Real-Time and Embedded Systems @ SIT

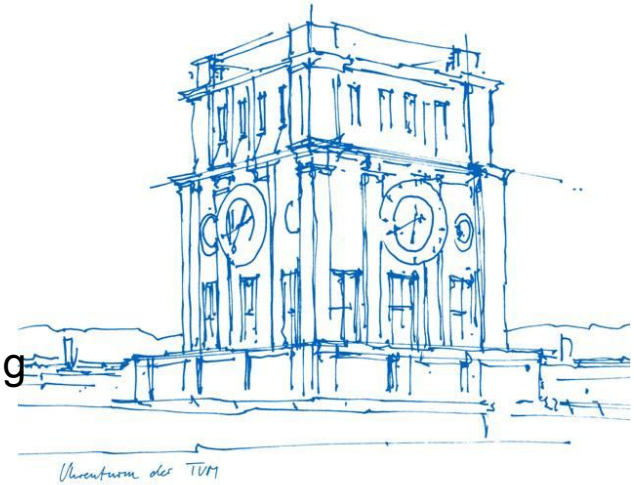
Caches

Alexander Hoffman

Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Real-Time Computer Systems

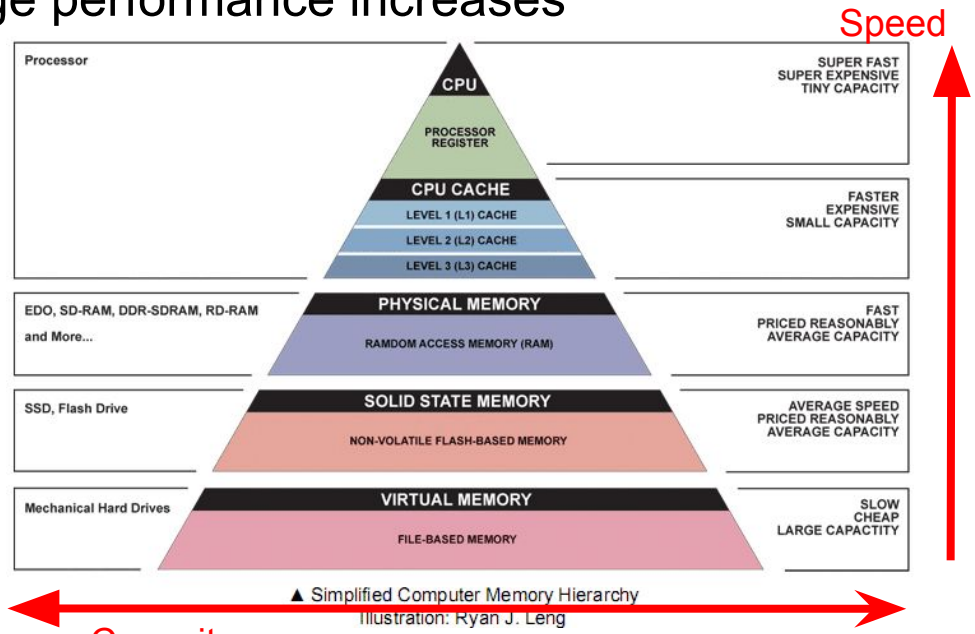


Memory: Types

- Registers
  - Fastest memory
  - Part of the CPU (very close, thus fast to access)
  - Access complexity is very low, eg. accessing main memory needs to be done via memory controllers, buses etc.
  - Not delayed by I/O, DMA, PCI etc
- SRAM
  - Flip-flops, thus 4 - 6 x more transistors required than DRAM
    - Expensive, cost in ~2010 for 1GB around $5000
  - Very fast, access times < 20ns
  - Used for cache
- DRAM
  - Uses one transistor and one capacitor
    - Cheap, cost in ~2010 for 1GB around $20-$75
  - Access times < 150ns
  - Used for system main memory (“RAM”)

Memory Hierarchy

- Idea system has large amounts of super fast memory
- Due to costs a memory hierarchy is created such that small amounts of very fast memory can provide large performance increases
- Each level acts as a buffer between lower levels and the higher levels, ultimately the CPU
- Higher the level the closer we get to the CPU, the faster we get due to locality
- Higher levels use smaller amounts of faster memory



https://sites.google.com/site/cachememory2011/memory-hierarchy

Memory Hierarchy Requirements

- If a level is closer to the processor, it must….
  - Be smaller
  - Be faster
  - Contain a subset (most recently used data) of the lower levels beneath it
  - Contain **all** the data in the higher levels above it
- Thus, lowest level (usually disk or main memory) contains all available data

