code–cache oblivious algorithms

# Table of contents

## Announcements

## Locality: How to create illusion of fast access to capacious data

- Spatial locality

- Temporal locality

## Caches: motivation

- Hardware caches supports software locality

- Software locality exploits hardware caches

## Cache placement policy (how to find data at address for read and write hit)

- Fully associative cache

- Direct-mapped cache

- Set-associative cache

## PA5: Simulating a cache and optimizing programs for caches

# Locality: How to create illusion of fast access to capacious data

From the perspective of memory hierarchy, locality is using the data in at any particular level more frequently than accessing storage at next slower level.

First, let's experience the puzzling effect of locality in `sumArray.c`

- ▶ `sumArrayRows()`
- ▶ `sumArrayCols()`

Well-written programs maximize locality

- ▶ Spatial locality
- ▶ Temporal locality

# Spatial locality

---

```
1 double dotProduct (  
2     double a[N],  
3     double b[N],  
4 ) {  
5     double sum = 0.0;  
6     for(size_t i=0; i<N; i++){  
7         sum += a[i] * b[i];  
8     }  
9     return sum;  
10 }
```

---

## Spatial locality

- ▶ Programs tend to access adjacent data.
- ▶ Example: stride 1 memory access in a and b.

# Temporal locality

---

```
1 double dotProduct (  
2     double a[N],  
3     double b[N],  
4 ) {  
5     double sum = 0.0;  
6     for(size_t i=0; i<N; i++){  
7         sum += a[i] * b[i];  
8     }  
9     return sum;  
10 }
```

---

## Temporal locality

- ▶ Programs tend to access data over and over.
- ▶ Example: `sum` gets accessed  $N$  times in iteration.

# Table of contents

## Announcements

## Locality: How to create illusion of fast access to capacious data

- Spatial locality

- Temporal locality

## Caches: motivation

- Hardware caches supports software locality

- Software locality exploits hardware caches

## Cache placement policy (how to find data at address for read and write hit)

- Fully associative cache

- Direct-mapped cache

- Set-associative cache

## PA5: Simulating a cache and optimizing programs for caches



# CPU / cache / DRAM main memory interface

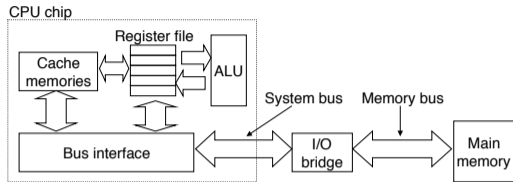


Figure: Cache resides on CPU chip close to register file. Image credit CS:APP

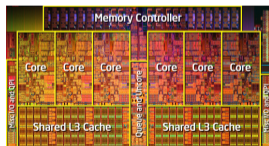
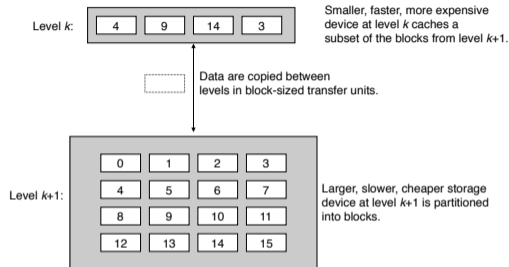
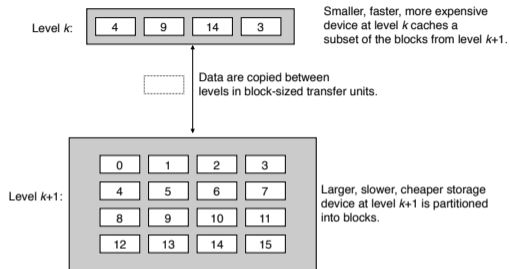


Figure: Intel 2020 Gulftown die shot. Image credit AnandTech

Figure: Cache stores a temporary copy from the slower main memory. Image credit CS:APP

# CPU / cache / DRAM main memory interactions



## When CPU loads (LD) from memory

- ▶ Cache read hit
- ▶ Cache read miss

## When CPU stores (ST) to memory

- ▶ Cache write hit
- ▶ Cache write miss

Figure: Cache stores a temporary copy from the slower main memory. Image credit CS:APP

