PennState

CMPSC 311 - Introduction to Systems Programming

Caching

Professors:

Suman Saha

(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

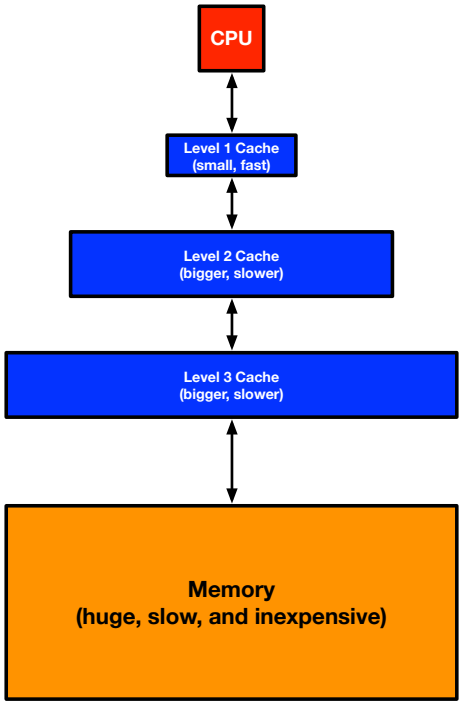
Caches

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- Big Idea: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Processor Caches

PennState

- Most modern computers have multiple layers of caches to manage data passing into and out of the processors
  - L1 – very fast and small, processor adjacent
  - L2 – a bit slower but often much larger
  - L3 –larger still, maybe off chip
    - May be shared amongst processors in multi-core system
  - Memory – slowest, least expensive

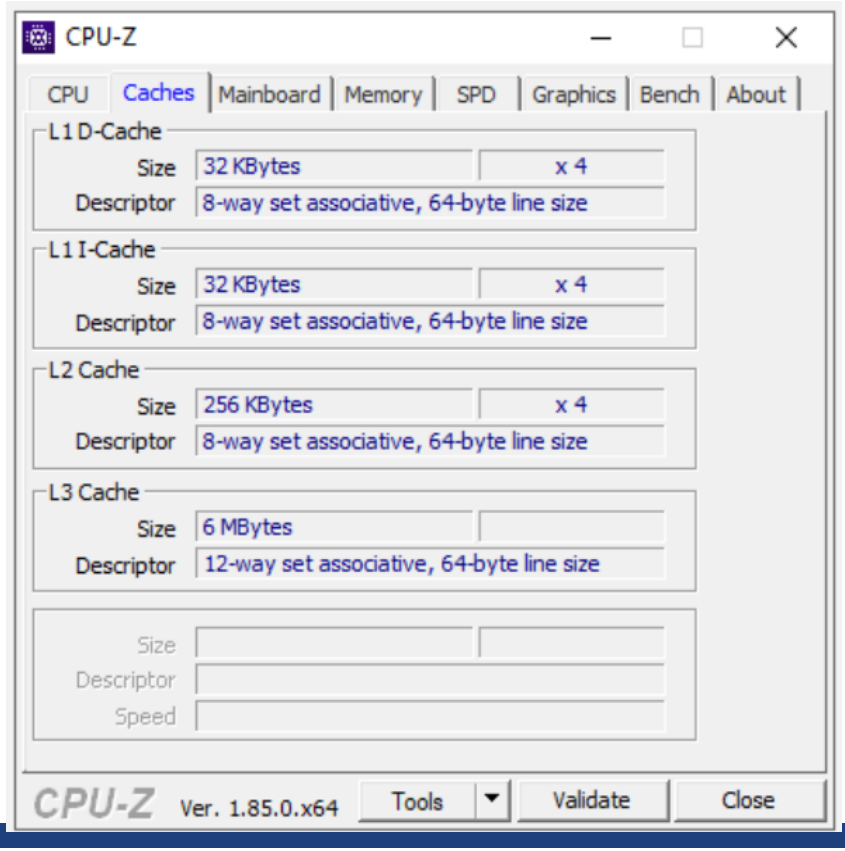


- AMD's Ryzen 5 5600X has a 384KB L1 cache and a 3MB L2 cache (plus a 32MB L3 cache).
- The L1 memory cache is typically 100 times faster than your RAM, while the L2 cache is around 25 times faster.
- Instruction caches are different from data caches

