Memory Hierarchy

Bojian Zheng

CSCD70 Spring 2018

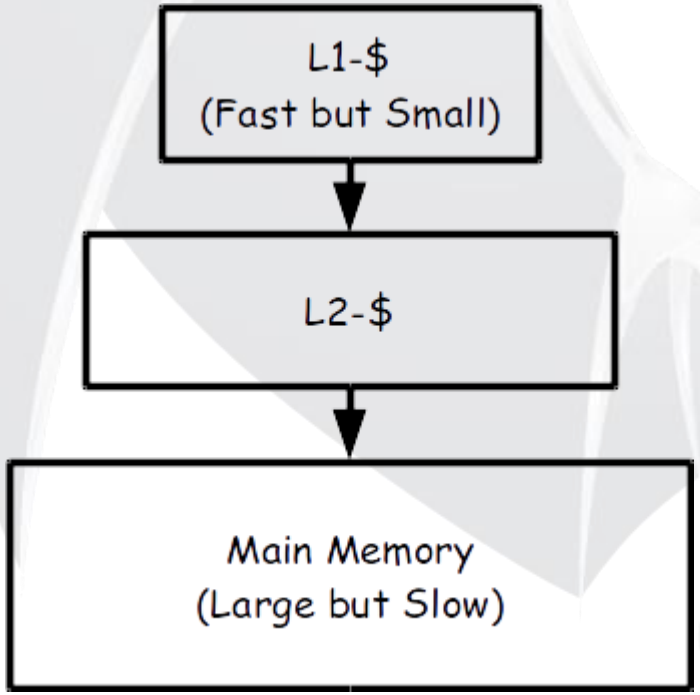
bojian@cs.toronto.edu

Memory Hierarchy

- From programmer's point of view, memory
  - has infinite capacity (i.e. can store infinite amount of data)
  - has zero access time (latency)
- But those two requirements **contradict** with each other.
  - Large memory usually has high access latency.
  - Fast memory cannot go beyond certain capacity limit.
- Therefore, we want to have **multiple levels of storage** and ensure most of the data the processor needs is kept in the fastest level.

Memory Hierarchy

What in reality



What we see (Ideally)

Locality

- **Temporal** Locality: If an address gets accessed, then it is very likely that the exact same address will be accessed **once again** in the **near future**.

```
unsigned a = 10;
...
// 'a' will hopefully be used again soon
```

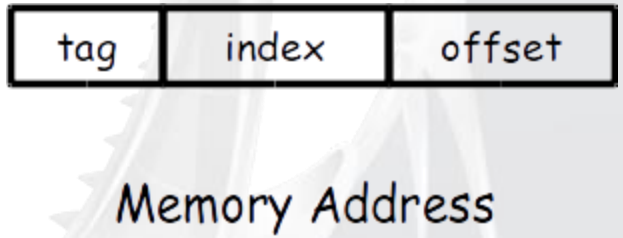
Locality

- **Spatial** Locality: If an address gets accessed, then it is very likely that **nearby** addresses will be accessed in the **near future**.

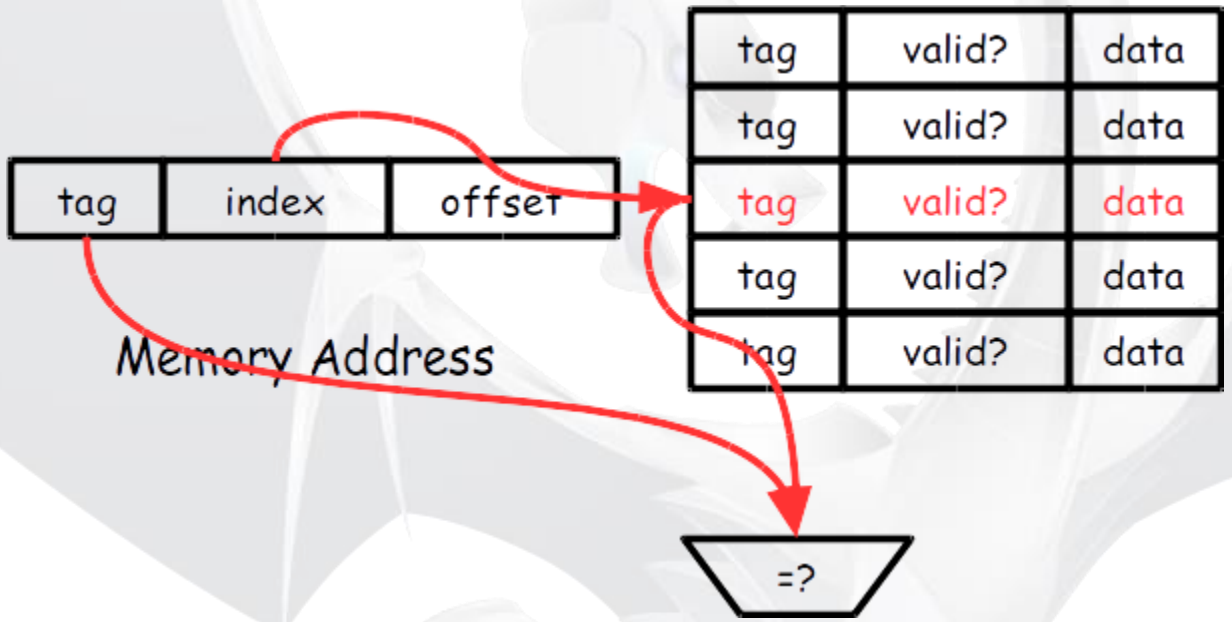
```
unsigned A[10];

A[5] = 10;
...
// 'A[4]' or 'A[6]' will
hopefully be used soon
```

