Performance metrics for caches

- Basic performance metric: hit ratio $h$
  
  \[ h = \frac{\text{Number of memory references that hit in the cache}}{\text{total number of memory references}} \]
  
  Typically $h = 0.90 \text{ to } 0.97$

- Equivalent metric: miss rate $m = 1 - h$

- Other important metric: Average memory access time
  
  \[ \text{Av. Mem. Access time} = h \times T_{\text{cache}} + (1-h) \times T_{\text{mem}} \]
  
  where $T_{\text{cache}}$ is the time to access the cache (e.g., 1 cycle) and $T_{\text{mem}}$ is the time to access main memory (e.g., 50 cycles)

  (Of course this formula has to be modified the obvious way if you have a hierarchy of caches)

Parameters for cache design

- Goal: Have $h$ as high as possible without paying too much for $T_{\text{cache}}$
- The bigger the cache size (or capacity), the higher $h$.
  - True but too big a cache increases $T_{\text{cache}}$
- Limit on the amount of “real estate” on the chip (although this limit is not present for 1st level caches)
- The larger the cache associativity, the higher $h$.
  - True but too much associativity is costly because of the number of comparators required and might also slow down $T_{\text{cache}}$ (extra logic needed to select the “winner”)

- Block (or line) size
  - For a given application, there is an optimal block size but that optimal block size varies from application to application

Parameters for cache design (ct’d)

- Write policy (see later)
  - There are several policies with, as expected, the most complex giving the best results

- Replacement algorithm (for set-associative caches)
  - Not very important for caches with small associativity (will be very important for paging systems)

- Split I and D-caches vs. unified caches.
  - First-level caches need to be split because of pipelining that requires an instruction every cycle. Allows for different design parameters for I-caches and D-caches
  - Second and higher level caches are unified (mostly used for data)

Example of cache hierarchies (don’t quote me on these numbers)

<table>
<thead>
<tr>
<th>MICRO</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21064</td>
<td>8K(I), 8K(D), WT, 32B</td>
<td>128K to 512K, WB, 2-way, 32B</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>8K(I), 8K(D), WT, 32B, D-L cache</td>
<td>64K, WB, on-chip, 3-way, 32B, L-1 free</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>64K(I), 64K(D), 3-way, 32B</td>
<td>up to 16MB</td>
</tr>
<tr>
<td>Pentium</td>
<td>8K(I), 8K(D), both, 2-way, 32B</td>
<td>Depends</td>
</tr>
<tr>
<td>Pentium II, III</td>
<td>16K(I), 16K(D), WB, 4-way(D), 2-way(D), 32BJ-1 free</td>
<td>512K, 32B, 4-way, tightly-coupled</td>
</tr>
</tbody>
</table>
Examples (cont’d)

<table>
<thead>
<tr>
<th>System</th>
<th>Size (I), Size (D), Type, Associativity, Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPC 620</td>
<td>32K(I), 32K(D), WB, 8-way, 64B</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>32K(I), 32K(D), LRU, 2-way, 32B</td>
</tr>
<tr>
<td>SUN UltraSparcI1</td>
<td>64K(I), 64K(D), LRU, 4-way, 32B</td>
</tr>
<tr>
<td>AMD K7</td>
<td>64K(I), 64K(D)</td>
</tr>
</tbody>
</table>

Back to associativity

- **Advantages**
  - Reduces conflict misses
- **Disadvantages**
  - Needs more comparators
  - Access time is longer (need to choose among the comparisons, i.e., need of a multiplexer)
  - Replacement algorithm is needed and could get more complex as associativity grows

Replacement algorithm

- None for direct-mapped
- Random or LRU or pseudo-LRU for set-associative caches
  - LRU = “least recently used”: means that the entry in the set which has not been used for the longest time will be replaced (think about a stack)

Impact of associativity on performance

- Typical curve: Biggest improvement from direct-mapped to 2-way, then 2 to 4-way, then incremental
Impact of block size

- Recall block size = number of bytes stored in a cache entry
- On a cache miss the whole block is brought into the cache
- For a given cache capacity, advantages of large block size:
  - decrease number of blocks: requires less real estate for tags
  - decrease miss rate IF the programs exhibit good spatial locality
  - increase transfer efficiency between cache and main memory
- For a given cache capacity, drawbacks of large block size:
  - increase latency of transfers
  - might bring unused data IF the programs exhibit poor spatial locality
  - Might increase the number of conflict/capacity misses

