Cache Performance

\[
\text{Avg. memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
\]

with
- Hit time: time to access a data item which is available in the cache
- Miss rate: ratio of no. of memory access leading to a cache miss to the total number of cache accesses
- Miss penalty: time/cycles required for moving a data item into the cache
Processor Performance

- CPU equation:
  \[
  \text{CPU time} = (\text{Clock cycles}_{\text{CPU execution}} + \text{Clock cycles}_{\text{memory stall}}) \times \text{clock cycle time}
  \]

- Can avg. memory access time really be ‘mapped’ to CPU time?
  - Not all memory stall cycles are due to cache misses
    - We ignore that on the following slides
  - Depends on the processor architecture
    - In-order vs. out-of-order execution

- For out-of-order processors need the ‘visible’ portion of the miss penalty
  \[
  \text{Memory stall cycles/instruction} = \frac{\text{Misses/instruction} \times (\text{Total miss latency} - \text{overlapped miss latency})}{\text{clock cycle time}}
  \]

Types of cache misses

- **Compulsory Misses**: first access to a block cannot be in the cache (cold start misses)

- **Capacity Misses**: cache cannot contain all blocks required for the execution

- **Conflict Misses**: cache block has to be discarded because of block replacement strategy
Six basic cache optimizations

• *Reducing cache miss rate*:  
  - Larger block size, larger cache size, higher associativity

• *Reducing miss penalty*:  
  - Multilevel caches, giving priority to read misses over writes

• *Reducing cache hit time*:  
  - Avoid address translation when indexing the cache

Larger block size

• Reduces compulsory misses
• For a constant cache size: reduces the number of blocks  
  - Increases conflict misses
Larger caches

- Reduces capacity misses
- Might increase hit time (e.g., if implemented as off-chip caches)
- Cost limitations

Higher Associativity

- Reduces miss rate
- Increases hit-time
Multilevel caches (I)

- Dilemma: should the cache be fast or should it be large?
- Compromise: multi-level caches
  - 1st level small, but at the speed of the CPU
  - 2nd level larger but slower

Avg. memory access time = Hit time \( L_1 \) + Miss rate \( L_1 \) x Miss penalty \( L_1 \)
and
Miss penalty \( L_1 \) = Hit time \( L_2 \) + Miss rate \( L_2 \) x Miss penalty \( L_2 \)

Multilevel caches (II)

- Local miss rate: rate of number of misses in a cache to total number of accesses to the cache
- Global miss rate: ratio of number of misses in a cache number of memory access generated by the CPU
  - 1st level cache: global miss rate = local miss rate
  - 2nd level cache: global miss rate = Miss rate \( L_1 \) x Miss rate \( L_2 \)
- Design decision for the 2nd level cache:
  1. Direct mapped or n-way set associative?
  2. Size of the 2nd level cache?
Multilevel caches (II)

- Assumptions in order to decide question 1:
  - Hit time L2 cache:
    - Direct mapped cache: 10 clock cycles
    - 2-way set associative cache: 10.1 clock cycles
  - Local miss rate L2:
    - Direct mapped cache: 25%
    - 2-way set associative: 20%
  - Miss penalty L2 cache: 200 clock cycles
    Miss penalty_{direct mapped L2} = 10 + 0.25 \times 200 = 60 \text{ clock cycles}
    Miss penalty_{2-way assoc. L2} = 10.1 + 0.2 \times 200 = 50.1 \text{ clock cycles}
- If L2 cache synchronized with L1 cache: hit time = 11 clock cycles
  Miss penalty_{2-way assoc. L2} = 11 + 0.2 \times 200 = 51 \text{ clock cycles}

Multilevel caches (III)

- Multilevel inclusion: 2\text{nd} level cache includes all data items which are in the 1\text{st} level cache
  - Applied if size of 2\text{nd} level cache >> size of 1\text{st} level cache
- Multilevel exclusion: Data of L1 cache is never in the L2 cache
  - Applied if size of 2\text{nd} level cache only slightly bigger than size of 1\text{st} level cache
  - Cache miss in L1 often leads to a swap of an L1 block with an L2 block
Giving priority to read misses over writes

