CSCE 5610: Computer Architecture

Review: Cache Design
Reading vs Writing
Write-through with write buffers

Address | Data to be written
---------|---------------------
1000     | Value

CPU is not stalled while writing to L2, L3,… unless write buffer is full

Write-back
If dirty lines are evicted, need to write dirty lines to lower memory before bringing new data

Replacement policies: Random, Least Recently Used, FIFO

Average memory access time (AMAT) = (time on hit) + (miss rate)*(miss penalty)

Converting MPKI (misses per 1000 instructions) into miss rate
need to know # accesses per instruction
1 for instruction fetch + (fraction of L/S instructions)

Split cache vs Unified Cache (separate Instruction cache and Data cache)

Unified cache → need to stall instruction fetch when the cache is used for data access

Instructions have spatial localities and data is likely to have temporal localities
Large cache blocks are good for spatial localities, but not for temporal localities
Smaller cache blocks are good for temporal localities
smaller blocks → more cache lines → accommodate more independent data items

Separate Array cache (or stream cache) and scalar data caches
6 basic techniques for improving

1. Larger blocks size – bring more data when you fetch a block (page B-27) due to locality of data/instructions, you are prefetching data that will be needed

   **miss penalty increases** – need to get more data on miss

<table>
<thead>
<tr>
<th>Cache size</th>
<th>Miss penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k</td>
<td>82 cycles</td>
</tr>
<tr>
<td>32</td>
<td>84 cycles</td>
</tr>
<tr>
<td>64</td>
<td>88 cycles</td>
</tr>
<tr>
<td>256</td>
<td>92 cycles</td>
</tr>
</tbody>
</table>

   Larger blocks reduce cold/compulsory misses

   But increases miss penalty

   Miss penalties

   - 82 cycles for 16 bytes
   - 84 cycles for 32 bytes
   - 88 cycles for 64 bytes

   More data to bring to cache
2. Larger caches are better (reduce capacity misses)
   Diminishing returns
   Non-uniformity of cache accesses

Some sets are accessed many times more than other sets
   – doubling cache size may not change the lines/set that receive most accesses

Consider problem B.1 (b) on page B.60
You are trying to appreciate how important the principle of locality is in justifying the use of a cache memory, so you experiment with a computer having an L1 data cache and a main memory (you exclusively focus on data accesses). The latencies (in CPU cycles) of the different kinds of accesses are as follows: cache hit, 1 cycle; cache miss, 110 cycles; main memory access with cache disabled, 105 cycles.

Note: Some people (even the authors of our textbook) sometime uses this equation
\[
AMAT = (hit rate) \times (hit time) + ((miss rate) \times (miss penalty))
\]
\[
= (0.97) \times 1 + (0.03 \times 110) = 4.27
\]

What is the logical reasoning behind these two formulas?

a). When you run a program with an overall miss rate of 3%, what will the average memory access time (in CPU cycles) be?

Soln. Average access time = (hit time) + (miss rate * miss time) = 1 + 0.03 * 110 = 4.3 cycles.

Note: Some people (even the authors of our textbook) sometime uses this equation
AMAT = (hit rate) \times (hit time) + ((miss rate) \times (miss penalty)) = (0.97) \times 1 + (0.03 \times 110) = 4.27

What is the logical reasoning behind these two formulas?
CSCE 5610: Computer Architecture

b. [10] < B.1 > Next, you run a program specifically designed to produce completely random data addresses with no locality. Toward that end, you use an array of size 1 GB (all of which fits in the main memory). Accesses to random elements of this array are continuously made (using a uniform random number generator to generate the elements indices). If your data cache size is 64 KB, what will the average memory access time be?

