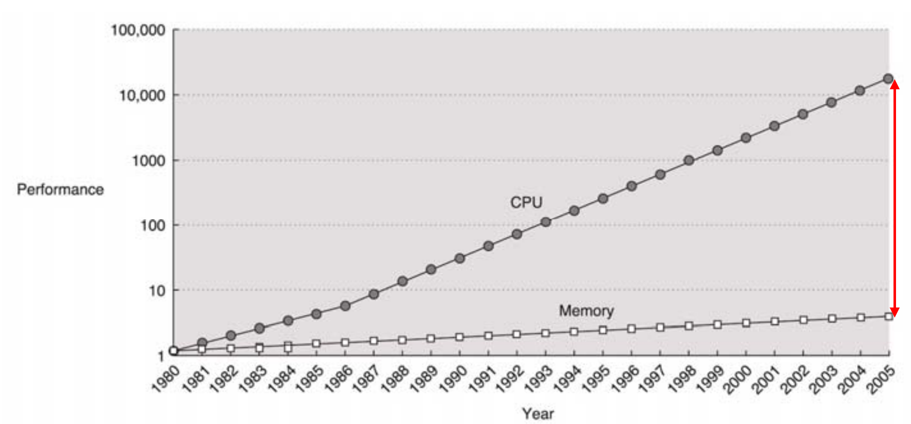
CS2410: Computer Architecture

L1 cache design

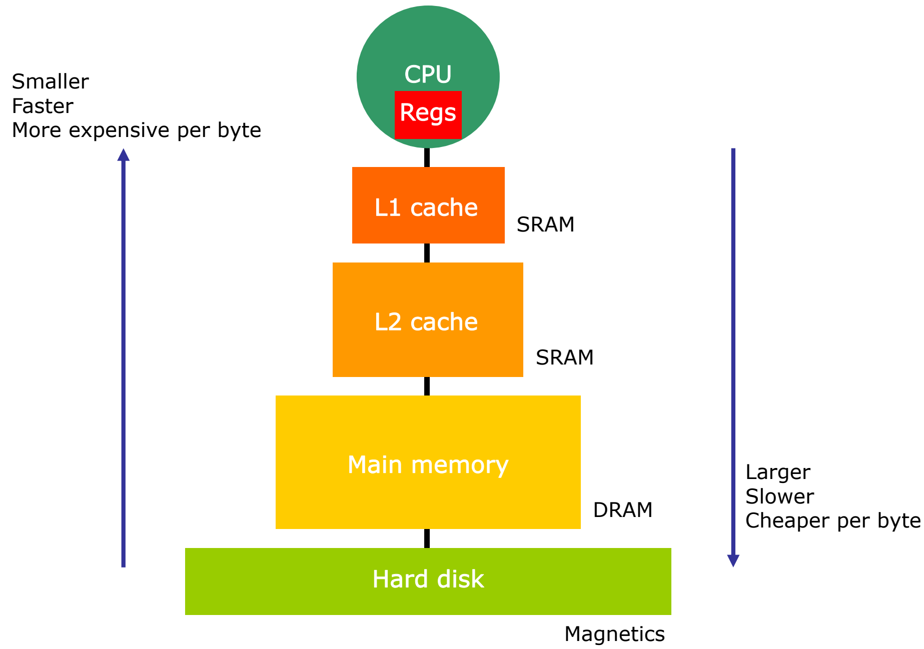
Sangyeun Cho

Computer Science Department
University of Pittsburgh

Why memory hierarchy?