- Write-through caches use a write-buffer to speed up write operations
- Write-buffer might contain a value required by a subsequent load operations
- Two possibilities for ensuring consistency:
  - A read resulting in a cache miss has to wait until write buffer is empty
  - Check the contents of the write buffer and take the data item from the write buffer if it is available
- Similar technique used in case of a cache-line replacement for n-way set associative caches

Avoiding address translation

- Translation of virtual address to physical address required to access memory/caches
- Using of virtual address in caches would avoid address translation
- Problems:
  - Two processes might use the same virtual address without meaning the same physical address <-> adding of a process identifier tag (PID) to the cache address tag
  - Page protection: cache has to be flushed for every process switch
- Separation of indexing of cache vs. comparing address
  - Use virtual address for indexing and physical address for tag comparison
Ten advanced cache optimizations

- **Reducing hit time:**
  - Small and simple caches, way prediction

- **Increase cache bandwidth:**
  - Pipelined caches, multi-bank caches and non-blocking caches

- **Reducing miss penalty:**
  - Critical word first and early restart, merging write buffers

- **Reducing miss rate:**
  - Compiler optimizations

- **Reducing miss penalty and miss rate via parallelism:**
  - Hardware prefetch, compiler prefetch

Small and simple caches

- Smaller and simpler caches are faster ‘by nature’
- Cache hit time depends on
  - Comparison of the tag and the address
  - No. of comparisons having to be executed
  - Clock-cycle of the cache, i.e. on-chip vs. off-chip caches
Way Prediction

- Way prediction
  - Add a bit to n-way associative caches in order to predict which of the cache blocks will be used
    - The predicted block is checked first on a data request
    - If the prediction was wrong, check the other entries
    - Speeds up the initial access of the cache in case the prediction is correct
- Pseudo-associative caches:
  - Implemented as a direct mapped cache with a ‘backup’ location
  - Only one address can hold the cache item
  - In case the item is not located in the cache a secondary address is also checked
    - Secondary address obtained by reverting the 1bit of the cache
  - Problem: have to avoid that all data-items are located in a secondary location after a while

Pipelined cache access

- Pipelined cache access:
  - Reduce latency of first level cache access by pipelining
  - E.g. Intel Core-i7 process takes 4 clock cycles for L1 instruction cache access
- Increases number of pipeline stages
  - Penalty on branch mispredictions
  - Larger number of cycles between issuing Load and Execute step
Non-blocking caches

- Allow data cache to continue to supply cache hits during a cache miss
- Reduces effective miss penalty
- Further optimization: allow overlap of multiple misses
  - Only useful if the memory system can service multiple misses.
- Increases complexity of cache controller

Multi-bank caches (I)

- Example: Costs of a cache miss
  - 4 clock cycles to send the address to main memory
  - 56 clock cycles to access a word
  - 4 clock cycles to send a word of data
- Assuming a cache block consists of 4 words, a loading a cache block takes $4 \times (4 + 56 + 4)$ cycles

- Increase width of main memory bus
  - assuming the memory width is 2 words, the costs of a cache miss go down to $2 \times (4+54+4)$ cycles
Multi-bank caches (II)

- Interleaved main memory
  - Try to take advantage of the fact that reading/writing multiple words use multiple chips (=banks)
  - Banks are typically one word wide
  - Often using sequential interleaving:
    - Bank 0 has all blocks with address \( \% 4 = 0 \)
    - Bank 1 has all blocks with address \( \% 3 = 1 \)
    - Etc.

