Memories (3)

Hung-Wei Tseng
Recap: von Neumann Architecture
Recap: Performance gap between Processor/Memory
Recap: Memory Hierarchy

- Processor
  - Core
    - Registers
  - SRAM
  - DRAM
- Storage

Memory Sizes:
- L1$: fastest, 32 or 64 words, tens of ns
- L2$: larger, a few ns
- L3$: tens of ns
- Storage: TBs, tens of us

Timing:
- Processor Core: < 1ns
- SRAM: tens of ns
- DRAM: hundreds of ns
- Storage: tens of us
Recap: Way-associative cache

memory address: \(0x0\)

Set

hit?

\(0\times0\) 8 2 4

set block

index offset

data tag

data tag

0b0000100000100100

V D tag data
1 1 0x29 IIJJKKLLMMNNOOPP
1 1 0xDE QQRRSTTUUVVWWXX
1 0 0x10 YYZZAABBCCDDEEFF
0 1 0x8A AABBCCDDEEGGFFHH
1 1 0x60 IIJJKKLLMMNNOOPP
1 1 0x70 QQRRSTTUUVVWWXX
0 1 0x10 QQRRSTTUUVVWWXX
0 1 0x11 YYZZAABBCCDDEEFF

Set

V D tag data
1 1 0x00 AABBCCDDEEGGFFHH
1 1 0x10 IIJJKKLLMMNNOOPP
1 0 0xA1 QQRRSTTUUVVWWXX
0 1 0x10 YYZZAABBCCDDEEFF
1 1 0x31 AABBCCDDEEGGFFHH
1 1 0x45 IIJJKKLLMMNNOOPP
1 1 0x41 QQRRSTTUUVVWWXX
0 1 0x68 YYZZAABBCCDDEEFF

=?

hit?

=?
Recap: What happens when we read data

- Processor sends load request to L1-$
  - if hit
    - return data
  - if miss
    - Select a victim block
      - If the target “set” is not full — select an empty/invalidated block as the victim block
      - If the target “set” is full — select a victim block using some policy
        - LRU is preferred — to exploit temporal locality!
      - If the victim block is “dirty” & “valid”
        - Write back the block to lower-level memory hierarchy
        - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
      - If write-back or fetching causes any miss, repeat the same process
Recap: What happens when we write data

- Processor sends load request to L1-$
  - if hit
    - return data — set DIRTY
  - if miss
    - Select a victim block
      - If the target “set” is not full — select an empty/invalidated block as the victim block
      - If the target “set” is full — select a victim block using some policy
        - LRU is preferred — to exploit temporal locality!
    - If the victim block is “dirty” & “valid”
      - Write back the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
    - Present the write “ONLY” in L1 and set DIRTY
Recap: \( C = \text{ABS} \)

- **C**: Capacity in data arrays
- **A**: Way-Associativity — how many blocks within a set
  - N-way: \( N \) blocks in a set, \( A = N \)
  - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
  - How many bytes in a block
- **S**: Number of Sets:
  - A set contains blocks sharing the same index
  - 1 for fully associative cache
  - Number of bits in block offset — \( \log(B) \)
  - Number of bits in set index: \( \log(S) \)
  - Tag bits: \( \text{address} \_\text{length} - \log(S) - \log(B) \)
    - \( \text{address} \_\text{length} \) is 32 bits for 32-bit machine
Recap: AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

<table>
<thead>
<tr>
<th>address in hex</th>
<th>tag</th>
<th>address in binary</th>
<th>offset</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0] 0x20000</td>
<td>0b10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[0] 0x30000</td>
<td>0b11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[0] 0x10000</td>
<td>0b01 0000 0000 0000 0000</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[1] 0x20004</td>
<td>0b10 0000 0000 0000 0100</td>
<td>0x4</td>
<td>0</td>
<td>miss, evict 0x6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[1] 0x30004</td>
<td>0b11 0000 0000 0000 0100</td>
<td>0x6</td>
<td>0</td>
<td>miss, evict 0x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[1] 0x10004</td>
<td>0b01 0000 0000 0000 0100</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C = ABS
64KB = 2 * 64 * S
S = 512
offset = log(64) = 6 bits
index = log(512) = 9 bits
tag = the rest bits