Memory hierarchy

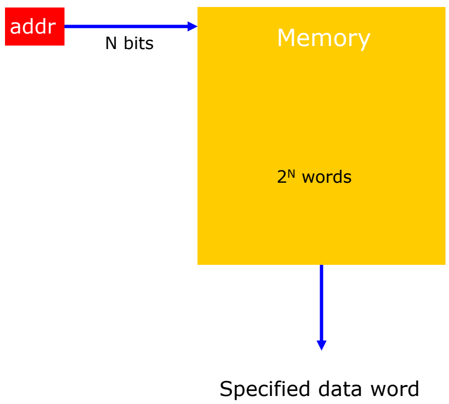


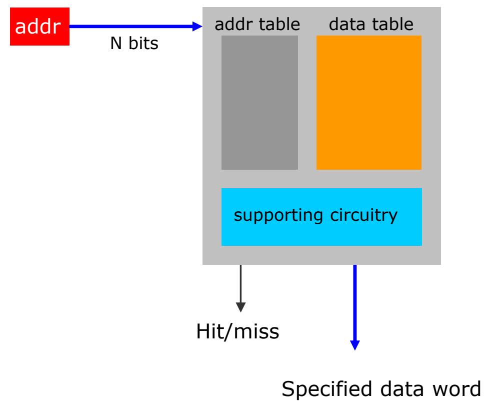
Memory hierarchy goals

- To provide CPU with necessary data (and instructions) as quickly as possible
  - To achieve this goal, a cache should keep frequently used data
  - "Cache hit" when CPU finds a requested data in cache
  - Hit rate = # of cache hits/# of cache accesses
  - Average memory access latency (AMAL) = cache hit time + (1 – cache hit rate) × miss penalty
    - To decrease AMAL, reduce hit time, increase hit rate, and reduce miss penalty
- To reduce traffic on memory bus
  - Cache becomes a "filter"
  - Reduces the bandwidth requirements from the main memory
  - Typically, max. L1 bandwidth (to CPU) > max. L2 bandwidth (to L1) > max. memory bandwidth

Cache organization

- Caches use “blocks” or “lines” (block > byte) as their granule of management
- Memory > cache: we can only keep a subset of memory blocks
- Cache is in essence a fixed-width hash table; the memory blocks kept in a cache are thus associated with their addresses (or “tagged”)





L1 cache vs. L2 cache

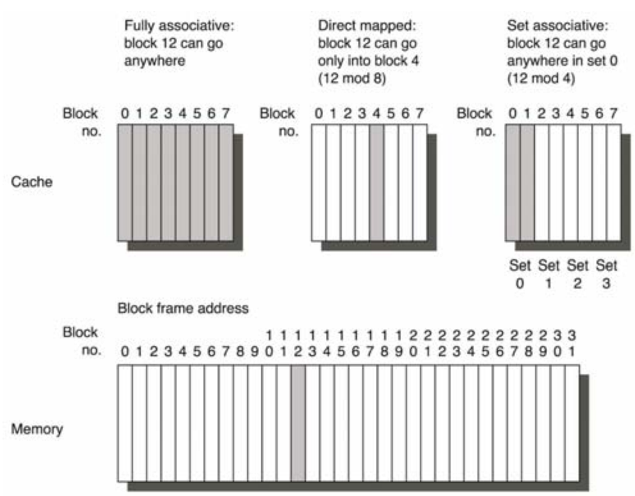
- Their basic parameters are similar
  - Associativity, block size, and cache size (the capacity of data array)
- Address used to index
  - L1: typically virtual address (to quickly index first)
    - Using a virtual address causes some complexities
  - L2: typically physical address
    - Physical address is available by then
- System visibility
  - L1: not visible
  - L2: *page coloring* can affect hit rate
- Hardware organization (esp. in multicores)
  - L1: private
  - L2: often shared among cores

Key questions

- Where to place a block?
- How to find a block?
- Which block to replace for a new block?
- How to handle a write?
  - Writes make a cache design much more complex!