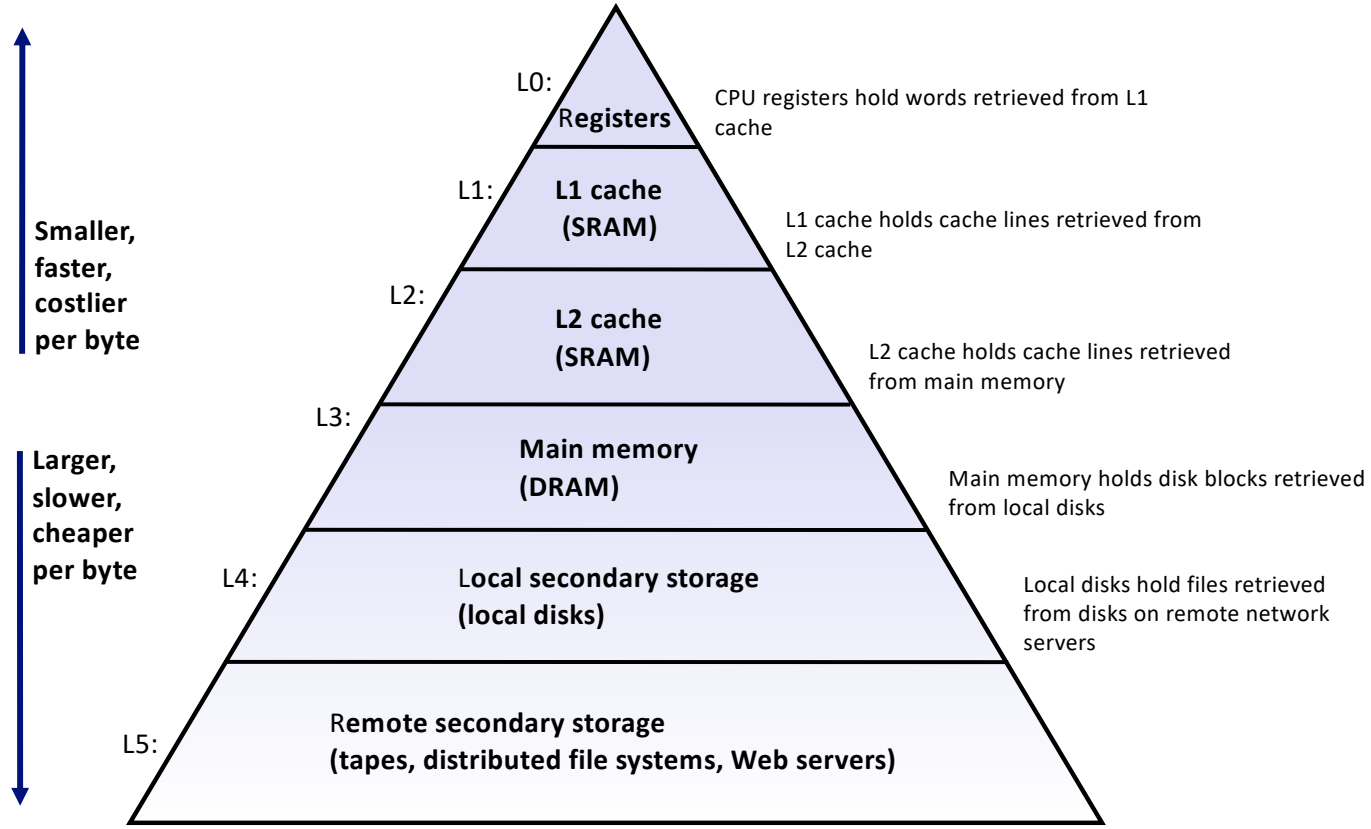
Example

PennState

- This image shows the CPU memory cache levels for an Intel Core i5-3570K CPU: (https://www.makeuseof.com/tag/what-is-cpu-cache/)