Soln. If we assume random accesses, we can compute the miss rate as following

\[
\frac{1 \text{GB}}{64 \text{KB}} = 2^{30}/2^{16} = 2^{14}
\]

\[
\text{AMAT} = 1 + (\text{miss rate}) \times 110 = 111
\]

(if you use the second equation, hit rate is almost zero

\[
\text{AMAT} = (0) \times 1 + 1 \times 110 = 110
\]

In this case, the cache is almost useless

c). If you compare the result obtained in part (b) with the main memory access time when the cache is disabled, what can you conclude about the role of the principle of locality in justifying the use of cache memory?

Soln: If we do not use cache, all accesses go to main memory and the AMAT = 105 cycles

CSCE 5610: Computer Architecture

3. Higher Associativity will reduce conflict misses
   But higher associativity will increase “cycle time” – Why?
   This may not be as significant issue now.

Example from Page B-19
   Assume that the CPI with a perfect cache is 1.0, the clock cycle time is 0.35 ns,
   there are 1.4 memory references per instruction, the size of both caches is 128 Kib, and both
   have a block size of 64 bytes.

   One cache is direct mapped and the other is two-way set associative.

   The processor clock cycle time must be stretched 1.35 times for 2-way
   cache miss penalty is 65 ns for either cache organization.

   Let us compute this slightly differently (from Tuesday)

   We normally translate miss penalty into clock cycles
   For 1-way, 65ns miss penalty = 65/0.35 = 185.7 rounded off to 186 cycles
   For 2-way, 65ns miss penalty = 65/(0.35*1.35) = 137.56 rounded off to 138 cycles
CSCE 5610: Computer Architecture

Now let us compute performance in terms of CPI

\[ \text{CPI}_{1\text{-way}} = (\text{cycles on hit}) + (\text{cycles on miss}) \]
\[ = 1 + (\text{miss rate}) \times (\text{miss penalty cycles}) \times (\text{accesses per instruction}) \]
\[ = 1 + 0.021 \times 186 \times 1.4 = 1 + 5.468 = 6.468 \text{ cycles} \]

\[ \text{CPI}_{2\text{-way}} = (\text{cycles on hit}) + (\text{cycles on miss}) \]
\[ = 1 + (\text{miss rate}) \times (\text{miss penalty cycles}) \times (\text{accesses per instruction}) \]
\[ = 1 + 0.019 \times 138 \times 1.4 = 1 + 3.67 = 4.67 \text{ cycles} \]

Execution Time\(_{1\text{-way}} = IC \times \text{CPI} \times (\text{cycle time}) = IC \times 6.468 \times 0.35 = IC \times 2.2638 \text{ ns} \)

Execution Time\(_{2\text{-way}} = IC \times \text{CPI} \times (\text{cycle time}) = IC \times 4.67 \times 0.35 \times 1.35 = IC \times 2.206 \text{ ns} \)

Note that either way of computing execution times results in very similar results

CSCE 5610: Computer Architecture

Another Example B-29. Here we are given how higher associativities increase cycle time (reduce clock frequencies)

- 2-way cycle time = 1.36 1-way cycle time
- 4-way cycle time = 1.44 1-way cycle time
- 8-way cycle time = 1.52 1-way cycle time

Assume hit time is one cycle (and cycle time is different for different associativities)
Miss penalty is 25 cycles based on 1-way clock time (and does not change with associativity)
Find Average memory access times for different cache sizes and associativities using miss rates from Figure B-28 (we have seen this table that shows the 3 C’s)

\[ \text{AMAT}_{1\text{-way}} = [1 + (\text{miss rate}) \times 25] \times (\text{cycle-time}_{1\text{-way}}) \]
\[ \text{AMAT}_{2\text{-way}} = [1.36 + (\text{miss rate}) \times 25] \times (\text{cycle-time}_{1\text{-way}}) \]
\[ \text{AMAT}_{4\text{-way}} = [1.44 + (\text{miss rate}) \times 25] \times (\text{cycle-time}_{1\text{-way}}) \]
\[ \text{AMAT}_{8\text{-way}} = [1.52 + (\text{miss rate}) \times 25] \times (\text{cycle-time}_{1\text{-way}}) \]

We will use the total miss rate (combined Cold + Capacity + Conflict) for each different cache size
Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example.