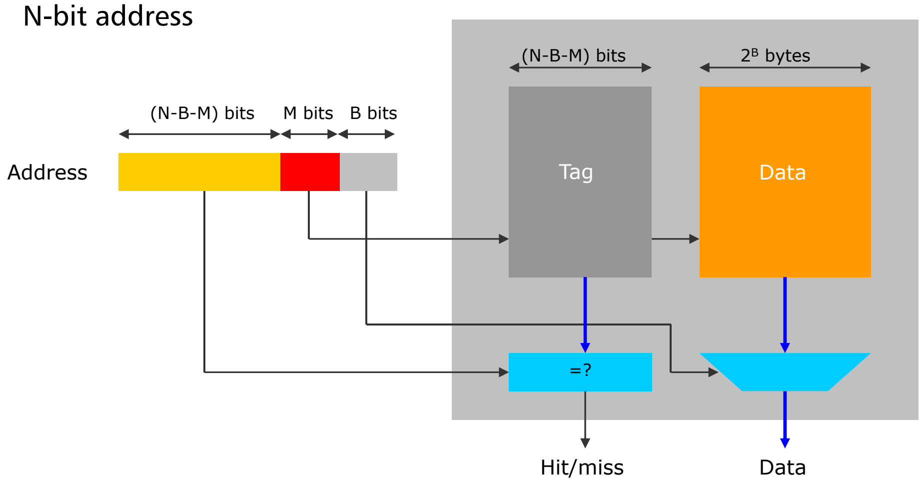
Where to place a block?

- Block placement is a matter of *mapping*
- If you have a simple rule to place data, you can find them later using the same rule



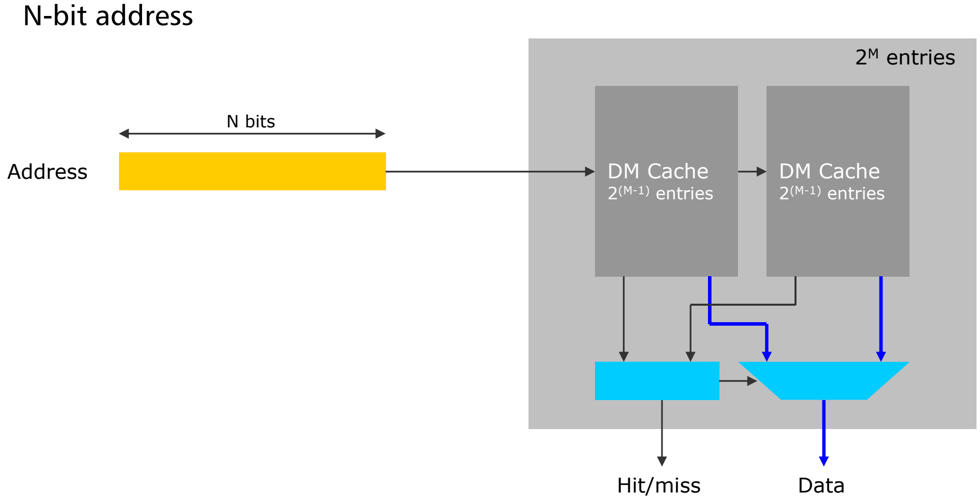
Direct-mapped cache

- 2^B byte block
- 2^M entries
- N-bit address



2-way set-associative cache

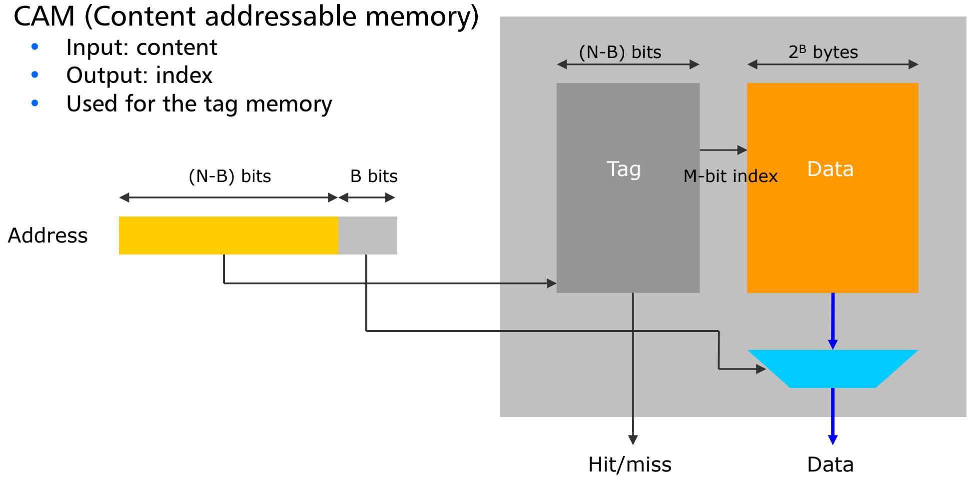
- 2^B byte block
- 2^M entries
- N-bit address