# Table of contents

## Announcements

## Locality: How to create illusion of fast access to capacious data

- Spatial locality

- Temporal locality

## Caches: motivation

- Hardware caches supports software locality

- Software locality exploits hardware caches

## Cache placement policy (how to find data at address for read and write hit)

- Fully associative cache

- Direct-mapped cache

- Set-associative cache

## PA5: Simulating a cache and optimizing programs for caches

# Cache placement policy (how to find data at address for read and write hit)

## Several designs for caches

- ▶ Fully associative cache
- ▶ Direct-mapped cache
- ▶ N-way set-associative cache

## Cache design options use $m$ -bit memory addresses differently

- ▶  $t$ -bit tag
- ▶  $s$ -bit set index
- ▶  $b$ -bit block offset

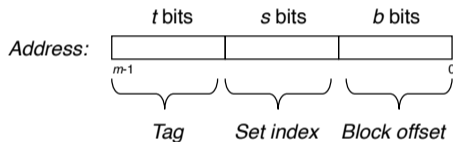
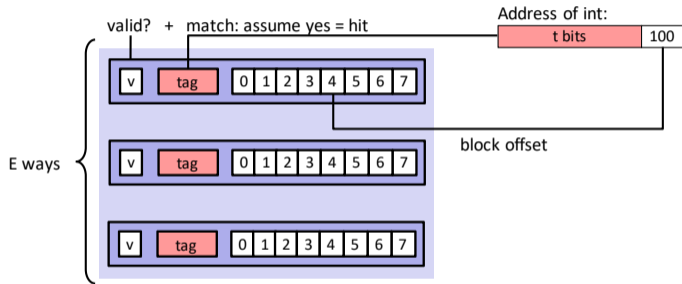


Figure: Memory addresses. Image credit CS:APP

# Fully associative cache



$m$ -bit memory address  
split into:

- ▶  $t$ -bit tag
- ▶  $b$ -bit block offset

Figure: Fully associative cache. Image credit CS:APP

# Fully associative cache

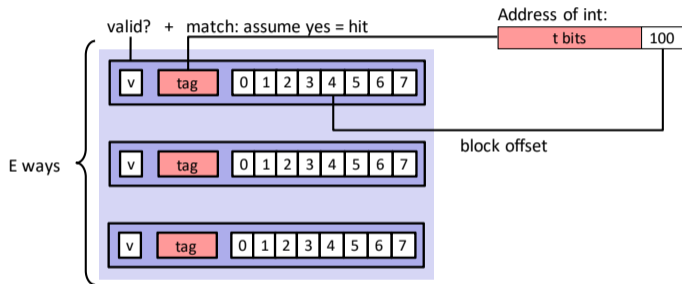


Figure: Fully associative cache. Image credit CS:APP

## $b$ -bit block offset

- ▶ here,  $b = 3$
- ▶ The number of bytes in a block is  $B = 2^b = 2^3 = 8$
- ▶ A block is the minimum number of bytes that can be cached
- ▶ Good for capturing spatial locality, short strides

# Fully associative cache

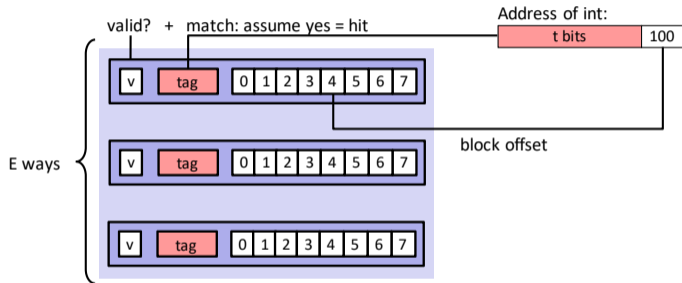


Figure: Fully associative cache. Image credit CS:APP

## $t$ -bit tag

- ▶ here,  
 $t = m - b = m - 3$
- ▶ When CPU wants to read from or write to memory, all  $t$ -bits in tag need to match for read/write hit.

# Fully associative cache

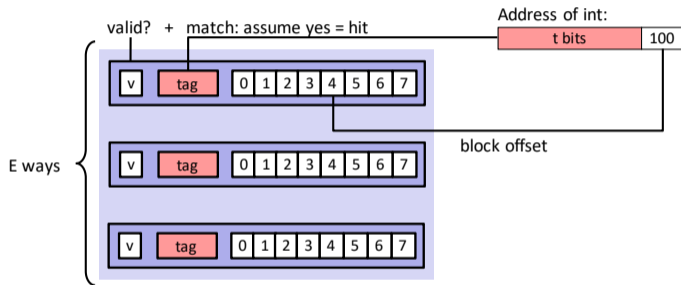


Figure: Fully associative cache. Image credit CS:APP

## Full associativity

- ▶ Blocks can go into any of  $E$  ways
- ▶ Here,  $E = 3$
- ▶ Good for capturing temporal locality: cache hits can happen even with intervening reads and writes to other tags.



