Lecture-17 (Hardware Prefetching)
CS422-Spring 2018

Biswa@CSE-IITK

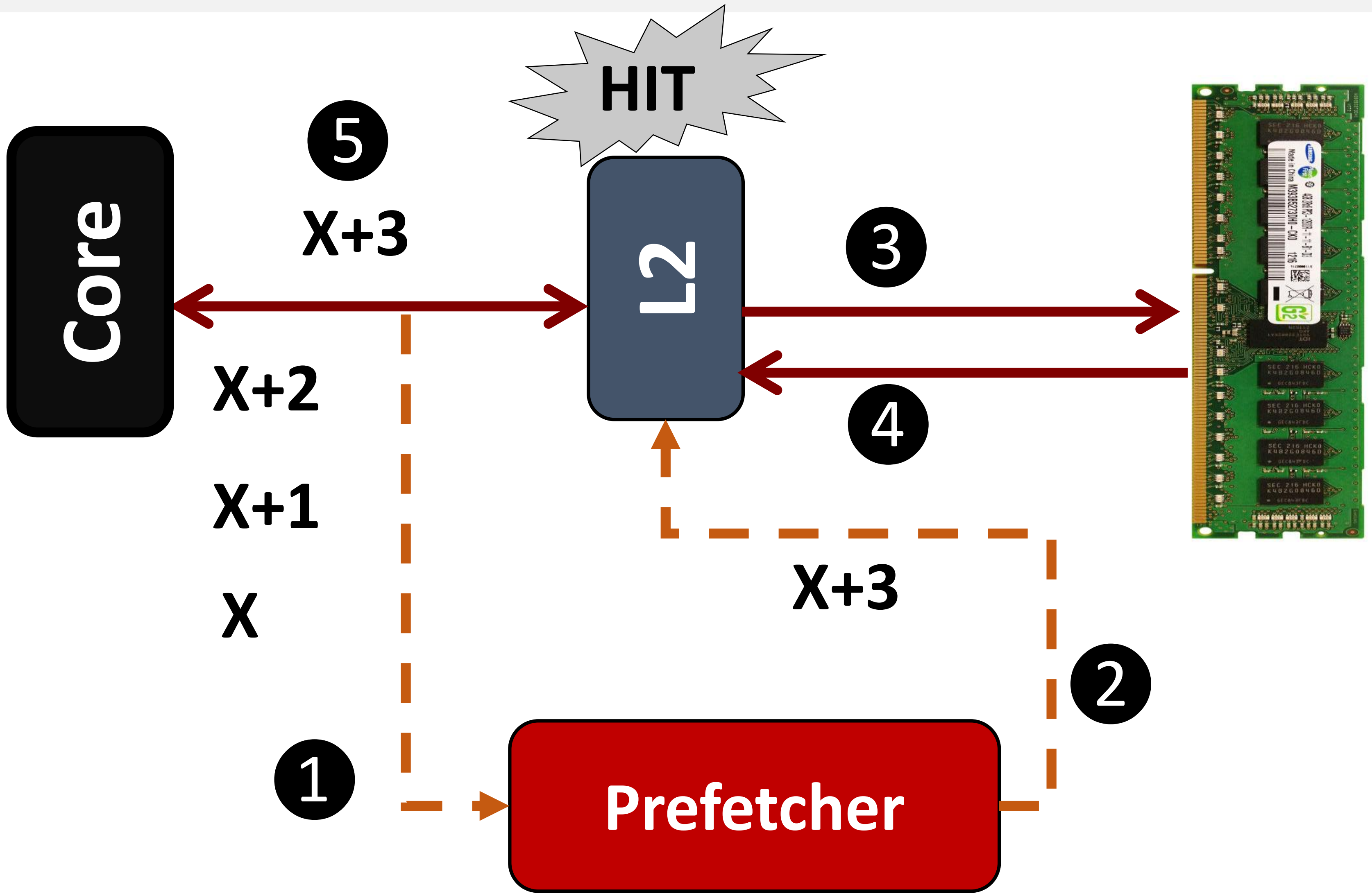
Hardware Prefetching

What?
Latency-hiding technique - Fetches data before the core demands.