Reminder: Memory Hierarchy

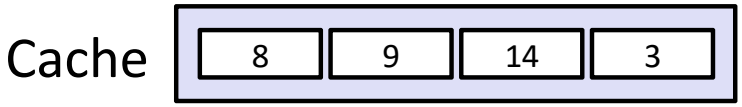


Locality

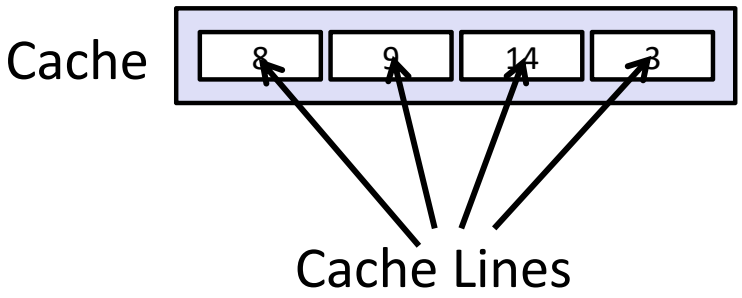
PennState

- Caches exploit locality to improve performance, of which there are two types:
  - Spatial locality: data to be accessed tend to be close to data you already accessed
  - Temporal (time) locality: data that is accessed is likely to be accessed again soon
- This leads to two cache design strategies
  - Spatial: cache items in blocks larger than that accessed
  - Temporal: keep stuff used recently around longer