100% miss rate!
Recap: intel Core i7

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

<table>
<thead>
<tr>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000</td>
<td>0x20</td>
<td>0</td>
<td>miss</td>
</tr>
<tr>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
<td>miss</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
<td>miss</td>
</tr>
<tr>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
<td>hit</td>
</tr>
</tbody>
</table>
```
| 0x2003C   | 0x20  | 0     | hit   |
| 0x3003C   | 0x30  | 0     | hit   |
| 0x1003C   | 0x10  | 0     | hit   |
| 0x20040   | 0x20  | 1     | miss  |
| 0x30040   | 0x30  | 1     | miss  |
| 0x1003C   | 0x10  | 1     | miss  |
```

C = ABS
32KB = 8 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = 64 - lg(64) - lg(64) = 52 bits

\[
32 \times 3 \div (512 \times 3) = 1 \div 16 = 6.25\% (93.75\% \text{ hit rate!})
\]
Outline

• Cause of cache misses
• Remedies to cache misses
Cause of cache misses
3Cs of misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
  - \( C = A B S \)
  - \( S = 256/(16*1) = 16 \)
  - \( \lg(16) = 4 : 4 \) bits are used for the index
  - \( \lg(16) = 4 : 4 \) bits are used for the byte offset
  - The tag is 48 - (4 + 4) = 40 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0000

\[
\text{tag} \quad \text{index} \quad \text{offset}
\]
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tag**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b11</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
<td>0100</td>
</tr>
</tbody>
</table>

- **Compulsory Miss**: 0b10 0000 0000
- **Hit!**: 0b10 0000 1000
- **Compulsory Miss**: 0b10 0001 0000
- **Hit!**: 0b10 0001 0100
- **Compulsory Miss**: 0b11 0001 0000
- **Hit!**: 0b10 0000 0000
- **Hit!**: 0b10 0000 1000
- **Hit!**: 0b10 0001 0000
- **Conflict Miss**: 0b10 0001 0100
- **Hit!**: 0b10 0001 0100
Simulate a 2-way cache

- Consider a 2-way cache with 256 bytes total capacity, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000, 0b1000010100, 0b1100010000

  - \( C = A B S \)
  - \( S = \frac{256}{(16*2)} = 8 \)
  - 8 = \(2^3\) : 3 bits are used for the index
  - 16 = \(2^4\) : 4 bits are used for the byte offset
  - The tag is 32 - (3 + 4) = 25 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0001 0000
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b100</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0b100</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>1000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0100</td>
</tr>
<tr>
<td>0b11</td>
<td>0001</td>
</tr>
<tr>
<td>0b10</td>
<td>0000</td>
</tr>
<tr>
<td>0b10</td>
<td>1000</td>
</tr>
<tr>
<td>0b10</td>
<td>0001</td>
</tr>
</tbody>
</table>

- compulsory miss
- hit!
D-L1 Cache configuration of AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