# Fully associative cache

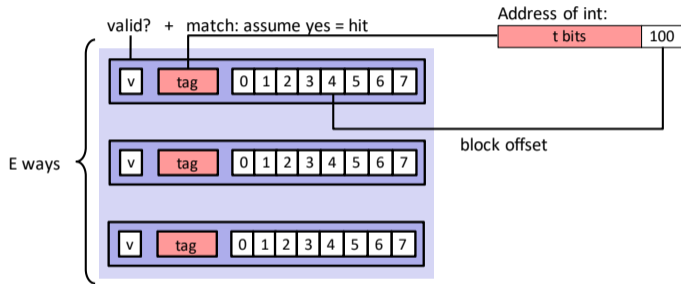


Figure: Fully associative cache. Image credit CS:APP

## Capacity of cache

- ▶ Total capacity of fully associative cache in bytes:  $C = EB = E * 2^b$
- ▶ Here,  $C = E * 2^b = 3 * 2^3 = 24$  bytes

# Fully associative cache

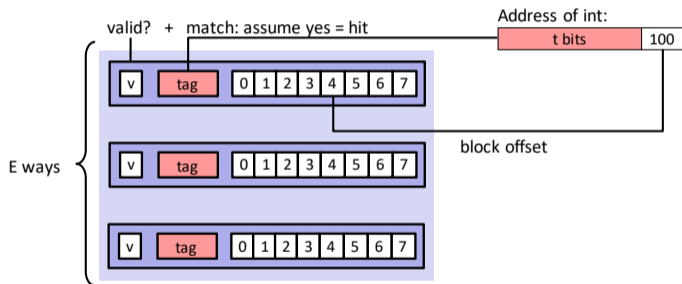


Figure: Fully associative cache. Image credit CS:APP

## Strengths

- ▶ Blocks can go into any of  $E$ -ways.
- ▶ Hit rate is only limited by total capacity.

## Weaknesses

- ▶ Searching across all valid tags is expensive.
- ▶ Figuring out which block to evict on read/write miss is also expensive.
- ▶ Requires a lot of

# Direct-mapped cache

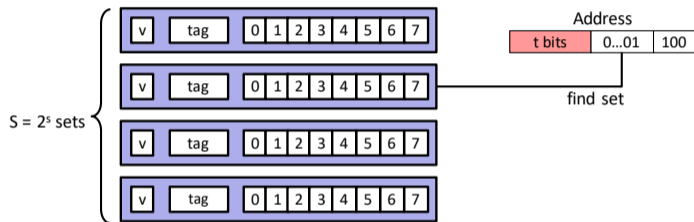


Figure: Direct-mapped cache. Image credit CS:APP

$m$ -bit memory address  
split into:

- ▶  $t$ -bit tag
- ▶  $s$ -bit set index
- ▶  $b$ -bit block offset

# Direct-mapped cache

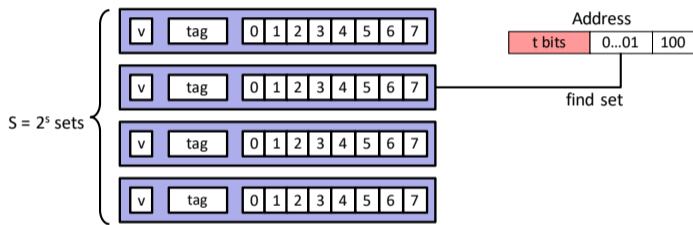


Figure: Direct-mapped cache. Image credit CS:APP

## $b$ -bit block offset

- ▶ here,  $b = 3$
- ▶ The number of bytes in a block is  $B = 2^b = 2^3 = 8$
- ▶ A block is the minimum number of bytes that can be cached
- ▶ Good for capturing spatial locality, short strides

