COSC 6385
Computer Architecture
- Memory Hierarchies (II)
Edgar Gabriel
Spring 2010

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 instructions
- Miss penalty: time/cycles required for making a data item in the cache
Processor Performance

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

\[ \text{Memory stall cycles/instruction} = \text{Misses/instruction} \times (\text{Total miss latency} - \text{overlapped miss latency}) \]

Reducing cache miss penalty

- Five techniques
  - Multilevel caches
  - Critical word first and early restart
  - Giving priority to read misses over writes
  - Merging write buffer
  - Victim caches
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 \(_{L1}\) + Miss rate \(_{L1}\) x Miss penalty \(_{L1}\)

Miss penalty \(_{L1}\) = Hit time \(_{L2}\) + Miss rate \(_{L2}\) x Miss penalty \(_{L2}\)

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 \(_{L1}\) x Miss rate \(_{L2}\)

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: 100 clock cycles
- Miss penalty \( \text{direct mapped L2} = 10 + 0.25 \times 100 = 35 \) clock cycles
- Miss penalty \( \text{2-way assoc. L2} = 10.1 + 0.2 \times 100 = 30.1 \) clock cycles

Multilevel caches (III)

- Multilevel inclusion: 2\textsuperscript{nd} level cache includes all data items which are in the 1\textsuperscript{st} level cache
  - Applied if size of 2\textsuperscript{nd} level cache >> size of 1\textsuperscript{st} level cache
- Multilevel exclusion: Data of L1 cache is never in the L2 cache
  - Applied if size of 2\textsuperscript{nd} level cache only slightly bigger than size of 1\textsuperscript{st} level cache
  - Cache miss in L1 often leads to a swap of an L1 block with an L2 block
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 load, 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 normal order
    - Forward the requested data item to the processor as soon as available
    - Fill up the rest of the cache block afterwards

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
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>124</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Victim caches

- Question: how often is a cache block which has just been replaced by another cache block required soon after that again?
- Victim cache: fully associative cache between the ‘real’ cache and the memory keeping blocks that have been discarded from the cache
  - Typically very small
Reducing miss rate

- Three categories 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
    -> increase cache size
  - **Conflict Misses**: cache block has to be discarded because of block replacement strategy
    -> increase cache size and/or associativity.

Reducing miss rate (II)

- Five techniques to reduce the miss rate
  - Larger cache block size
  - Larger caches
  - Higher associativity
  - Way prediction and pseudo-associative caches
  - Compiler optimization
Larger block size

- Larger block size will reduce compulsory misses
- Assuming that the cache size is constant, a larger block size also 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
Way Prediction and Pseudo-associative caches

- 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
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)

Reducing Cache Miss penalty/rate via Parallelism

- Nonblocking caches
- Hardware prefetch of Instructions and Data
- Compiler controlled prefetching
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

Hardware prefetching

- Prefetch items before they are requested by the processor
  - E.g. processor prefetches two blocks on a miss, assuming that the block following the missing data item will also be requested
  - 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
- 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+1 ) {
      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
  \[ \rightarrow 3 \times 100/2 = 150 \text{ cache misses because of } a \]
- 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 \( i=0 \)
  \[ \rightarrow 1 + 100 = 101 \text{ cache misses} \]

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!

Reducing Hit time

- Small and simple caches
- Avoiding address translation
- Pipelined cache access
- Trace caches
Small and simple caches

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

Avoiding address translation

- Translation of virtual address to physical address required to access memory/caches
- Can a cache use virtual addresses?
  - In practice: no
    - Page protection has to be enforced, even if cache would use virtual addresses
    - Cache has to be flushed for every process switch
      - 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 would solve parts of this problem
    - Operating systems might use different virtual addresses for the same physical address (aliases)
      - Could lead to duplicate copies of the same data in the cache
Pipelined cache access and trace caches

- Pipelined cache access:
  - Reduce latency of first level cache access by pipelining

- Trace caches:
  - Find a dynamic sequence of instructions to load into a instruction cache block
  - The cache gets ‘dynamic traces of the executed instructions’ of the CPU and is not bound the spatial locality in the memory
  - Works well on taken branches
  - Used e.g. by Pentium 4

Memory organization

- 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
Improving memory bandwidth

- Wider main memory bus
  - E.g. for the previous example: assuming the memory width is 2 words, the costs of a cache miss go down to 2 \( \times (4+54+4) \) cycles

- 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
  - Memory interleaving tries to utilize all chips in the system
  - E.g. for the previous example: costs of a cache miss are \( 4+54+(4\times4) \) cycles

- Independent memory banks
  - Allow independent access to different memory chips/banks
  - Multiple memory controllers allow banks to operate independently
    - Might required separate data buses, address lines
  - Usually required by non-blocking caches