Cache Basics

- Memory is divided into **fixed size blocks**. Each block maps to one location in the cache (called **cache block** or **cache line**), determined by the index bites.



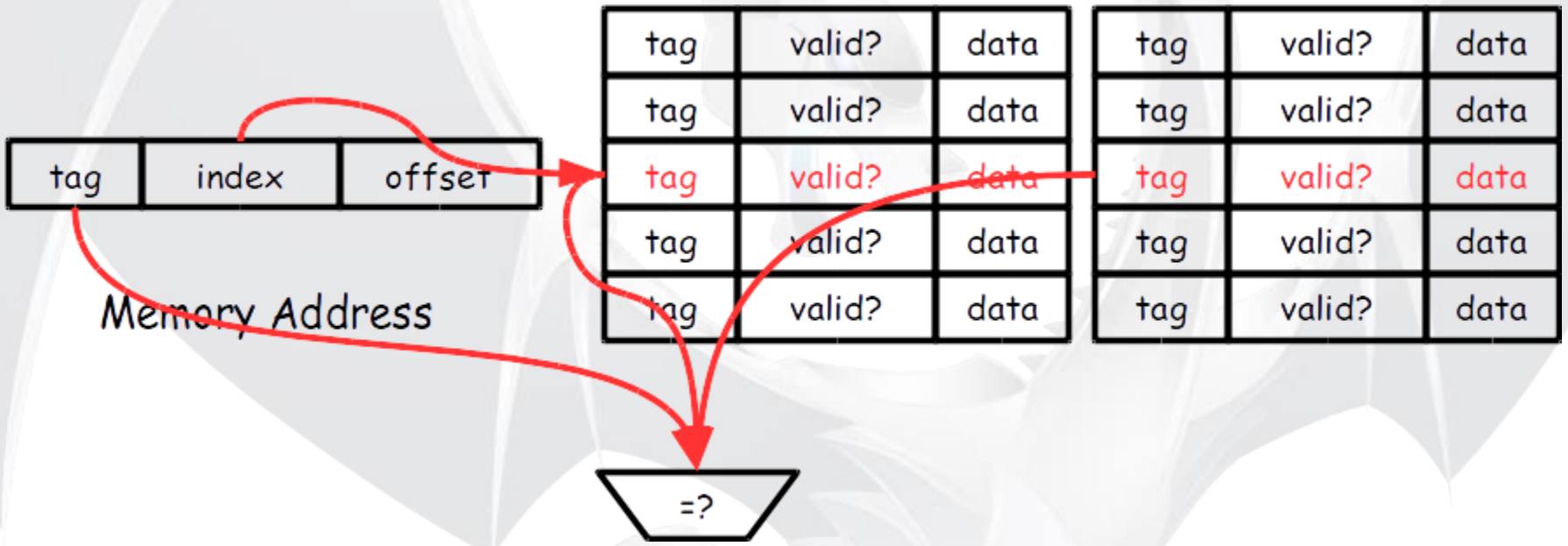
Direct-Mapped Cache



Direct-Mapped Cache: Problem

- Suppose that two variables `A` (address `0'b10000000`) and `B` (address `0'b11000000`) map to the same cache line. Interleaving accesses to them will lead to **0** hit rate (i.e. `A->B->A->B->`...).
- Those are called **conflict** misses (more later).

Set-Associative Cache



Set-Associative Cache: Problem

- More expansive **tag comparison**.
- More complicated design – lots of things to consider:
  - Where should we insert the new incoming cache block?
  - What happens when a cache hit occurs? How should we adjust the priorities?
  - What happens when a cache miss occurs, and the cache set has been fully occupied (**Replacement Policy**)?

Replacement Policy

- Which one to evict under the condition that the cache set is full?
  - Least-Recently-Used?
  - Random?
  - Pseudo-Least-Recently-Used?
- **Belady's** (a.k.a. Optimal) Replacement Policy
  - Evict block that **will be** reused furthest in the future.
  - **Impossible** to implement in hardware … why?
  - But is still **useful** … why?

