Computer System Organization Recitation [Fall 2018]

CSCI-UA 201-006

R7: Cache

Cache

- Why do we need cache?
  - Memory is faster than disk but is still too slow compared to CPU's performance.
- Why do we need memory? Why can't we all use cache?
  - Cache is much more expensive than memory.
  - Technical issue: making such a large cache may be very difficult.
- Hierarchical design is seen everywhere in computer science.
  - https://en.wikipedia.org/wiki/Memory_hierarchy
  - https://gist.github.com/jboner/2841832

Cache implementation

- **Direct-mapped cache**
  - Like hash.
  - There may be a lot of collisions.
  - Page 17 of http://news.cs.nyu.edu/~jinyang/sp18-cso/notes/16-Memory_Cache.pdf

- **Fully-associative cache**
  - Each memory block can be put in arbitrary cache line.
  - Each lookup needs to search all the cache lines - computation costly.

- **Multi-way set-associative set**
  - Compromise of direct-mapped cache and fully-associative cache.
  - Page 24 of http://news.cs.nyu.edu/~jinyang/sp18-cso/notes/16-Memory_Cache.pdf

TLB

- TLB is a kind of cache — a cache of page tables to speedup virtual address translation.

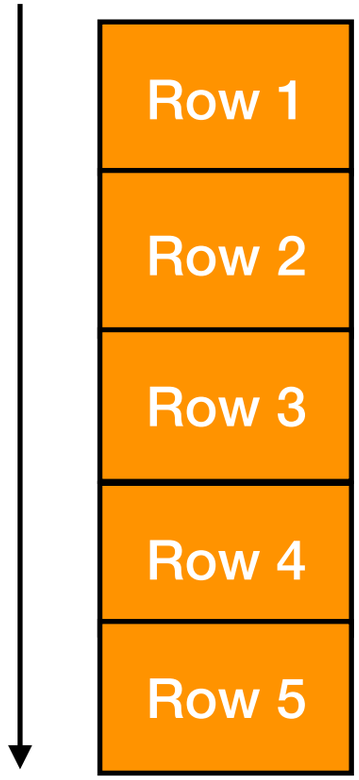
- TLB v.s memory cache
  - The input to a TLB is a virtual address.
  - The inputs to a memory cache are **a virtual address and a physical address(received later than the virtual address).**
  - The reason why a memory cache take both virtual address and physical address is to use the virtual address to lookup the set and use the physical address to lookup the tag.
    - This can parallelize searching cache and searching TLB.
    - Virtually indexed physically tagged cache.

Cache-friendly programming

- Temporal locality
  - A memory address that is referenced at one point in time will be referenced again sometime in the near future.
  - Example: Matrix multiplication
  - Inspiring cache eviction algorithm: LRU
- Spatial locality
  - The likelihood of referencing a memory address is high if an nearby memory address wast just referenced.
  - Example: array traversing
  - Counter-example: pointer chasing (linked list)

Cache-friendly programming

- Temporal locality
  - Example: Matrix multiplication
  - Inspiring cache eviction algorithm: LRU



Cache-friendly programming

A 3x4 integer array.

0	1	2	3
4	5	6	7
8	9	10	11

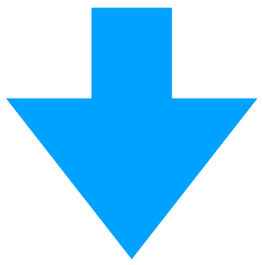
Suppose each cache line is 16 bytes.
And there is only one cache line.

How many cache miss if a program access this array row by row?

How many cache miss if a program access this array column by column?

Compact memory usage is cache-friendly

```
struct Song {
    char*          singer;
    int            length;
    long long      release_date;
}
```



If your design requires only 4 bytes for release_date

```
struct Song {
    char*          singer;
    int            length;
    int            release_date;
}
```

Suppose each cache line is 16 bytes.
And there is only one cache line.

How many cache miss per structure if we reference the original Song structure?

How many cache miss per structure if we reference the new Song structure?