How many of the cache misses are **conflict** misses?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%
### AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*)
```

<table>
<thead>
<tr>
<th>Address in Hex</th>
<th>Tag</th>
<th>Address in Binary</th>
<th>Offset</th>
<th>Tag</th>
<th>Index</th>
<th>Hit? Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000</td>
<td>0x4</td>
<td>0b10 0000 0000 0000 0000</td>
<td>0</td>
<td>0x4</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x30000</td>
<td>0x6</td>
<td>0b11 0000 0000 0000 0000</td>
<td>0</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x2</td>
<td>0b01 0000 0000 0000 0000</td>
<td>0</td>
<td>0x2</td>
<td>0</td>
<td>compulsory miss, evict</td>
</tr>
<tr>
<td>0x20004</td>
<td>0x4</td>
<td>0b10 0000 0000 0000 0100</td>
<td>0</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>0x30004</td>
<td>0x6</td>
<td>0b11 0000 0000 0000 0100</td>
<td>0</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>0x10004</td>
<td>0x2</td>
<td>0b01 0000 0000 0000 0100</td>
<td>0</td>
<td>0x2</td>
<td>0</td>
<td>conflict miss, evict 0x4</td>
</tr>
<tr>
<td>0x2003C</td>
<td>0x4</td>
<td>0b10 0000 0000 0111 1100</td>
<td>0</td>
<td>0x4</td>
<td>0</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>0x3003C</td>
<td>0x6</td>
<td>0b11 0000 0000 0111 1100</td>
<td>0</td>
<td>0x6</td>
<td>0</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>0x1003C</td>
<td>0x2</td>
<td>0b01 0000 0000 0111 1100</td>
<td>0</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>0x20040</td>
<td>0x4</td>
<td>0b10 0000 0000 0100 0000</td>
<td>0</td>
<td>0x4</td>
<td>1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x30040</td>
<td>0x6</td>
<td>0b11 0000 0000 0100 0000</td>
<td>0</td>
<td>0x6</td>
<td>1</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x10040</td>
<td>0x2</td>
<td>0b01 0000 0000 0100 0000</td>
<td>0</td>
<td>0x2</td>
<td>1</td>
<td>compulsory miss, evict</td>
</tr>
</tbody>
</table>
D-L1 Cache configuration of AMD Phenom II

Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

How many of the cache misses are **conflict** misses?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%

C = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits
• D-L1 Cache configuration of intel Core i7 processor
  • Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

How many of the cache misses are compulsory misses?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%
D-L1 Cache configuration of intel Core i7 processor

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

How many of the cache misses are **compulsory** misses?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}

<table>
<thead>
<tr>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x20</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>hit</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>hit</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x20</td>
<td>compulsory miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x30</td>
<td>hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10</td>
<td>hit</td>
</tr>
</tbody>
</table>

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)
D-L1 Cache configuration of intel Core i7 processor

- Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

How many of the cache misses are **compulsory** misses?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%
Improving 3Cs
Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity

How many of the following are correct?

1. Increasing associativity can reduce conflict misses
2. Increasing associativity can reduce hit time
3. Increasing block size can increase the miss penalty
4. Increasing block size can reduce compulsory misses

A. 0
B. 1
C. 2
D. 3
E. 4
Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity

How many of the following are correct?

1. Increasing associativity can reduce conflict misses
2. Increasing associativity can reduce hit time
3. Increasing block size can increase the miss penalty
4. Increasing block size can reduce compulsory misses

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
- How many of the following are correct?
  1. Increasing associativity can reduce conflict misses
  2. Increasing associativity can reduce hit time
  3. Increasing block size can increase the miss penalty
  4. Increasing block size can reduce compulsory misses

A. 0
B. 1
C. 2
D. 3
E. 4
Improvement of 3Cs

- 3Cs and A, B, C of caches
  - Compulsory miss
    - Increase B: increase miss penalty (more data must be fetched from lower hierarchy)
  - Capacity miss
    - Increase C: increase cost, access time, power
  - Conflict miss
    - Increase A: increase access time and power
- Or modify the memory access pattern of your program!
Programming and memory performance
Data layout
Memory addressing/alignment

- Almost every popular ISA architecture uses “byte-addressing” to access memory locations.
- Instructions generally work faster when the given memory address is aligned.
  - Aligned — if an instruction accesses an object of size $n$ at address $X$, the access is aligned if $X \mod n = 0$.
  - Some architecture/processor does not support aligned access at all.
  - Therefore, compilers only allocate objects on “aligned” address.
The result of `sizeof(struct student)`

- Consider the following data structure:
  ```c
  struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
  };
  ```

  What’s the output of `printf("%lu\n", sizeof(struct student))`?
  
  A. 20  
  B. 28  
  C. 32  
  D. 36  
  E. 40
Array of structures or structure of arrays

Array of objects

```c
struct grades
{
    int id;
    double *homework;
    double average;
};
```

Object of arrays

```c
struct grades
{
    int *id;
    double **homework;
    double *average;
};
```

<table>
<thead>
<tr>
<th>ID</th>
<th>*homework</th>
<th>average</th>
<th>ID</th>
<th>*homework</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

average of each homework

```c
for(i=0;i<homework_items; i++)
{
    gradesheet[total_number_students].homework[i] = 0.0;
    for(j=0;j<total_number_students;j++)
        gradesheet[total_number_students].homework[i] += gradesheet[j].homework[i];
    gradesheet[total_number_students].homework[i] /= (double)total_number_students;
}
```

```c
for(i = 0; i < homework_items; i++)
{
    gradesheet.homework[i][total_number_students] = 0.0;
    for(j = 0; j < total_number_students; j++)
    {
        gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j];
    }
    gradesheet.homework[i][total_number_students] /= total_number_students;
}
What data structure is performing better