The Bold numbers indicate when a lower associativity results in lower AMAT. For caches larger than 16KB, the slower clock (and slower hit time) outweighs the reduction in miss rates.

<table>
<thead>
<tr>
<th>Cache size (KIB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.44</td>
<td>3.25</td>
<td>3.22</td>
<td>3.28</td>
</tr>
<tr>
<td>8</td>
<td>2.69</td>
<td>2.58</td>
<td>2.55</td>
<td>2.62</td>
</tr>
<tr>
<td>16</td>
<td>2.23</td>
<td>2.40</td>
<td>2.46</td>
<td>2.53</td>
</tr>
<tr>
<td>32</td>
<td>2.06</td>
<td>2.30</td>
<td>2.37</td>
<td>2.45</td>
</tr>
<tr>
<td>64</td>
<td>1.92</td>
<td>2.14</td>
<td>2.18</td>
<td>2.25</td>
</tr>
<tr>
<td>128</td>
<td>1.52</td>
<td>1.84</td>
<td>1.92</td>
<td>2.00</td>
</tr>
<tr>
<td>256</td>
<td>1.32</td>
<td>1.66</td>
<td>1.74</td>
<td>1.82</td>
</tr>
<tr>
<td>512</td>
<td>1.20</td>
<td>1.55</td>
<td>1.59</td>
<td>1.66</td>
</tr>
</tbody>
</table>

Instead of increasing associativity, we can explore other techniques:

**Victim Cache (or victim buffer).**
- When a data item is evicted from a set on a conflict, place it in a temporary cache (victim cache).
- Victim cache is a small fully associative cache.

*at UNT we investigated (and are investigating) interesting ideas for victim caches*

**Class Project possibility**

Intelligently use victim cache – keep victims only if they are likely to be needed again.

**Miss Cache**
- When there is a conflict, place the new data in a temporary location.
- Miss cache.
- Is this better than Victim cache? Why or why not?

If the new item is used only once or twice, you should not evict more heavily used data.
CSCE 5610: Computer Architecture

Pseudo Associative cache  (also known as column associative)

First treat the cache as direct mapped. If a miss, then treat the cache as a 2-way Associative
On miss, invert the most significant bit of the index

How do we find the access times when using Pseudo Associative Caches?
Remember we view the cache first as direct mapped
If a miss, then use an extra cycle to check in the "second location"

Average access time\[\text{pseudo}\] = (hit time)\[\text{pseudo}\] + (miss rate)\[\text{pseudo}\] * (miss penalty)\[\text{pseudo}\]

(hit time)\[\text{pseudo}\] = (hit time)\[1\text{-way}\] + 2*(hit rate in alternate cache line)
(hit rate in alternate cache line) = (hit rate)\[2\text{-way}\] – (hit rate)\[1\text{-way}\]

\[\text{hit rate in alternate cache line} = [1-(miss rate)\[2\text{-way}\]] - [1-(miss rate)\[1\text{-way}\]] = (miss rate)\[1\text{-way}\] – (miss rate)\[2\text{-way}\]\n
(hit time)\[\text{pseudo}\] = (hit time)\[1\text{-way}\] + 2* [(miss rate)\[1\text{-way}\] – (miss rate)\[2\text{-way}\]]

Consider an example
Let us assume that 128KB cache
(hit rate)\[1\text{-way}\] = 0.021; (hit rate)\[2\text{-way}\] = 0.019

Let us assume that the miss penalty for 1-way cache is 50 cycles

Average access time\[\text{pseudo}\] = (hit time)\[1\text{-way}\] + 2* [(miss rate)\[1\text{-way}\] – (miss rate)\[2\text{-way}\]]

What if we only have 1-way
Average memory access\[1\text{-way}\] = 1 + (miss rate)\[1\text{-way}\]*miss penalty = 1+(0.021)*50 = 2.05
Skewed Associative Caches

View caches as associative
Use different index bits for each way
Can be viewed as using a different hashing function
normal index = modulo (number of cache set)

<table>
<thead>
<tr>
<th>TAG</th>
<th>INDEX</th>
<th>BYTE OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Address of one set element</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG</th>
<th>INDEX</th>
<th>TAG</th>
<th>BYTE OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address of second set element</td>
</tr>
</tbody>
</table>

Note that what we are trying to do is improve the performance of set-associative cache.