Why?
Off-chip DRAM latency has grown up to 400 to 800 cycles.

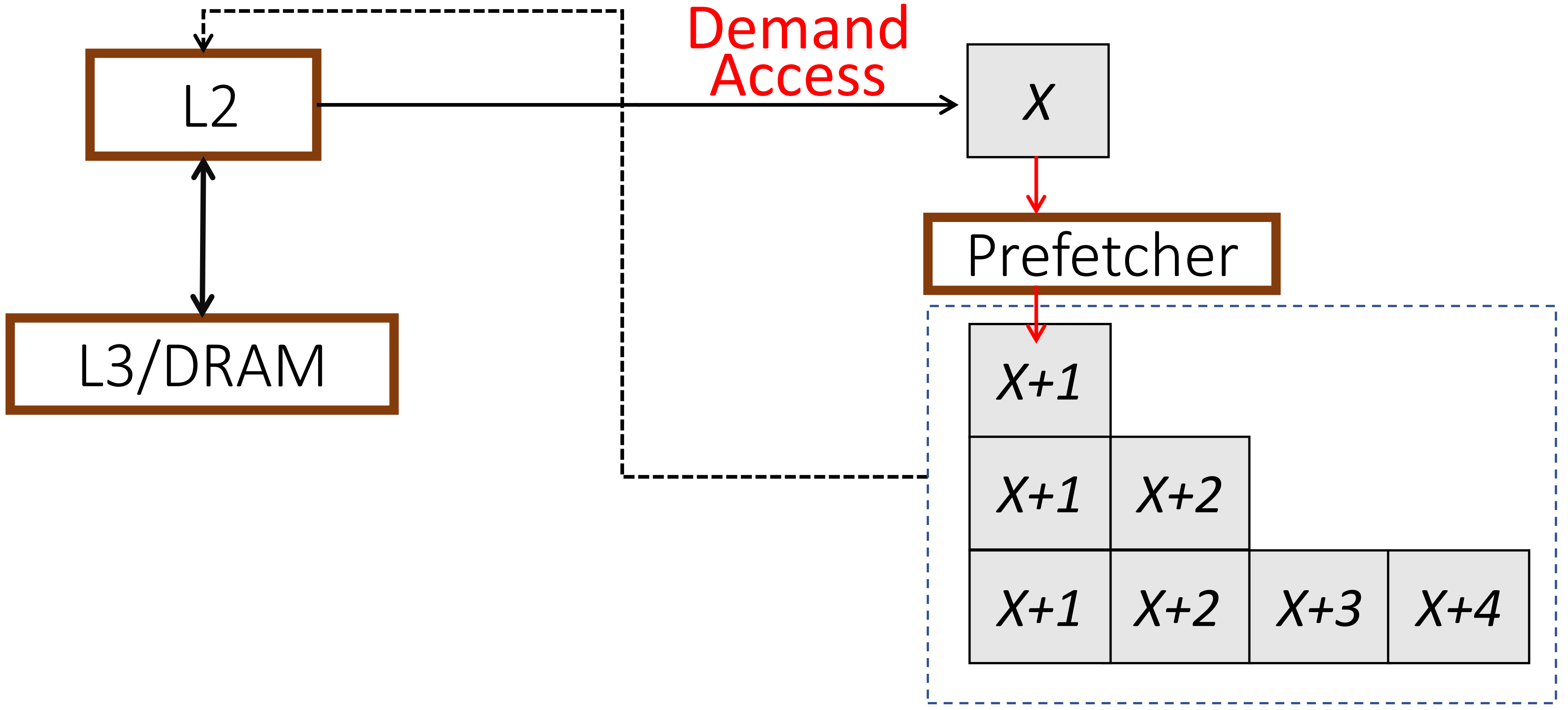
How?
By observing/predicting the demand access (LOAD/STORE) patterns.