Fully associative cache

- 2^B byte block
- 2^M entries
- N-bit address

- CAM (Content addressable memory)
  - Input: content
  - Output: index
  - Used for the tag memory



Why caches work (or do not work)

- Principle of locality
  - Temporal locality
    - If the location A is accessed now, it'll be accessed again soon
  - Spatial locality
    - If the location A is accessed now, the location nearby (e.g., A+1) will be accessed soon

- Can you explain how locality is manifested in your program (at the source code level)?
  - Data
  - Instructions

- Can you write the same program twice, having
  - A high degree of locality
  - Badly low locality

Which block to replace?

- Which block to replace, to make room for a new block on a miss?
- Goal: minimize the # of total misses
- Trivial in a direct-mapped cache
- N choices in N-way associative cache

- What is the optimal policy?
  - MRU (most remotely used) is considered optimal
  - This is an oracle scheme – we do not know the future

- Replacement approaches
  - LRU (least recently used) – look at the past to predict the future
  - FIFO (first in first out) – honor the new ones
  - Random – don't remember anything
  - Cost-based – what is the cost (e.g., latency) of bringing this block again?

How to handle a write?

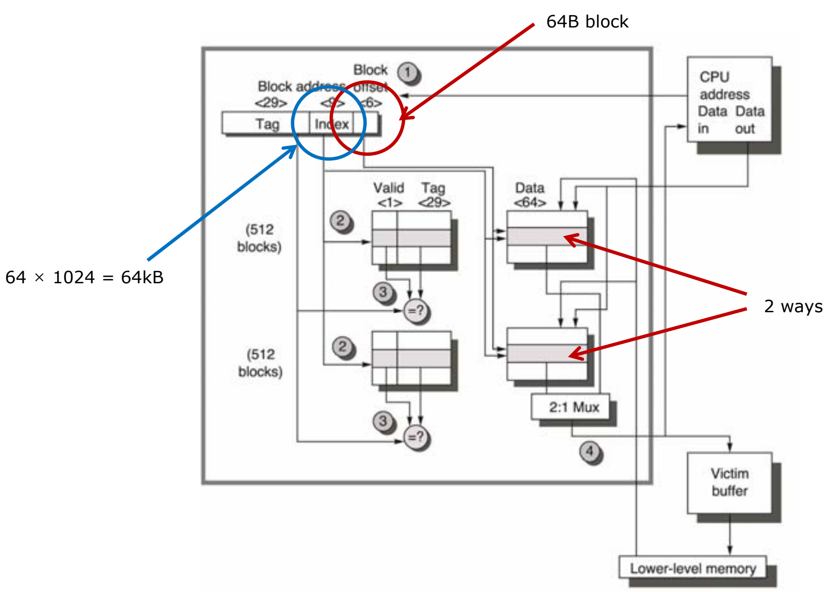
- Design considerations
  - Performance
  - Design complexity
- Allocation policy (on a miss)
  - Write-allocate
  - No-write-allocate
  - Write-validate
- Update policy
  - Write-through
  - Write-back

- Typical combinations
  - Write-back with write-allocate
  - Write-through with no-write-allocate

Write-through vs. write-back

- L1 cache: advantages of write-through + no-write-allocate
  - Simple control
  - No stalls for evicting dirty data on L1 miss with L2 hit
  - Avoids L1 cache pollution with results that are not read for a while
  - Avoids problems with coherence (L2 is consistent with L1)
  - Allows efficient transient error handling: parity protection in L1 and ECC in L2
  - What about high traffic between L1 and L2, esp. in a multicore processor?
- L2 cache: advantages of write-back + write-allocate
  - Typically reduces overall bus traffic by filtering all L1 write-through traffic
  - Better able to capture temporal locality of infrequently written memory locations
  - Provides a safety net for programs where write-allocate helps a lot
    - Garbage-collected heaps
    - Write-followed-by-read situations
    - Linking loaders (if unified cache, need not be flushed before execution)
- Some ISA/caches support explicitly installing cache blocks with empty contents or common values (*e.g.*, zero)