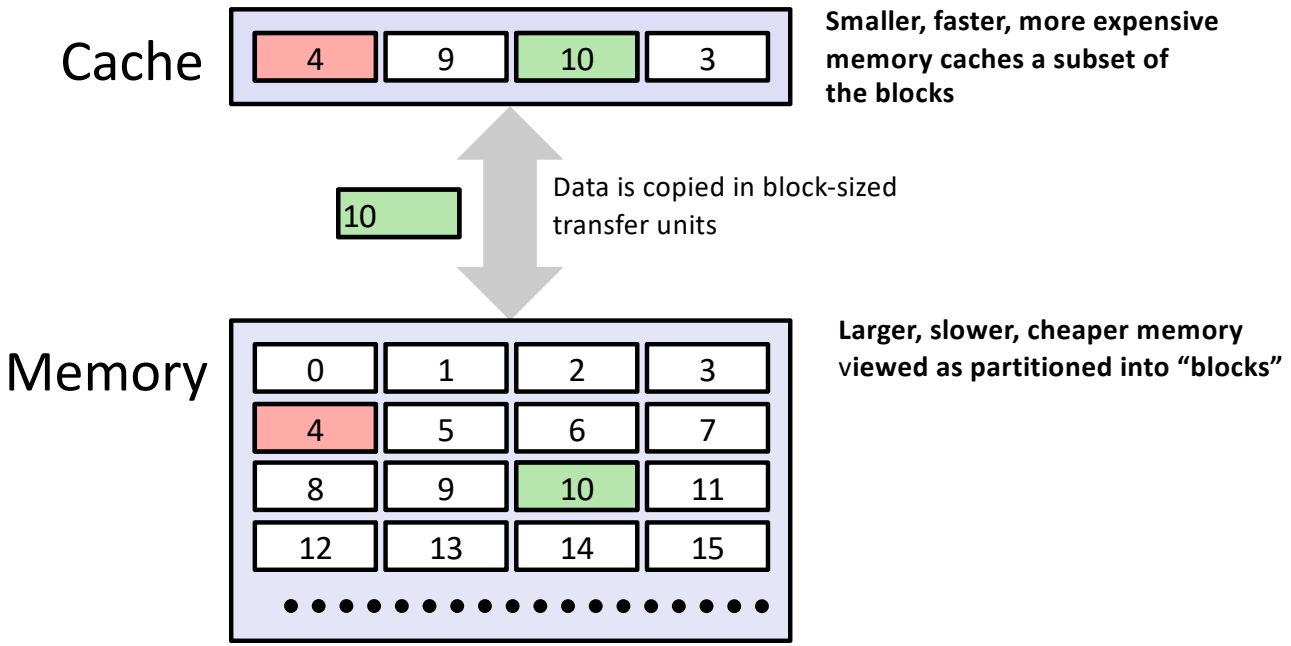
General Cache Concepts



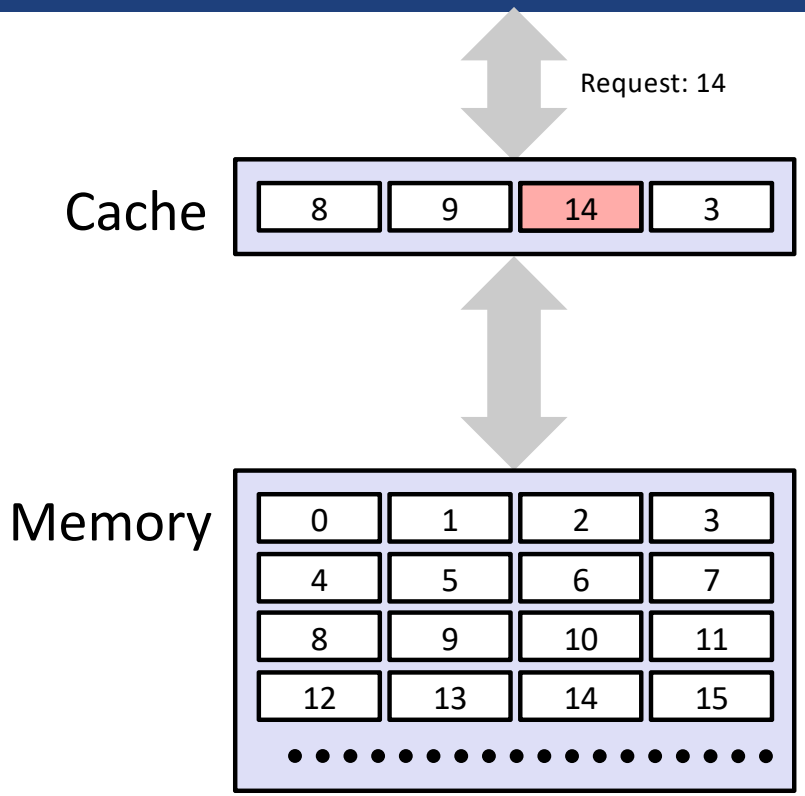
General Cache Concepts



General Cache Concepts



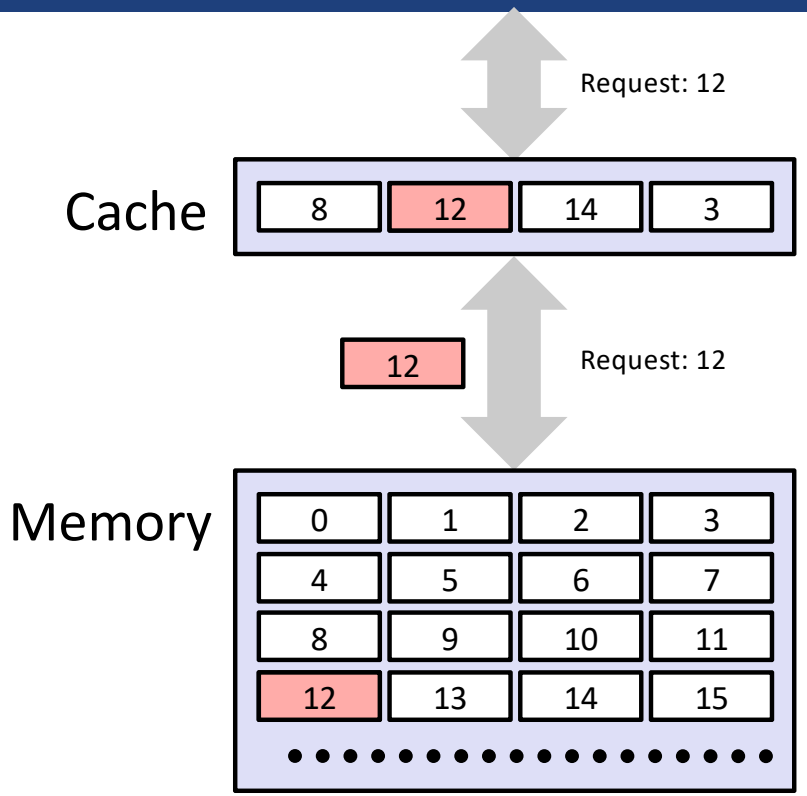
Cache Hit



Data in block b is needed

Block b is in cache:
Hit!

Cache Miss



Data in block b is needed

Block b is not in cache:
Miss!

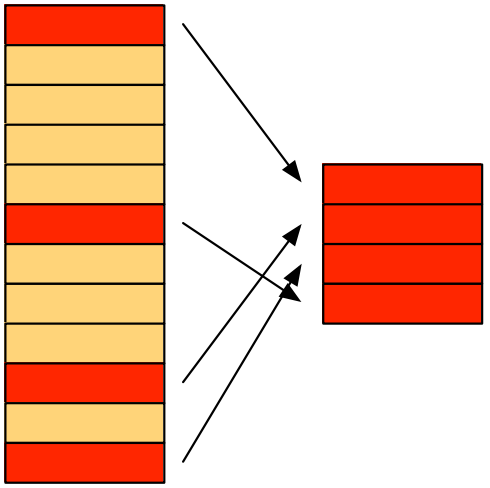
Block b is fetched from *memory*

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (called the victim)