Memory Speed

- Seperate from memory technology speed is affected by locality
- Locality
  - **Temporal locality**: item will be referenced again soon
  - **Spatial locality**: items nearby (physically) will tend to be referenced soon
- We are focusing on cache and main memory
- Terms:
  - Memory block: minimum unit of data that can be loaded into cache
  - Hit: when required data is in the upper level (cache)
  - Miss: when the required data is not in the upper level (cache)

Cache

- Thus, cache is smaller than main memory
- Almost always cannot contains all required program data at any point in time
- This presents the problems:
  - How do we know if our required data is in the cache
  - If it is, how do we find it?
  - If it is not then where do we put it when loading it?

Cache Placement Policies

- Decide how to search cache
- Where to place new data blocks in cache
- Common policies
  - Direct mapped cache
  - Fully associative cache
  - Set associative cache

We will focus on direct-mapped cache in this course

Cache Hits vs Misses

- **Hit**, cache policy has determined that the data we want is in cache!
  - Reading:
    - Do nothing, just use values in cache
  - Writing:
    - Problem is that the modified value in cache must be reflected in main memory
    - Either:
      - Replace data in cache and memory (*write-through)*
      - Write the data into the cache (*write-back* the cache later)

Cache Hits vs Misses

- **Miss**, cache policy has determined that the data we want is **NOT** in cache!
  - Reading:
    - Stall CPU, fetch block from memory, deliver to cache and restart load instruction
  - Writing:
    - Read the missing block into cache and then write the word (*allocate* on write miss)
    - Do not read the cache line; just write to main memory (*no allocate* on write miss)

Cache Hits vs Misses

- Performance of cache can thus be improved by:
  - Decreasing the miss ration: *associativity*
  - Decreasing the miss penalty: *multilevel caches*

Cache Line

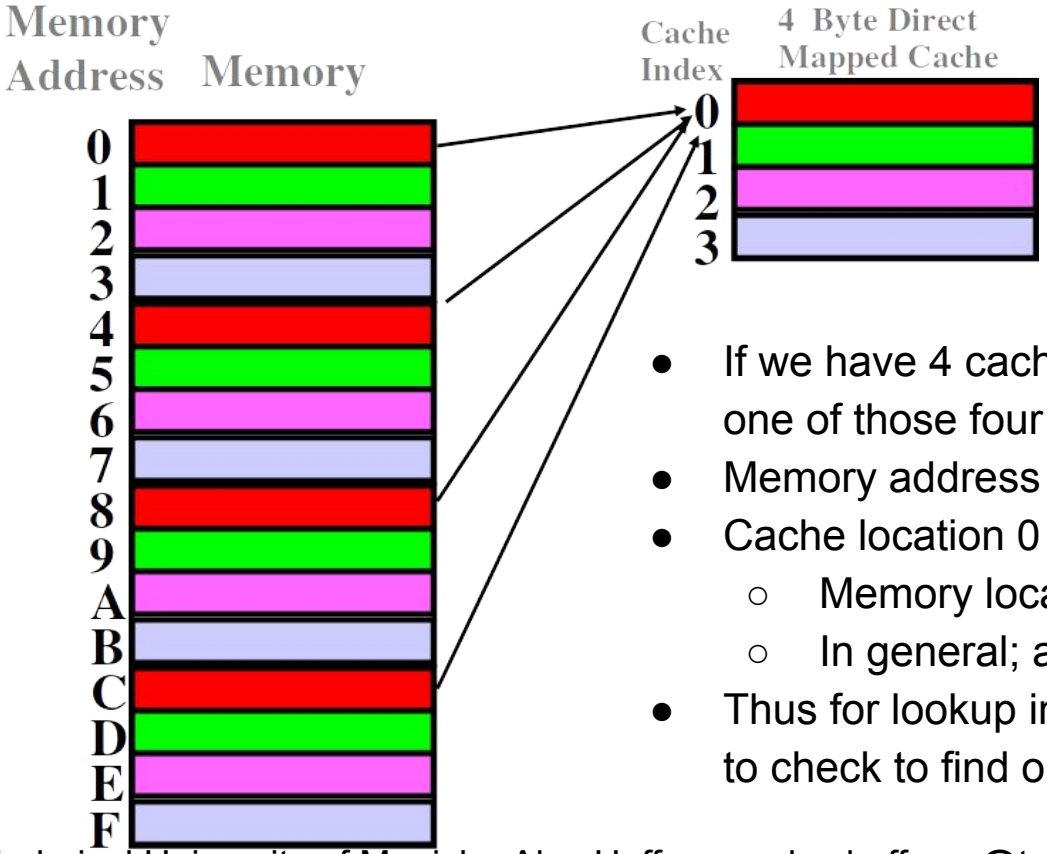
- We know that there is a minimum memory block size that the cache is constructed from, often 64 bytes
- The total cache is thus a collection of these blocks
- Each minimum memory block size memory block in cache is called a “cache line”
- Thus, every memory block when read from main memory ends up filling a cache line completely
- Cache lines are never partially filled