# Direct-mapped cache

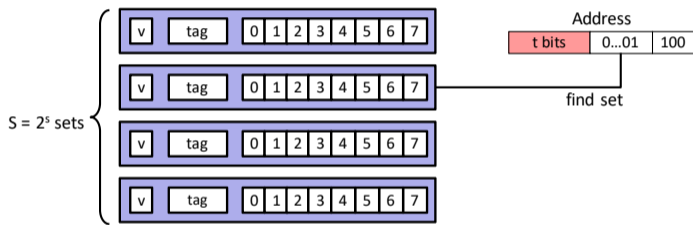


Figure: Direct-mapped cache. Image credit CS:APP

## s-bit set index

- ▶ here,  $s = 2$
- ▶ The number of sets in cache is  $S = 2^s = 2^2 = 4$
- ▶ A hash function that limits exactly where a block can go
- ▶ Good for further increasing ability to exploit spatial locality, short strides

# Direct-mapped cache

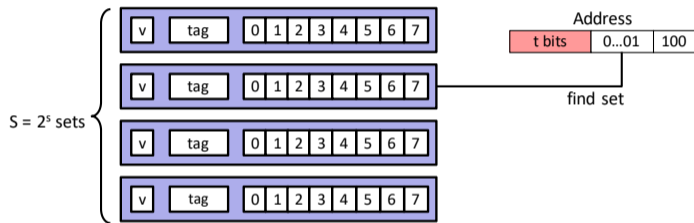


Figure: Direct-mapped cache. Image credit CS:APP

## $t$ -bit tag

- ▶ here,  
 $t = m - s - b = m - 2 - 3$
- ▶ When CPU wants to read from or write to memory, all  $t$ -bits in tag need to match for read/write hit.

# Direct-mapped cache

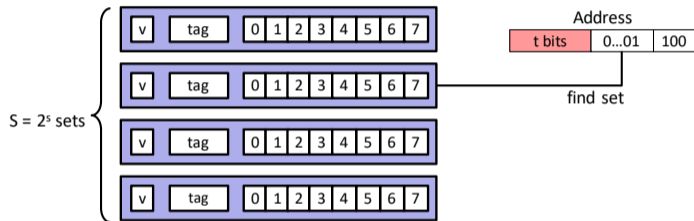


Figure: Direct-mapped cache. Image credit CS:APP

## Direct mapping

- ▶ In direct-mapped cache, blocks can go into only one of  $E = 1$  way

# Direct-mapped cache

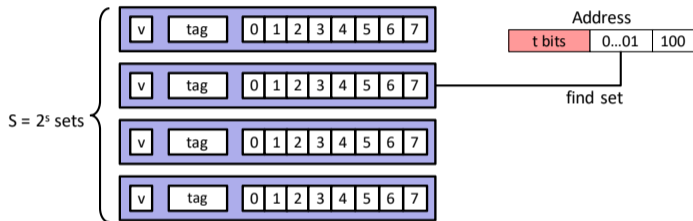


Figure: Direct-mapped cache. Image credit CS:APP

## Capacity of cache

- ▶ Total capacity of fully associative cache in bytes:  
 $C = SEB = 2^s * E * 2^b$
- ▶ Here,  $C = 2^s * E * 2^b = 2^2 * 1 * 2^3 = 32$  bytes



# Direct-mapped cache

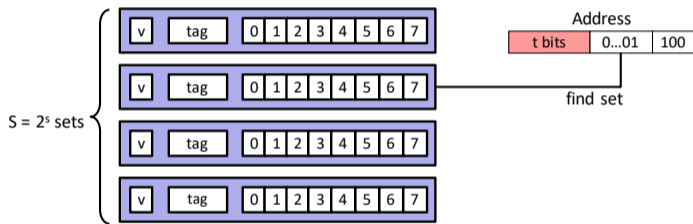


Figure: Direct-mapped cache. Image credit CS:APP

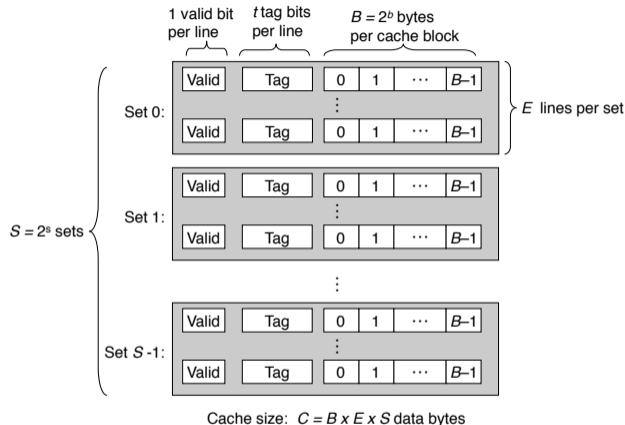
## Strengths

- ▶ Simple to implement.
- ▶ No need to search across tags.