Placement Policy

PennState

- Q: When a new block comes in, where in the cache can you keep it?
- A: Depends on the placement policy
  - Anywhere (fully associative)
    - Why not do this all the time?
  - Exactly one cache line (direct-mapped)
    - Commonly, block i is mapped to cache line (i mod t) where t is the total number of lines
  - One of n cache lines (n-way set-associative)

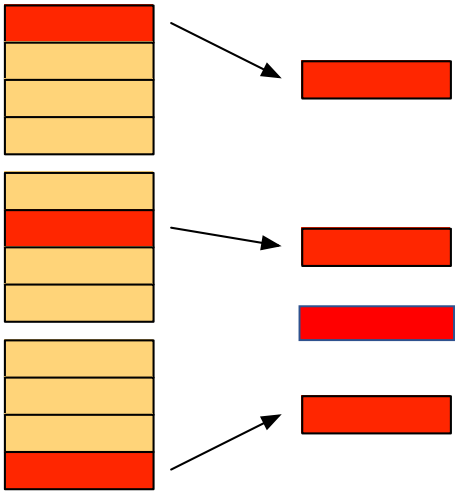


Fully Associative

Placement Policy

PennState

- Q: When a new block comes in, where in the cache can you keep it?
- A: Depends on the placement policy
  - Anywhere (fully associative)
    - Why not do this all the time?
  - Exactly one cache line (direct-mapped)
    - Commonly, block i is mapped to cache line (i mod t) where t is the total number of lines
  - One of n cache lines (n-way set-associative)

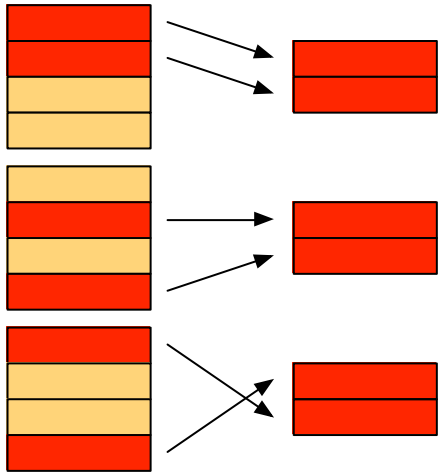


Direct Mapped

Placement Policy

PennState

- Q: When a new block comes in, where in the cache can you keep it?
- A: Depends on the placement policy
  - Anywhere (fully associative)
    - Why not do this all the time?
  - Exactly one cache line (direct-mapped)
    - Commonly, block i is mapped to cache line (i mod t) where t is the total number of lines
  - One of n cache lines (n-way set-associative)



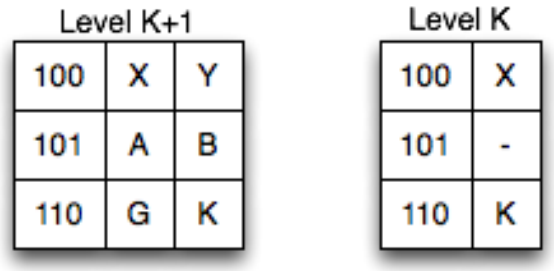
N-Way Set Associated

Types of Cache Misses

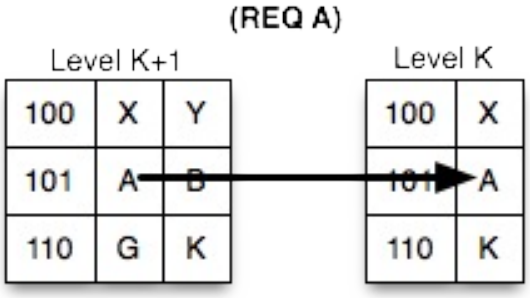
PennState

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
- Conflict miss (set-associative and direct mapping only)
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, … would miss every time.