<table>
<thead>
<tr>
<th></th>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct grades</td>
<td>{ int id;</td>
<td>{ int *id;</td>
</tr>
<tr>
<td></td>
<td>double *homework;</td>
<td>double **homework;</td>
</tr>
<tr>
<td></td>
<td>double average;</td>
<td>double *average;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>}</td>
</tr>
</tbody>
</table>

average of each homework

```c
for(i=0;i<homework_items; i++)
{ 
    gradesheet[total_number_students].homework[i] = 0.0;
    for(j=0;j<total_number_students;j++)
        gradesheet[total_number_students].homework[i] += gradesheet[j].homework[i];
    gradesheet[total_number_students].homework[i] /= (double)total_number_students;
}
```

```c
for(i = 0; i < homework_items; i++)
{ 
    gradesheet.homework[i][total_number_students] = 0.0;
    for(j = 0; j < total_number_students; j++)
        gradesheet.homework[i][j] += gradesheet.homework[i][j];
    gradesheet.homework[i][total_number_students] /= total_number_students;
}
```

• Considering your workload would like to calculate the average score of one of the homework for all students, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays
What data structure is performing better

<table>
<thead>
<tr>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
</table>
| **struct** grades
  {
    int id;
    double *homework;
    double average;
  } | **struct** grades
  {
    int *id;
    double **homework;
    double *average;
  } |

average of each homework

```c
for(i=0;i<homework_items; i++)
  {
    gradesheet[total_number_students].homework[i] = 0.0;
    for(j=0;j<total_number_students; j++)
      gradesheet[total_number_students].homework[i] += gradesheet[j].homework[i];
    gradesheet[total_number_students].homework[i] /= (double)total_number_students;
  }
```

```c
for(i = 0;i < homework_items; i++)
  {
    gradesheet.homework[i][total_number_students] = 0.0;
    for(j = 0; j <total_number_students; j++)
      gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j];
    gradesheet.homework[i][total_number_students] /= total_number_students;
  }
```

- Considering your workload would like to calculate the average score of one of the homework for all students, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays
**What data structure is performing better**

<table>
<thead>
<tr>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
</table>
| `struct grades {
  int id;
  double *homework;
  double average;
};` | `struct grades {
  int *id;
  double **homework;
  double *average;
};` |
| average of each homework
  
  ```c
  for(i=0; i<homework_items; i++)
  {
    gradesheet[total_number_students].homework[i] = 0.0;
    for(j=0; j<total_number_students; j++)
      gradesheet[total_number_students].homework[i] += gradesheet[j].homework[i];
    gradesheet[total_number_students].homework[i] /= (double)total_number_students;
  }
  ```
| average of each homework
  
  ```c
  for(i = 0; i < homework_items; i++)
  {
    gradesheet.homework[i][total_number_students] = 0.0;
    for(j = 0; j < total_number_students; j++)
      gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j];
    gradesheet.homework[i][total_number_students] /= total_number_students;
  }
  ```

- Considering your workload would like to calculate the average score of one of the homework for all students, which data structure would deliver better performance?
  
  What if we want to calculate average scores for each student?

  A. Array of objects

  **B. Object of arrays**
Column-store or row-store

• If you’re designing an in-memory database system, will you be using

<table>
<thead>
<tr>
<th>RowId</th>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>Jones</td>
<td>Bob</td>
<td>55000</td>
</tr>
</tbody>
</table>

• column-store — stores data tables column by column

10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;

if the most frequently used query looks like —

select Lastname, Firstname from table

• row-store — stores data tables row by row

001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
Loop interchange/fission/fusion
Demo — programmer & performance

A

\[
\begin{align*}
\text{for}(i = 0; & \ i < \ \text{ARRAY\_SIZE}; \ i++) \\
\quad & \ \{
\text{for}(j = 0; & \ j < \ \text{ARRAY\_SIZE}; \ j++) \\
\quad & \quad \{
\quad & \quad \quad \text{c}[i][j] = a[i][j]+b[i][j];
\quad & \quad \}
\quad & \}
\end{align*}
\]

