COSC 6385
Computer Architecture
- Memory Hierarchies (II)

Edgar Gabriel
Fall 2007

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
Split vs. unified cache

- Assume two machines:
  - Machine 1: 16KB instruction cache + 16 KB data cache
  - Machine 2: 32KB unified cache
- Assume for both machines:
  - 36% of instructions are memory references/data transfers
  - 74% of memory references are instruction references
  - Misses per 1000 instructions:
    - 16 KB instruction cache: 3.82
    - 16 KB data cache: 40.9
    - 32 KB unified cache: 43.3
- Hit time:
  - 1 clock cycle for machine 1
  - 1 additional clock cycle for machine 2 (structural hazard)
- Miss penalty: 100 clock cycles

Split vs. unified cache (II)

- Questions:
  1. Which architecture has a lower miss-rate?
  2. What is the average memory access time for both machines?

Miss-rate per instruction can be calculated as:

\[
\text{Miss rate} = \frac{\text{Misses}}{\text{1000 Instructions}} \times \frac{1000}{\text{Memory access}} \times \frac{\text{Instruction}}{1000}
\]
Split vs. unified cache (III)

- Machine 1:
  - since every instruction access requires exactly one memory access:
    Miss rate \(16\) KB instruction \(= (3.82/1000)/1.0 = 0.00382 \approx 0.004\)
  - Since 36% of the instructions are data transfer:
    Miss rate \(16\) KB data \(= (40.9/1000)/0.36 = 0.114\)
  - Overall miss rate: since 74% of memory access are instructions references:
    Miss rate split cache \(= (0.74 \times 0.004) + (0.26 \times 0.114) = 0.0324\)

Split vs. unified cache (IV)

- Machine 2:
  - Unified cache needs to account for the instruction fetch and data access
    Miss rate \(32\) KB unified \(= (43.4/1000)/(1 + 0.36) = 0.0318\)

  → Answer to question 1: the 2\(\text{nd}\) architecture has a lower miss rate
Split vs. unified cache (V)

- Average memory access time (AMAT):

\[
AMAT = \%\text{instructions} \times (\text{Hit time} + \text{Instruction Miss rate} \times \text{Miss penalty}) + \%
\text{data} \times (\text{Hit time} + \text{Data Miss rate} \times \text{Miss penalty})
\]

- Machine 1:
  \[AMAT_1 = 0.74 \times (1 + 0.004 \times 100) + 0.26 \times (1 + 0.114 \times 100) = 4.24\]

- Machine 2:
  \[AMAT_2 = 0.74 \times (1 + 0.0318 \times 100) + 0.26 \times (1 + 1 + 0.0318 \times 100) = 4.44\]

→ Answer to question 2: the 1st machine has a lower average memory access time.

Direct mapped vs. set associative

- Assumptions:
  - CPI without cache misses ( = perfect cache) : 2.0
  - No. of memory references per instruction: 1.5
  - Cache size: 64 KB
    - Machine 1: direct mapped cache
      - Clock cycle time: 1ns
      - Miss rate: 1.4%
    - Machine 2: 2-way set associative
      - Clock cycle time: 1.25 ns
      - Miss rate: 1.0%
  - Cache miss penalty: 75ns
  - Hit time: 1 clock cycle
Direct mapped vs. set associative (II)

- Average memory access time (AMAT):
  \[ \text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \]

\[ \text{AMAT}_1 = 1.0 + (0.014 \times 75) = 2.05 \text{ ns} \]
\[ \text{AMAT}_2 = 1.25 + (0.010 \times 75) = 2.0 \text{ ns} \]

→ avg. memory access time better for 2-way set associative cache

Direct mapped vs. set associative (III)

- CPU performance:
  \[ \text{CPU time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Misses/instruction} \times \text{Miss penalty}) \times \text{Clock cycle time} \]
  \[ = \text{IC} \times [(\text{CPI}_{\text{exec}} \times \text{Clock cycle time}) + (\text{Miss rate} \times \text{memory access/instruction} \times \text{Miss penalty} \times \text{Clock cycle time})] \]

\[ \text{CPU time}_1 = \text{IC} \times (2 \times 1.0 + (1.5 \times 0.014 \times 75 \times 1.0)) = 3.58 \text{ IC} \]
\[ \text{CPU time}_2 = \text{IC} \times (2 \times 1.25 + (1.5 \times 0.01 \times 75 \times 1.25)) = 3.63 \text{ IC} \]

→ Direct mapped cache leads to better CPU time
Processor Performance

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

- 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 \( L_1 \) + 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: 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: 2nd level cache includes all data items which are in the 1st level cache
  - Applied if size of 2nd level cache >> size of 1st level cache
- Multilevel exclusion: Data of L1 cache is never in the L2 cache
  - Applied if size of 2nd level cache only slightly bigger than size of 1st 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
Higher Associativity

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];

Compiler optimizations(II)

- Blocking
  - Example: Matrix-multiply of two dense matrices
  - “Trivial” code

    for ( i=0; i<dim; i++ ) {
      for ( j=0; j<dim; j++ ) {
        for ( k=0; k<dim; k++) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }
Matrix-multiply

- Performance of the trivial implementation on an 2.2 GHz AMD Opteron with 2 GB main memory 1 MB 2\textsuperscript{nd} level cache

<table>
<thead>
<tr>
<th>Matrix dimension</th>
<th>Execution time [sec]</th>
<th>Performance [MFLOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x256</td>
<td>0.118</td>
<td>284</td>
</tr>
<tr>
<td>512x512</td>
<td>2.05</td>
<td>130</td>
</tr>
</tbody>
</table>

Blocked code

```c
for ( i=0; i<dim; i+=block ) {
    for ( j=0; j<dim; j+=block ) {
        for ( k=0; k<dim; k+=block) {
            for (ii=i; ii<(i+block); ii++) {
                for (jj=j; jj<(j+block); jj++) {
                    for (kk=k; kk<(k+block); kk++) {
                        c[ii][jj] += a[ii][kk] * b[kk][jj];
                    }
                }
            }
        }
    }
}
```
### Performance of the blocked code

<table>
<thead>
<tr>
<th>Matrix dimension</th>
<th>block</th>
<th>Execution time [sec]</th>
<th>Performance [MFLOPS]</th>
<th>“trivial” [MFLOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x256</td>
<td>4</td>
<td>0.065</td>
<td>513</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.046</td>
<td>726</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.51</td>
<td>657</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.043</td>
<td>777</td>
<td>284</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.049</td>
<td>677</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.113</td>
<td>296</td>
<td></td>
</tr>
<tr>
<td>512x512</td>
<td>4</td>
<td>0.686</td>
<td>391</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.422</td>
<td>635</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.447</td>
<td>599</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.501</td>
<td>535</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.00</td>
<td>266</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.994</td>
<td>269</td>
<td></td>
</tr>
</tbody>
</table>