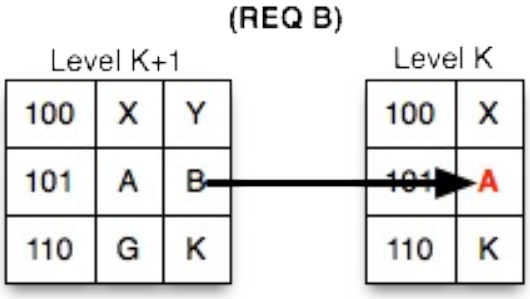
Conflict Miss



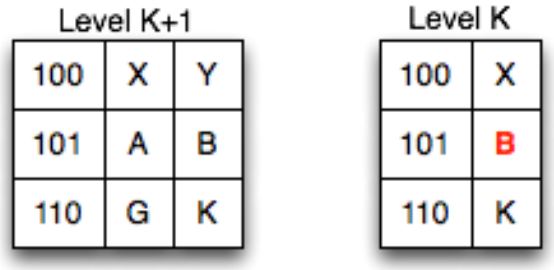
Conflict Miss



Conflict Miss



Conflict Miss

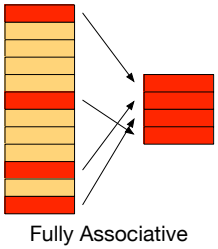


Note: Accessing A, B, A, B, … would miss every time, when other block in Level K can be replace to improve the hit ratio.

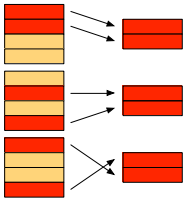
Cache replacement policy

PennState

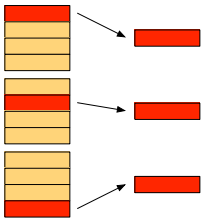
- When your cache is full and you acquire a new value, you must evict a previously stored value
  - Performance of cache is determined by how smart you are in evicting values, known as a cache eviction policy
  - Popular policies
    - Least recently used (LRU) - eject the value that has been in the cache the longest without being accessed
    - Least frequently used (LFU) - eject the value that accessed the least number of times
    - First in-first out (FIFO) - eject the same order they come in
  - Policy efficiency is measured by the hit ratio (how often is something asked for and found) and measured costs
    - Determined by working set (workload)



Fully Associative



N-Way Set Associated



Direct Mapped

Cache performance

PennState

- A cache hit is when the referenced information is served out of the cache
- A cache miss occurs referenced information cannot be served out of the cache
- The *hit ratio* is the:

$$hit\ ratio = \frac{\#\ cache\ hits}{\#\ total\ accesses}$$

- The efficiency of a cache is almost entirely determined by the hit ratio.

Cache performance

PennState

- The *average memory access time* can be calculated:

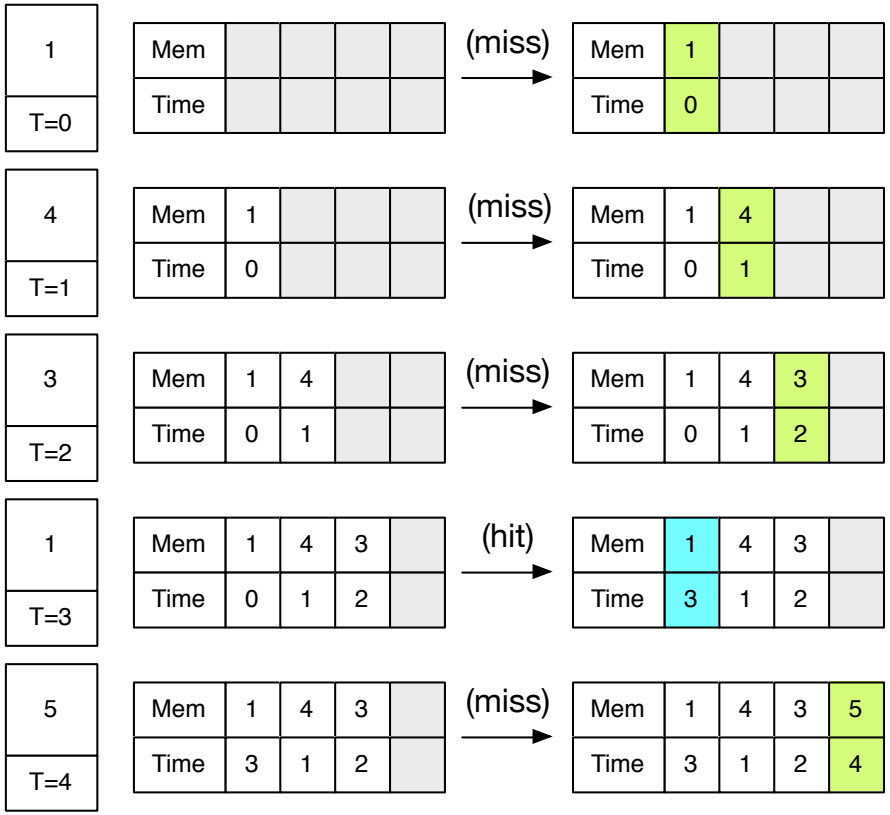
*average memory latency = hit time + miss rate * miss penalty*