Prefetch Engine



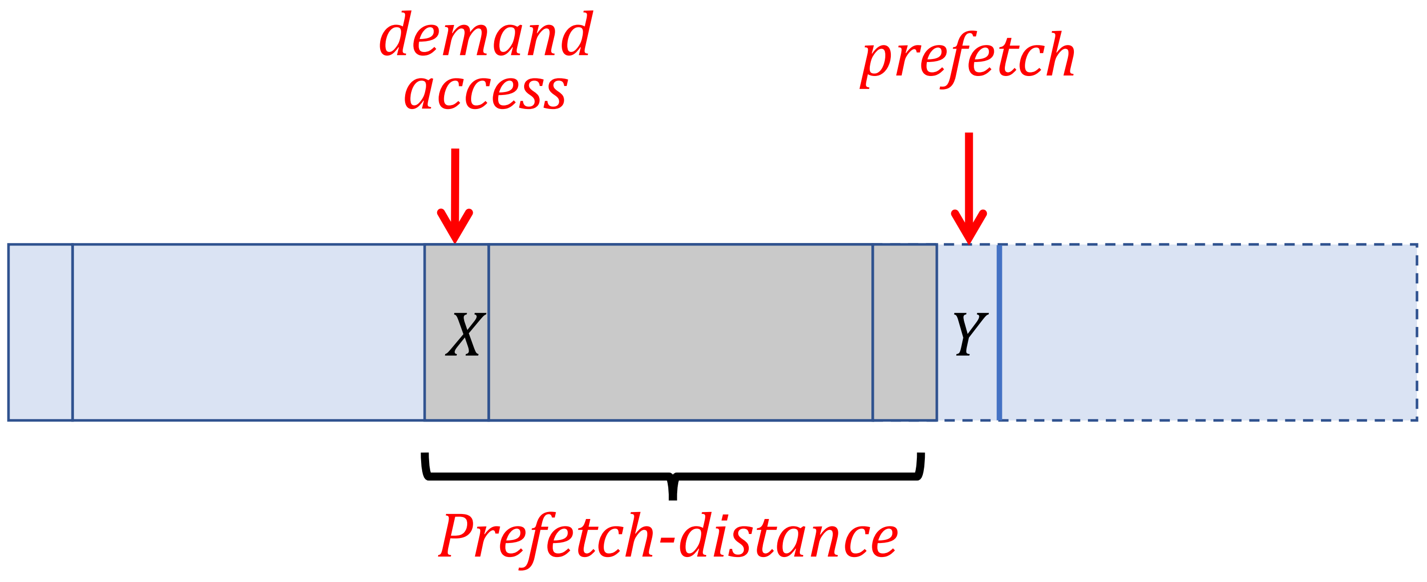
Prefetch Degree

Prefetch Degree: Number of prefetch requests to issue at a given time.



Prefetch Distance

Prefetch Distance: How far ahead of the demand access stream are the prefetch requests issued?