Cache Misses

- There are **3** types of cache misses:
  - **Cold Misses**: happens whenever we reference one variable (memory location) for the first time. Such misses are **unavoidable (???)**.
  - **Capacity Misses**: happens because our cache is too small.
    - Misses that happen even under fully-associative cache and optimal replacement policy.
    - Cache size is smaller than working set size.
  - **Conflict Misses**: happens because we do not have enough associativity.

Handling Writes

- **Write-Back** vs. **Write-Through**
  - Write-Back: Write modified data to memory **when the cache block is evicted**.
    - (+) Can effectively combine multiple writes to the same block.
    - (-) Needs an extra bit indicating **dirty** or clean.
  - Write-Through: Write modified data to memory **whenever write occurs**.
    - (+) Simple and makes it easy when arguing about **consistency**.
    - (-) Cannot combine writes and is more bandwidth intensive.

Prefetching

- Upon one cache miss, try to **predict** what the next cache miss will be.
- For instance, miss on `A[0]` ⇒ prefetch `A[1]` into the cache.
- Good for memory access patterns that are highly **predictable**, e.g.
  - Instruction Memory
  - Array Accesses (Uniform Stride)
- **Risk: Cache Pollution**
- **Goal: Timeliness, Coverage, Accuracy**

Questions?

- Cache Basics
  - Cache
  - Locality
  - Set-Associativity
    - Replacement Policy
- Cache Misses
  - Cold
  - Capacity
  - Conflict
- Handling Writes
  - Write-Back
  - Write-Through
- Prefetching
  - Risk
  - Goal

Preview: Cache & Compiler

- Ok … So how this is related to compiler?

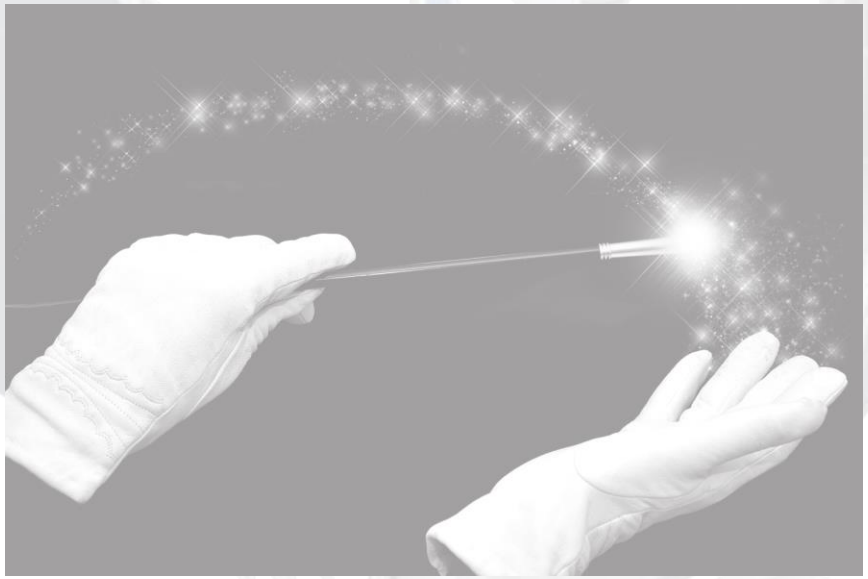
```
unsigned A[20][10];

for (unsigned i = 0; i < 10; ++i)
        for (unsigned j = 0; j < 20; ++j)
                A[j][i] = j * 10 + i; // Assume A is row-major.
```

- This is not very nice … because cache has been utilized badly : (

Preview: Cache & Compiler

```
for (i ∈ [0, 10))
    for (j ∈ [0, 20))
        A[j][i] = j * 10 + i;
```

```
for (j ∈ [0, 20))
    for (i ∈ [0, 10))
        A[j][i] = j * 10 + i;
```



Preview: Cache & Compiler

- Consider another example:

```
unsigned A[100][100];

for (unsigned i = 0; i < 100; ++i)
        for (unsigned j = 0; j < 100; ++j)
                sum += A[i][j]
```

Preview: Cache & Compiler

- Apply **prefetching**:

```
unsigned A[100][100];

for (unsigned i = 0; i < 100; ++i)
        for (unsigned j = 0; j < 100; j += $_BLOCK_SIZE)
                for (unsigned jj = j; jj < j + $_BLOCK_SIZE; ++jj)
                        prefetch(&A[i][j] + $_BLOCK_SIZE)
                        sum += A[i][jj]
```

Preview: Design Tradeoff

- Consider the following code:

```
unsigned A[20][10];

for (unsigned i = 0; i < 10; ++i)
        for (unsigned j = 0; j < 19; ++j)
                A[j][i] = A[j+1][i];
```

Preview: Design Tradeoff

Loop Parallelization

```
for (i ∈ [0, 10))
    for (j ∈ [0, 19))
        A[j][i] = A[j+1][i];
```

Cache Locality

```
for (j ∈ [0, 19))
    for (i ∈ [0, 10))
        A[j][i] = A[j+1][i];
```