- If your cache consists of four banks:
  - costs of a cache miss are \( 4 + 54 + (4 \times 4) \) cycles

Critical word first and early restart

- In case of a cache-miss, an entire cache-block has to be loaded from memory
- Idea: don’t wait until the entire cache-block has been loaded, focus on the required data item
  - Critical word first:
    - ask for the required data item
    - Forward the data item to the processor
    - Fill up the rest of the cache block afterwards
  - Early restart:
    - Fetch words of a cache block in sequential order
    - Forward the requested data item to the processor as soon as available
    - Fill up the rest of the cache block afterwards
Merging write buffers

- Check in the write buffer whether multiple entries can be merged to a single one

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Compiler optimizations

- Re-organize source code such that caches are used more efficiently
- Two examples:
  - Loop interchange
    /* original code jumps through memory and cache*/
    for (j=0; j<N; j++)
      for (i=0; i<M; i++) x[i][j] = 2*x[i][j];

    /*optimized code */
    for (i=0; i<M; i++)
      for (j=0; j<N; j++) x[i][j] = 2*x[i][j];

  - Blocked matrix-matrix multiply (see introductory lecture 1)
Hardware prefetching

- Prefetch items before they are requested by the processor
- E.g. for instruction cache: processor fetches two blocks on a miss
  - Requested block is placed in the instruction cache
  - Prefetched block is placed in an instruction stream buffer
  - In case of a request to the prefetched block, the original cache request is cancelled by the hardware and the block is read from the stream buffer
- Can be utilized for data caches as well
- Downside: wasting memory bandwidth if block is not required

Compiler-controlled prefetch

- Register prefetch: load value into register
- Cache prefetch: load data into cache
- *Faulty and non-faulting* prefetches: a non-faulting prefetch turns into a no-op in case of a virtual address fault/protection violation
Compiler-controlled prefetch (II)

• Example:
  
  ```c
  for (i=0; i<3; i++ ) {
    for (j=0; j<100; j++ ) {
      a[i][j] = b[j][0] * b[j+1][0];
    }
  }
  ```

• Assumptions:
  - Each element of `a`, `b` are 8 byte (double precision fp)
  - 8 KB direct mapped cache with 16 byte blocks
    • Each block can hold 2 consecutive values of either `a` or `b`

Compiler controlled prefetching (III)

• No. of cache misses for `a`:
  - Since a block contains `a[i][j]`, `a[i][j+1]` every even value of `j` will lead to a cache miss and every odd value of `j` will be a cache hit
    → 3 x 100/2 = 150 cache misses because of `a`

• No. of cache misses because of `b`:
  - `b` does not benefit from spatial locality
  - Each element of `b` can be used 3 times (for `i=0,1,2`)
  - Ignoring cache conflicts, this leads to a
    • Cache miss for `b[0][0]` for `i=0`
    • cache miss for every `b[j+1][0]` when `i=0`
    → 1 + 100 = 101 cache misses
Compiler controlled prefetching (IV)

- Assuming that a cache miss takes 7 clock cycles
- Assuming that we are dealing with non-faulting prefetches
- Does not prevent cache misses for the first 7 iterations

```c
for (j=0; j<100; j++) {
    prefetch b[j+7][0];
    prefetch a[0][j+7];
    a[0][j] = b[j][0] * b[j+1][0];
}
for (i=1; i<3; i++) {
    for (j=0; j<100; j++) {
        prefetch a[i][j+7];
        a[i][j] = b[j][0] * b[j+1][0];
    }
}
```

Compiler controlled prefetching (V)

- New number of cache misses:
  - 7 misses for b[0][0], b[1][0], ... in the first loop
  - 7/2 = 4 misses for a[0][0], a[0][2], ... in the first loop
  - 7/2 = 4 misses for a[1][0], a[1][2], ... in the second loop
  - 7/2 = 4 misses for a[2][0], a[2][2], ... in the second loop
  - total number of misses: 19
  - reducing the number of cache misses by 232!
- A write hint could tell the cache: I need the elements of a in the cache for speeding up the writes. But since I do not read the element, don’t bother to load the data really into the cache - it will be overwritten anyway!