$Y = X + 4$

$Y = X + 8$

$Y = X + 16$

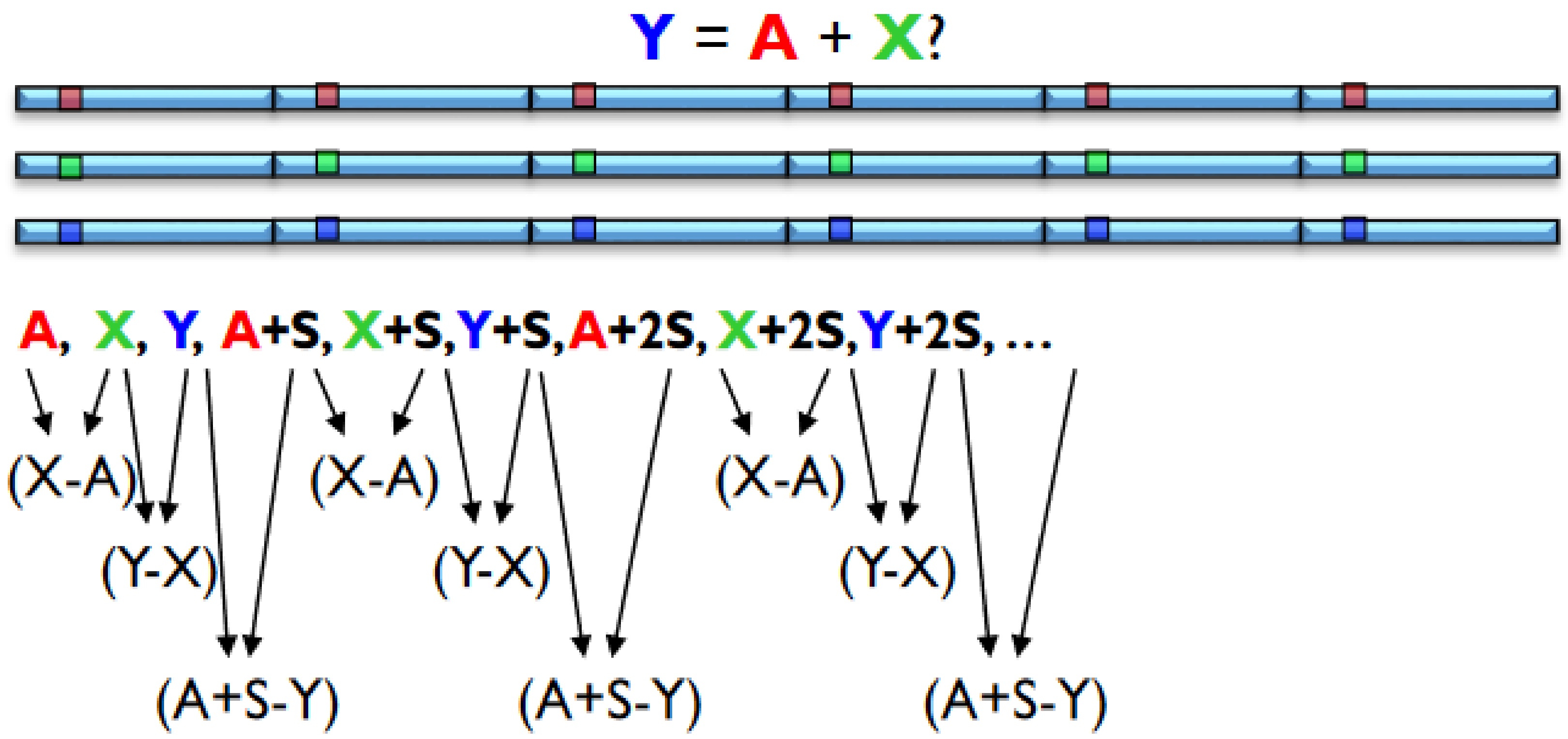
Next-line Prefetcher

Next Line: Miss to cache block X , prefetch X+1. Degree=1, Distance=1

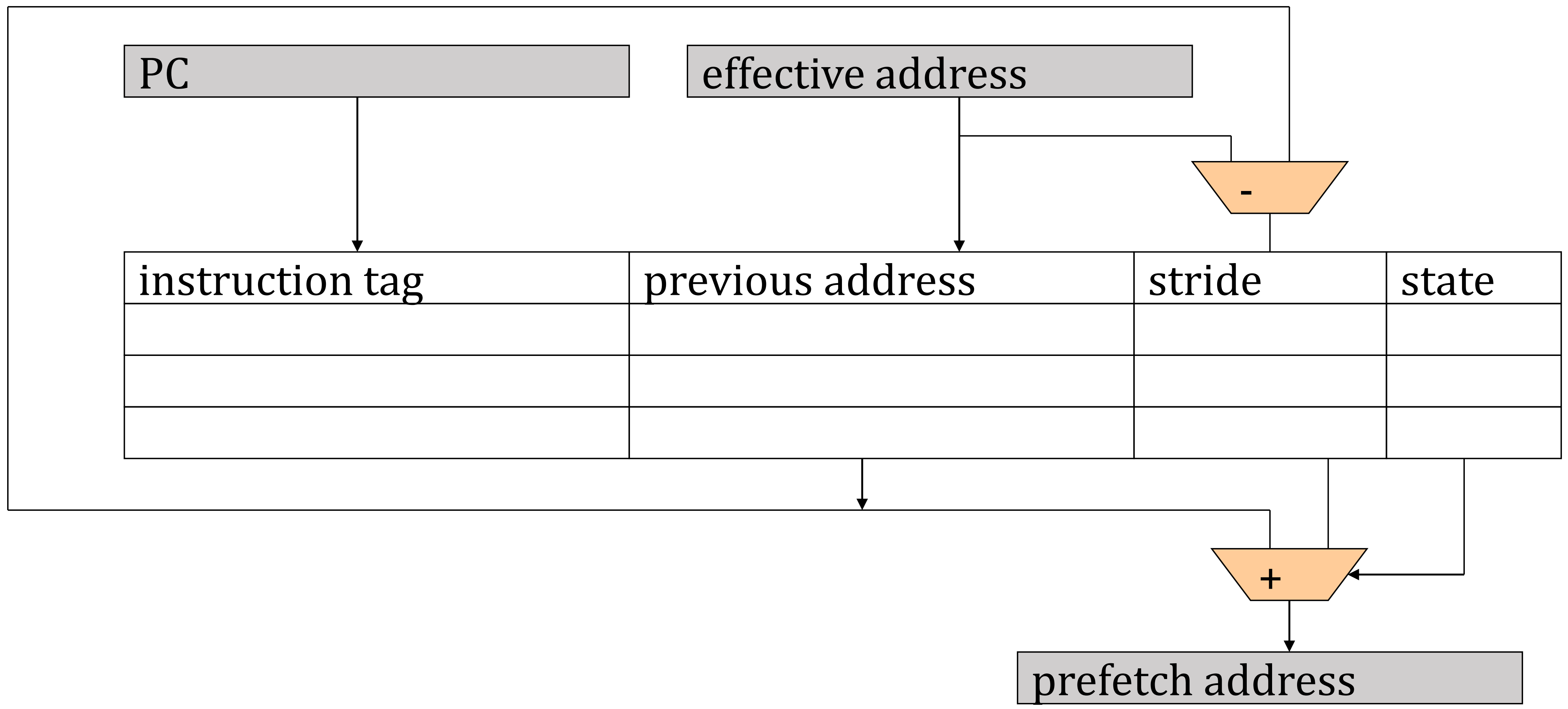
Works well for L1 Icache and L1 Dcache.

What About This?

```
Load R1 = [R2]
Load R3 = [R4]
Add R5, R1, R3
Store [R6] = R5
```



Stride Prefetching



Quantifying Prefetchers

$$\text{PrefetchAccuracy}(i) = \frac{\text{Prefetch}_{\text{hits}}(i)}{\text{Prefetch}_{\text{issued}}(i)}$$

Prefetched Block in the Cache.

$$\text{Lateness}(i) = \frac{\text{Prefetch}_{\text{late}}(i)}{\text{Prefetch}_{\text{hits}}(i)}$$

Prefetched Block Still on its way

$$\text{Pollution}(i) = \frac{\text{LLC Poll}(i)}{\text{Demand}_{\text{misses}}(i)}$$

Prefetched Block evicted a demand block that will be reused

$$\text{Coverage}(i) = \frac{\text{Prefetch Hits}(i)}{\text{Prefetch Hits}(i) + \text{Demand}_{\text{misses}}(i)}$$

Fraction of misses avoided

Design Issues: Unified vs Split

- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  - -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - for the last reason above
- Second and higher levels are almost always unified