- Where
  - hit time is the time it takes to read data from cache
  - miss penalty is the cost to serve out of main memory
  - miss ratio is the probability of a cache access resulting in a miss, i.e., *1 – hit-ratio*
- E.g., for a hit time of 25 usec and, penalty of 250 usec, and hit ratio of 80%:

*25 usec + 0.2 * 250 usec = 25 usec + 50 usec = 75 usec*

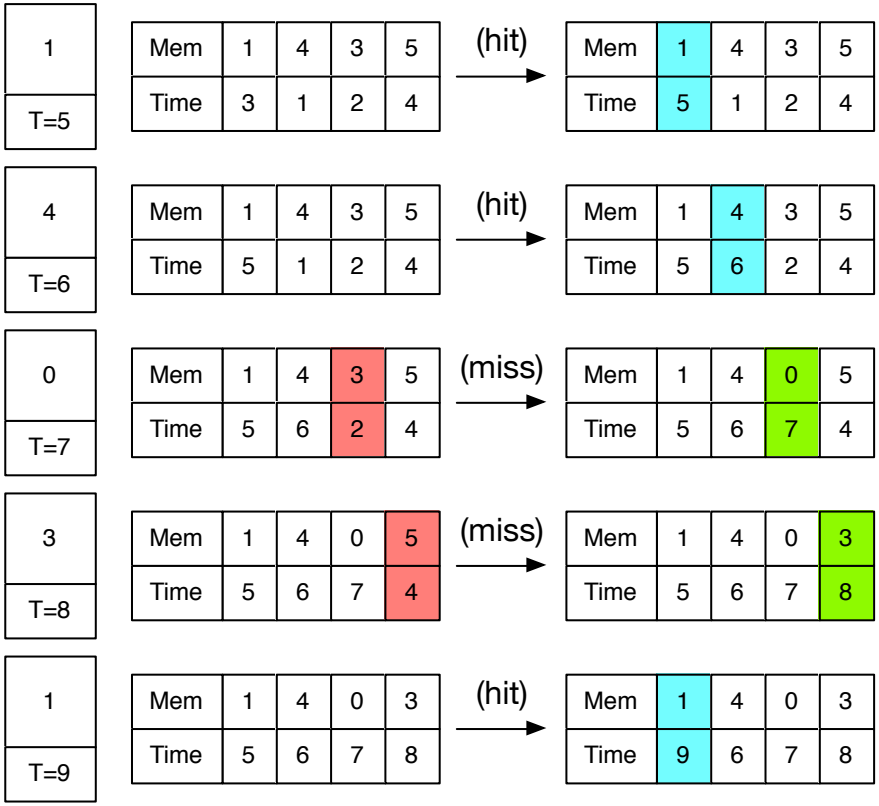
- This is the average access time through the cache.

Example: 4 Line LRU Cache




```
time 0, read memory[1]
time 1, read memory[4]
time 2, read memory[3]
time 3, read memory[1]
time 4, read memory[5]
time 5, read memory[1]
time 6, read memory[4]
time 7, read memory[0]
time 8, read memory[3]
time 9, read memory[1]
```

Example: 4 Line LRU Cache




```
time 0, read memory[1]
time 1, read memory[4]
time 2, read memory[3]
time 3, read memory[1]
time 4, read memory[5]
time 5, read memory[1]
time 6, read memory[4]
time 7, read memory[0]
time 8, read memory[3]
time 9, read memory[1]
```

Example: 4 Line LRU Cache

PennState

- Result: 6 misses, 4 hits
  - Pr(miss) = miss ratio = 0.6
- Assume
  - Hit time (100 usec)
  - Miss penalty (1000 usec)

```
time 0, read memory[1]
time 1, read memory[4]
time 2, read memory[3]
time 3, read memory[1]
time 4, read memory[5]
time 5, read memory[1]
time 6, read memory[4]
time 7, read memory[0]
time 8, read memory[3]
time 9, read memory[1]
```

- So the average memory access time is:

$$100\ usec + (0.6 * 1000\ usec) = 100 + 600 = 700\ usec$$

- Q: Why is the performance so poor?