Eg. Cache with capacity of 8192 bytes and 64 byte cache line size will have 128 cache lines

Direct Mapped Cache

- Cache is organized into multiple sets where each set is associated with a “cache line”
  - Cache line: the unit of data transfer between cache and main memory, ie. a memory block
- Based on the address of the memory block in main memory it will be “directly mapped” to a certain single cache line
- Ie. there is exactly **one** location in cache where each data block would be found
- Let’s use a visual to help explain

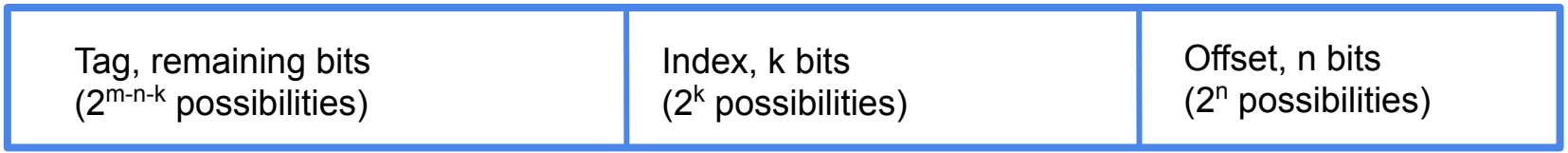
Direct Mapped Cache



- If we have 4 cache lines then each block is mapped onto one of those four lines
- Memory address used to determine which line
- Cache location 0 can be occupied by data from:
  - Memory location 0, 4, 8,...
  - In general; any memory location that is a multiple of 4
- Thus for lookup in cache we know which line we would need to check to find our data

Direct Mapped Cache Addresses

- Address has three fields
  - Offset: specifies where in the **cache line** the specific data word/byte is located
  - Index: specifies which **cache line** the **data block** mapps into
  - Tag: remaining bits that can be used for checking that block in cache is correct



Direct Mapped Cache Address Example

- 32 bit system
- Address is 0b10110000100101010010100111100010
- Let's say we have cache lines that hold 64 bytes, ie. the offset specifies where between 0-63 the byte we want is stored once we have the correct block
  - Thus we need 6 bits to show 0-63
  - Our byte of interest is at 100010 offset, ie it is the 34th byte in the line (zero index)
- Let's say our system has 32 cache lines, ie. each block maps onto one of 0-31 cache lines
  - Need 5 bytes to represent this
  - 00111 our data block maps onto the 7th (zero index) cache line
- Remaining bits in red are used to see if this is the correct memory block as it is unique to this memory block because other memory blocks with this tag will be on other lines

Direct Mapped Cache Lookup

Thus, process of checking if cache hits or misses is:

- Use index to find appropriate cache line
- Compare tag to determine if we have a cache hit or miss
- If hit:
  - Use offset to access data we are after
- If miss:
  - Load data into cache from memory

Fully Associative Cache

- A memory block can occupy any cache line, ie. no index field in address, only tag and offset
- Cache lines are tracked as valid or invalid (initialized at boot to invalid) and set to valid when data placed into line
- Placing block:
  - An invalid cache line is searched for
  - If one is not found then a cache line is evicted and used
  - Eviction is done via a replacement policy
- Searching for block:
  - Tag portion of address is compared to each block in cache

Fully Associative Cache

- Advantages:
  - Flexible
  - Large cache utilization
  - Can use a wide variety of replacement algorithms for cache misses
- Disadvantages:
  - Slow due to iterating through all lines in search
  - Power hungry due to search
  - Most expensive due to the cost of associative-comparison hardware

Set Associative Cache

- Trade off between direct mapped and fully associative cache
- Memory block is mapped onto a set of cache lines and then placed into any cache line in the set
- Thus, also has an index field in the address
- Placing:
  - Set is determined from index in address
  - Data is placed in available cache line, if none are available the replacement policy frees a line
- Locating:
  - Set is determined from index in address
  - Address tag is compared with all cache lines in set to see if the data exists in cache