Alpha 21264 example



More examples

- IBM Power5
  - L1I: 64kB 2-way 128B block LRU
  - L1D: 32kB 4-way 128B block write-through LRU
  - L2: 1.875MB (3 banks) 10-way 128B block pseudo LRU

- Intel Core Duo
  - L1I: 32kB 8-way 64B block LRU
  - L1D: 32kB 8-way 64B block LRU write-through
  - L2: 2MB 8-way 64B line LRU write-back

- Sun Niagara
  - L1I: 16kB 4-way 32B block random
  - L1D: 8kB 4-way 16B block random write-through write no-allocate
  - L2: 3MB 4 banks 64B block 12-way write-back

Impact of caches on performance

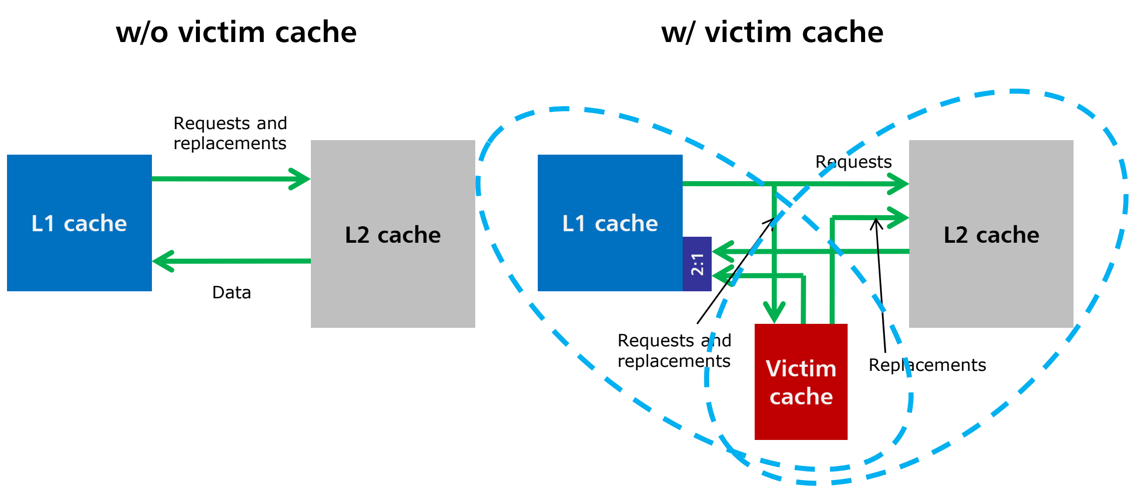
- Average memory access latency (AMAL) = cache hit time + (1 – cache hit rate) × miss penalty

- Example 1
  - Hit time = 1 cycle
  - Miss penalty = 100 cycles
  - Miss rate = 2%
  - Average memory access latency?
- Example 2
  - 1GHz processor
  - Two configurations: 16kB direct-mapped, 16kB 2-way
  - Two miss rates: 3%, 2%
  - Hit time = 1 cycle, but clock cycle time is stretched by 1.1 in 2-way
  - Miss penalty = 100ns (how many cycles?)
  - Average memory access latency?

1. Reducing miss penalty

- Multi-level caches
  - miss $penalty_{L1}$ = hit $time_{L2}$ + miss $rate_{L2}$ × miss $penalty_{L2}$
- Critical word first and early restart
  - When L2-L1 bus width is smaller than L1 cache block
- Giving priority to read misses
  - Esp. in dynamically scheduled processors
- Merging write buffer
- Victim caches
- Non-blocking caches
  - Esp. in dynamically scheduled processors

Victim cache



- Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," ISCA 1990.