Column associative cache (Pseudo associative) tries to reduce the time on “hit” – higher associativity slows clock

Skewed associative cache tries to minimize miss rates

*Victim cache increases “associativity” only for frequently conflicting data
But can only accommodate very few victims*
CSCE 5610: Computer Architecture

Can we think of other ways to improve associative caches?
- Programmable associativity
- Higher associativity to only data that is conflicting often
  victim caches are "fully associative" and are costly

Caches addressing can be viewed somewhat like “hashing”

How do we deal with conflicts in hash tables?
- Normally we use a linked list to store all values hashing to the same entry
- Any other ideas to handle conflicts?

<table>
<thead>
<tr>
<th>Tag</th>
<th>V</th>
<th>Partner Index</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heavily accessed cache lines will be provided with alternate locations as indicated by “partner index”

Pier’s adaptive cache uses two tables
- Set-reference History Table (SHT) – tracks heavily used cache lines
- Out-of-position directory (OUT) – tracks alternate locations

SHT keeps track of heavily accessed sets.
- If an address maps to the heavily accessed set, and not found
  it may have been relocated.

OUT provides the set number for relocated data items

Performance depends on the sizes of SHT and OUT
- 3/8 and 4/16 of the number of lines in cache are good

This method incurs additional cycles if not found in primary location


CSCE 5610: Computer Architecture

Comparing Reduction in Miss Rates – compared conventional indexing method

<table>
<thead>
<tr>
<th>SPEC 2006 Benchmarks</th>
<th>Adaptive_Cache</th>
<th>B_Cache</th>
<th>Column_assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>bzip2</td>
<td>15</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>calculix</td>
<td>12</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>gromacs</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>hmmer</td>
<td>15</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>libquantum</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>milc</td>
<td>2</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>namd</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>sjeng</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Average</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>

Comparing Reduction in Miss Rates
– compared conventional indexing method

Remember cache indexes can be viewed as “hash” functions
we are using modulo “number of sets” as hash function

Can we design indexing using Perfect hash functions?

Different approaches have been studied to find optimal indexing
(or hashing an address into cache)

- Givargis quality bits
- X-Or some tag bits with index bits
- Add an Odd multiple of tag to index
- Use prime modulo


Also, we compared side by side all these techniques

CSCE 5610: Computer Architecture

Givargis quality bits
Collect address traces, and identify unique addresses
For each address bit, define quality
\[ Q_i = \min(Z_i, O_i)/\max(Z_i, O_i) \]

Correlation between 2 bits i and j
\[ C_{ij} = \min(E_{ij}, D_{ij})/\max(E_{ij}, D_{ij}) \]

Find bits with high quality and low (pairwise) correlations

Does not account for frequency of accesses to addresses

X-OR: New index = (conventional index) XOR (Tag bits)

Prime Modulo: Set address = (Address) mod (prime number)

Odd Multiplier: New Index = (conventional index) + (Tag)*(odd number)

Can think of other methods
in general find a perfect hash function

CSCE 5610 Sept. 20, 2018 21

CSCE 5610: Computer Architecture

Comparing Reduction in Miss Rates – compared conventional indexing method

- XOR
- Odd Multiplier
- Prime Modulo
- Givargis
- Givargis XOR

SPEC 2006 Benchmarks

CSCE 5610 Sept. 20, 2018 22
Use different indexes for different data types
or different programs/cores/threads

Selecting different bits for index is easy to implement in hardware
Use a mask register

Different decoders for different data types or programs or threads
Somewhat like Skewed Associative Caches