Classifying the cache misses: The 3 C’s

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large block sizes
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses
- There is a fourth C: coherence misses (cf. multiprocessors)

Impact of block size on performance

Classifying the cache misses: The 3 C’s

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large block sizes
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses
- There is a fourth C: coherence misses (cf. multiprocessors)

Impact of block size on performance

Classifying the cache misses: The 3 C’s

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large block sizes
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses
- There is a fourth C: coherence misses (cf. multiprocessors)

Impact of block size on performance

Classifying the cache misses: The 3 C’s

- Compulsory misses (cold start)
  - The first time you touch a block. Reduced (for a given cache capacity and associativity) by having large block sizes
- Capacity misses
  - The working set is too big for the ideal cache of same capacity and block size (i.e., fully associative with optimal replacement algorithm). Only remedy: bigger cache!
- Conflict misses (interference)
  - Mapping of two blocks to the same location. Increasing associativity decreases this type of misses
- There is a fourth C: coherence misses (cf. multiprocessors)
**Writing in a cache**

- **On a write hit, should we write:**
  - In the cache only (write-back policy)
  - In the cache and main memory (or next level cache) (write-through policy)
- **On a cache miss, should we**
  - Allocate a block as in a read (write-allocate)
  - Write only in memory (write-around)

**Write-through policy**

- Write-through (aka store-through)
  - On a write hit, write both in cache and in memory
  - On a write miss, the most frequent option is write-around, i.e., write only in memory
- **Pros:**
  - Consistent view of memory;
  - Memory is always coherent (better for I/O);
  - More reliable
    - Memory units typically store extra bits with each word to detect/correct errors ("ECC" = Error-correcting code)
  - ECC not required for cache if write-through is used
- **Cons:**
  - More memory traffic (can be alleviated with write buffers)

**Write-back policy**

- Write-back (aka copy-back)
  - On a write hit, write only in cache (requires dirty bit to say that value has changed)
  - On a write miss, most often write-allocate (fetch on miss) but variations are possible
  - We write to memory when a dirty block is replaced
- **Pros:**
  - Less memory traffic
  - Pro-con reverse of write-through

**Cutting back on write backs**

- In write-through, you write only the word (byte) you modify
- In write-back (when finally writing to memory), you write the entire block
  - But you could have one dirty bit/word so on replacement you’d need to write only the words that are dirty
Hiding memory latency

- On write-through, the processor has to wait till the memory has stored the data.
- Inefficient since the store does not prevent the processor to continue working.
- To speed-up the process, have write buffers between cache and main memory:
  - Write buffer is a (set of) temporary register that contains the contents and the address of what to store in main memory.
  - The store to main memory from the write buffer can be done while the processor continues processing.
- Same concept can be applied to dirty blocks in write-back policy.

Coherency: caches and I/O

- In general I/O transfers occur directly to/from memory from/to disk.
- The problem: what if the processor and the I/O are accessing the same words of memory?
  - Want processor and I/O to have a "coherent" view of memory.
- Similar coherence problem arises with multiple CPUs:
  - Each CPU accesses the same memory, but keeps its own cache.

Preserving coherences with I/O

- What happens for memory to disk:
  - With write-through memory is up-to-date. No problem.
  - With write-back memory is not up-to-date. Before I/O is done, need to "purge" cache entries that are dirty and that will be sent to the disk.
- What happens from disk to memory:
  - The I/O may change a memory location that is currently in the cache.
  - The entries in the cache that correspond to memory locations that are read from disk must be invalidated.
  - Need of a valid bit in the cache (or other techniques).

Reducing Cache Misses with more "Associativity" -- Victim caches

- Example of an "hardware assist":
- Victim cache: Small fully-associative buffer "behind" the cache and "before" main memory.
- Of course can also exist if cache hierarchy (behind L1 and before L2, or behind L2 and before main memory).
- Main goal: remove some of the conflict misses in direct-mapped caches (or any cache with low associativity).
Operation of a Victim Cache

- 1. Hit in L1; Nothing else needed
- 2. Miss in L1 for block at location b, hit in victim cache at location v: swap contents of b and v (takes an extra cycle)
- 3. Miss in L1, miss in victim cache: load missing item from next level and put in L1; put entry replaced in L1 in victim cache; if victim cache is full, evict one of its entries.
- Victim buffer of 4 to 8 entries for a 32KB direct-mapped cache works well.