B

\[
\begin{align*}
\text{for}(j = 0; & \ j < \ \text{ARRAY\_SIZE}; \ j++) \\
\quad & \ \{
\text{for}(i = 0; & \ i < \ \text{ARRAY\_SIZE}; \ i++) \\
\quad & \quad \{
\quad & \quad \quad \text{c}[i][j] = a[i][j]+b[i][j];
\quad & \quad \}
\quad & \}
\end{align*}
\]

\begin{align*}
\textcolor{red}{O(n^2)} & & \text{Complexity} & & \textcolor{red}{O(n^2)} \\
\text{Same} & & \text{Instruction Count?} & & \text{Same} \\
\text{Same} & & \text{Clock Rate} & & \text{Same} \\
\text{Better} & & \text{CPI} & & \text{Worse}
\end{align*}
AMD Phenom II

• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
  c[i] = a[i] + b[i];
  //load a, b, and then store to c
}
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = 64 - lg(512) - lg(64) = 49 bits
What if the code look like this?

• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
  c[i] = a[i]; //load a and then store to c
for(i = 0; i < 512; i++)
  c[i] += b[i]; //load b, load c, add, and then store to c
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
D-L1 Cache configuration of AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i]; //load a and then store to c
for(i = 0; i < 512; i++)
    c[i] += b[i]; //load b, load c, add, and then store to c
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
What if the code look like this?

• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i]; //load a and then store to c
for(i = 0; i < 512; i++)
    c[i] += b[i]; //load b, load c, add, and then store to c
```

What’s the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%
Loop Fusion

/* Before */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
       d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {
       a[i][j] = 1/b[i][j] * c[i][j];
       d[i][j] = a[i][j] + c[i][j];
   }

2 misses per access to a & c vs. one miss per access
Blocking
Case study: Matrix Multiplication

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Algorithm class tells you it’s $O(n^3)$

If $n=1024$, it takes about 1 sec

How long is it take when $n=2048$?
Matrix Multiplication

• If each dimension of your matrix is 2048
  • Each row takes 2048*8 bytes = 16KB
  • The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  • You can only hold at most 2 rows/columns of each matrix!
  • You need the same row when j increase!

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Very likely a miss if array is large
Block algorithm for matrix multiplication

• Discover the cache miss rate
  • valgrind --tool=cachegrind cmd
    • cachegrind is a tool profiling the cache performance
• Performance counter
  • Intel® Performance Counter Monitor http://www.intel.com/software/pcm/
Block algorithm for matrix multiplication

for (i = 0; i < ARRAY_SIZE; i++) {
    for (j = 0; j < ARRAY_SIZE; j++) {
        for (k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}

for (i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for (j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for (k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for (ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for (jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for (kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}

You only need to hold these sub-matrices in your cache.
Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an Intel Core i7)

- **Compulsory miss**
- **Capacity miss**
- **Conflict miss**
- **Capacity & conflict miss**
- **Compulsory & conflict miss**

**Naive**

```c
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        c[i][j] += a[i][k]*b[k][j];
    }
}
```

**Block**

```c
for(i = 0; i < ARRAY_SIZE; i+= (ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+= (ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+= (ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```
Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an Intel Core i7)

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Naive

```
for(i = 0; i < ARRAY_SIZE; i++) {
    for(j = 0; j < ARRAY_SIZE; j++) {
        for(k = 0; k < ARRAY_SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}
```
Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block algorithm help to remove? (assuming an intel Core i7)

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss
Matrix Transpose

// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            c[ii][jj] += a[ii][kk]*b[kk][jj];
        }
    }
}

// Compute on b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
        }
    }
}

for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}
What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+= (ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+= (ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+= (ARRAY_SIZE/n)) {
            for(ii = i; ii < i + (ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j + (ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k + (ARRAY_SIZE/n); kk++)
                        c[ii][jj] += a[ii][kk] * b_t[jj][kk];
        }
    }
}
```

```
// Compute on b_t
for(i = 0; i < ARRAY_SIZE; i+= (ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+= (ARRAY_SIZE/n)) {
        for(k = 0; k < ARRAY_SIZE; k+= (ARRAY_SIZE/n)) {
            for(ii = i; ii < i + (ARRAY_SIZE/n); ii++)
                for(jj = j; jj < j + (ARRAY_SIZE/n); jj++)
                    for(kk = k; kk < k + (ARRAY_SIZE/n); kk++)
                        // Compute on b_t
                        c[ii][jj] += a[ii][kk] * b_t[jj][kk];
        }
    }
}
```

Block + Transpose
What kind(s) of misses can matrix transpose remove?

• By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss
What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss
Announcements

• Reading quiz due tomorrow
• Assignment #4 is up on the website