## Weaknesses

- ▶ Can lead to surprising thrashing of cache with unfortunate access patterns.
- ▶ Unexpected conflict misses independent of cache capacity.

# E-way set-associative cache

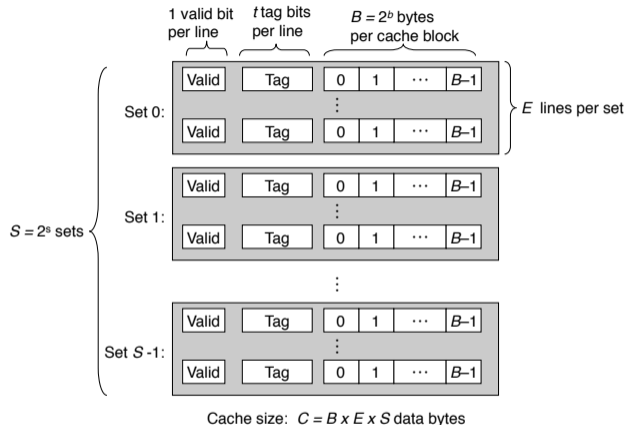


## Strengths

- ▶ Blocks can go into any of  $E$ -ways, increases ability to support temporal locality, thereby increasing hit rate.
- ▶ Only need to search across  $E$  tags. Avoids costly searching across all valid tags.
- ▶ Avoids conflict misses due to unfortunate access patterns.

Figure: Direct-mapped cache. Image credit CS:APP

# E-way set-associative cache

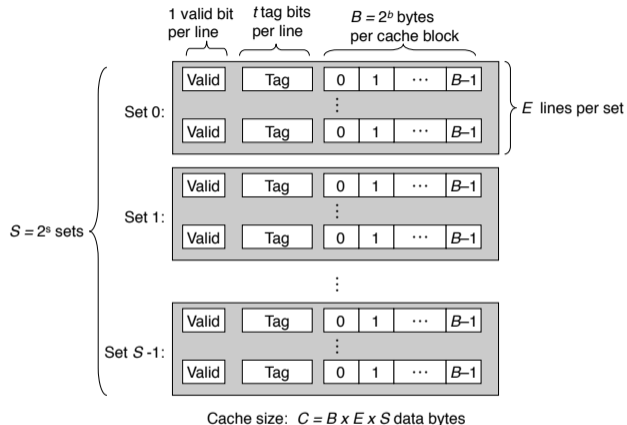


Used in practice in, e.g., a recent Intel Core i7:

- ▶  $C = 32\text{KB}$  L1 data cache per core
- ▶  $S = 64 = 2^6$  sets/cache ( $s = 6$  bits)
- ▶  $E = 8 = 2^3$  ways/set
- ▶  $B = 64 = 2^6$  bytes/block ( $b = 6$  bits)
- ▶  $C = S * E * B$
- ▶ Assuming memory addresses are  $m = 48$ , then tag size  
 $t = m - s - b = 48 - 6 - 6 = 36$  bits.

Figure: Direct-mapped cache. Image credit CS:APP

# E-way set-associative cache



Let's see textbook slides for a simulation

Figure: Direct-mapped cache. Image credit CS:APP

# Table of contents

## Announcements

## Locality: How to create illusion of fast access to capacious data

- Spatial locality

- Temporal locality

## Caches: motivation

- Hardware caches supports software locality

- Software locality exploits hardware caches

## Cache placement policy (how to find data at address for read and write hit)

- Fully associative cache

- Direct-mapped cache

- Set-associative cache

## PA5: Simulating a cache and optimizing programs for caches

# PA5: Simulating a cache and optimizing programs for caches

## Write a cache simulator

1. fullyAssociative
2. directMapped
3. setAssociative

## Optimize some code for better cache performance

1. cacheBlocking
2. cacheOblivious

# PA5: Simulating a cache and optimizing programs for caches

## A tour of files in the package

- ▶ trace files
- ▶ csim-ref