Categorizing misses

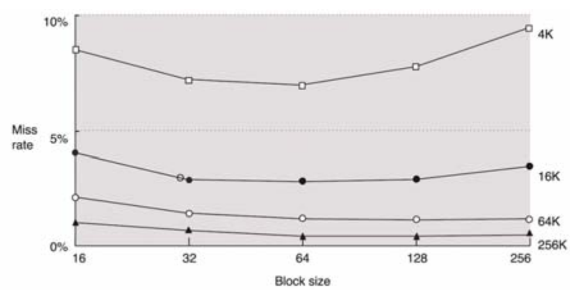
- Compulsory
  - "I've not met this block before…"
- Capacity
  - "Working set is larger than my cache…"
- Conflict
  - "Cache has space but blocks in use map to busy sets…"

- How can you measure contributions from different miss categories?

2. Reducing miss rate

- Larger block size
  - Reduces compulsory misses
- Larger cache size
  - Tackles capacity misses
- Higher associativity
  - Attacks conflict misses
- Prefetching
  - Relevancy issues (due to pollution)
- Pseudo-associative caches

- Compiler optimizations
  - Loop interchange
  - Blocking



3. Reducing hit time

- Small & simple cache (e.g., direct-mapped cache)
  - Test different cache configurations using cacti (http://quid.hpl.hp.com:9082/cacti/)
- Avoid address translation during cache indexing
- Pipeline access

Prefetching

- Memory hierarchy generally works well
  - L1 cache: 1- ~ 3-cycle latency
  - L2 cache: 8- ~ 13-cycle latency
  - Main memory: 100- ~ 300-cycle latency
  - Cache hit rates are critical to high performance
- Prefetching: if we know what data we'll need from level-N cache *a priori*, get data from level-(N+1) and place it in level-N cache before the data is accessed by the processor
- What are design goals and issues?
  - E.g., shall we load prefetched block in L1 or not?
  - E.g., instruction prefetch vs. data prefetch

Evaluating a prefetching scheme

- Coverage
  - By applying a prefetching scheme, how many misses are covered?
- Timeliness
  - Are prefetched data arriving early enough so that the (otherwise) miss latency is effectively hidden?
- Relevance and usefulness
  - Are prefetched blocks actually used?
  - Do they replace other useful data?

- How would program execution time change?
  - Especially, dynamically scheduled processors have intrinsic ability to tolerate long latencies to a degree

Hardware schemes

- Prefetch data under hardware control
  - Without software modification
  - Without increasing instruction count

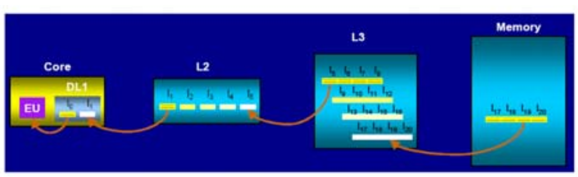
- Hardware prefetch engines
  - Programmable engines
    - Program this engine with data access pattern information
    - Who programs this engine, then?
  - Automatic
    - Hardware detects access behavior

Stride detection

- A popular method using a reference prediction table
  - Load instruction PC
  - Last address A_{i-1}
  - Last stride $S = A_{i-1} - A_{i-2}$
  - Other flags, *e.g.*, confidence, time stamp, …
  - Next address for this PC $A_i = A_{i-1} + S$
- In practice, simpler stream buffer like methods are often used
  - IBM Power4/5 use 8 stream buffers between L1 & L2, L2 & L3 (or main memory)

Power4 example

- 8 stream buffers: ascending/descending
  - Requires at least 4 sequential misses to install a stream
- Supports L2 to L1, L3 to L2, memory to L3



- Based on physical address
  - When page boundary is met, stop there
- Software interface
  - To explicitly (and quickly) install a stream

Software schemes

- Use *prefetch* instruction – needs ISA support
  - Check if the desired block is in cache already
    - If not, bring the cache block into the cache
    - If yes, do nothing
  - In any case, prefetch request is a performance hint and not a correctness requirement
    - Not acting on it must not affect program output
- Compiler or programmer then inserts prefetch instructions in programs
- Hardware prefetch vs. software prefetch

Software prefetch example

```
for (int i=0; i<100; ++i)
{
        a[i] = b[i] + c[i];
}
```

```
prefetch(&b[0]);
prefetch(&c[0]);
for (i=0; i<100; i++)
{
        prefetch (&b[i+4]);
        prefetch (&c[i+4]);
        a[i] = b[i] + c[i];
}
```