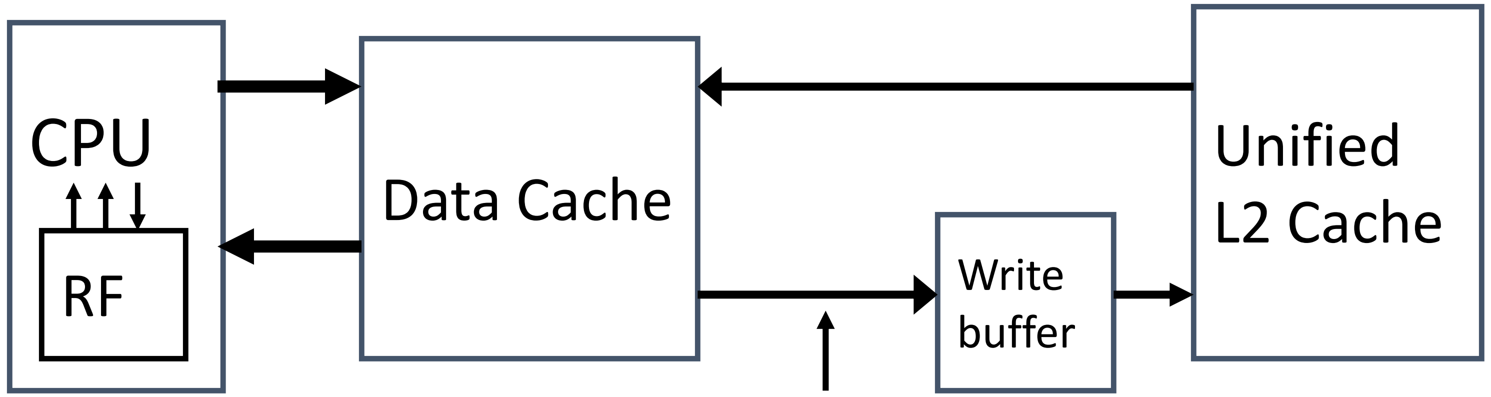
Reads are not Writes

- If a write enters the cache, what happens if
  - There is a cache miss
    - Does the cache need to bring in the cache line?
  - There is a cache hit
    - Does the cache need to write back to memory?

Write Policies

- Cache hit:
  - ***write through***: write both cache & memory
    - Generally higher traffic but simpler pipeline & cache design
  - ***write back***: write cache only, memory is written only when the entry is evicted
    - A dirty bit per line further reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store
- Cache miss:
  - ***no write allocate***: only write to main memory
  - ***write allocate*** (aka fetch on write): fetch into cache
- Common combinations:
  - write through and no write allocate
  - write back with write allocate

Write Buffers



Evicted dirty lines for writeback cache
OR
All writes in writethrough cache

Processor is not stalled on writes, and read misses can go ahead of write to main memory

Problem: Write buffer may hold updated value of location needed by a read miss

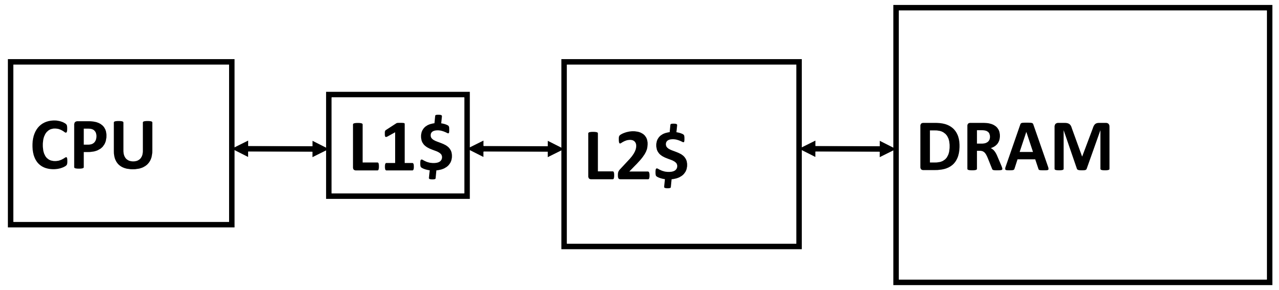
Simple solution: on a read miss, wait for the write buffer to go empty

Faster solution: Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

Multi-level Caches

Problem: A memory cannot be large and fast

Solution: Increasing sizes of cache at each level



Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions