Memory Hierarchy (III)

Hung-Wei Tseng
Recap: The memory gap problem

The memory gap problem is caused by the disparity in access times between the CPU and DRAM-based main memory. The access time of DRAM is around 50ns, which is 100x to the cycle time of a 2GHz processor!

SRAM, despite being as fast as the processor, is much more expensive. The memory bottleneck is crucial for performance optimization.

---

### Memory Bottleneck

<table>
<thead>
<tr>
<th>Memory technology</th>
<th>Typical access time</th>
<th>$ per GiB in 2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM semiconductor memory</td>
<td>0.5–2.5 ns</td>
<td>$500–$1000</td>
</tr>
<tr>
<td>DRAM semiconductor memory</td>
<td>50–70 ns</td>
<td>$10–$20</td>
</tr>
<tr>
<td>Flash semiconductor memory</td>
<td>5,000–50,000 ns</td>
<td>$0.75–$1.00</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>5,000,000–20,000,000 ns</td>
<td>$0.05–$0.10</td>
</tr>
</tbody>
</table>

---

**The access time of DRAM is around 50ns**

**100x to the cycle time of a 2GHz processor!**

SRAM is as fast as the processor, but **$$$$**
The memory hierarchy

- **Fastest, Most Expensive**
  - **CPU**
    - 32* 64-bit registers
  - **Cache**
    - L1: 16KB-64KB
    - L2: 128KB-512KB
    - L3: Several MBs
  - **Main Memory**
    - Several GBs
  - **Secondary Storage**
    - 500+ GB

- **Access time**
  - < 1ns
  - < 1ns ~ 20 ns
  - 100ns
  - 10,000,000ns
Recap: Accessing the cache

Hit: The data was found in the cache
Miss: The data was not found in the cache

Offset: The position of the requesting word in a cache block

Hit? miss?

Valid dirty tag index offset

1 0 1000 0001 0000 1000 0000

memory address: 0x8 0 0 0 0 0 1 5 8

Offset:

The position of the requesting word in a cache block
Recap: How many bits in each field?

- lg(number of sets)
- lg(block size)

Diagram:

- tag
- index
- offset
- valid
- dirty
- data

hit?
C = ABS

- **C**: Capacity in data arrays
- **A**: Way-Associativity
  - 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 associate cache
Corollary of $C = \text{ABS}$

- offset bits: $\lg(B)$
- index bits: $\lg(S)$
- tag bits: $\text{address\_length} - \lg(S) - \lg(B)$
  - $\text{address\_length}$ is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \% S = \text{set index}$
### 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>Address in Binary</th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
<th>Hit? Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x4</td>
<td>0</td>
<td>0x0000000000000000</td>
<td>miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>0</td>
<td>0x0000000000000000</td>
<td>miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x2</td>
<td>0</td>
<td>0x0000000000000000</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x4</td>
<td>0</td>
<td>0x0000000000000100</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x6</td>
<td>0</td>
<td>0x0000000000000100</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x2</td>
<td>0</td>
<td>0x0000000000000100</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x4</td>
<td>0</td>
<td>0x0000000000011100</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x6</td>
<td>0</td>
<td>0x0000000000011100</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x2</td>
<td>0</td>
<td>0x0000000000011100</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>0x4</td>
<td>1</td>
<td>0x0000000001000000</td>
<td>miss</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>0x6</td>
<td>1</td>
<td>0x0000000001000000</td>
<td>miss</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x10040</td>
<td>0x2</td>
<td>1</td>
<td>0x0000000001000000</td>
<td>miss, evict 0x4</td>
</tr>
</tbody>
</table>

---

**C = ABS**

- 64KB = 2 * 64 * S
- S = 512
- Offset = lg(64) = 6 bits
- Index = lg(512) = 9 bits
- Tag = the rest bits

**100% miss rate!**
```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>load a[0]</td>
<td>0x20000</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>0x20</td>
<td>1</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>0x30</td>
<td>1</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>1</td>
</tr>
</tbody>
</table>

$$32 \times 3 / (512 \times 3) = 1/16 = 6.25\% (93.75\% \text{ hit rate!})$$
Cache & Performance

- Application: 80% ALU, 20% Loads
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle
- L2 U-Cache miss rate: 20%, hit time: 10 cycles
- Main memory hit time: 100 cycles
- What’s the average CPI?

\[
CPI_{Average} = CPI_{base} + miss\_rate*\text{miss\_penalty} \\
= 1 + 100\% \ast (5\% \ast (10 + 20\% \ast (1 \ast 100))) + 20\% \ast (10\% \ast (1 \ast (10 + 20\% \ast (1 \ast 100)))) \\
= 3.1
\]
Outline

• Performance evaluation with cache
• Cause of misses
• Optimizations
Cache & Performance

• 5-stage MIPS processor.
  • Application: 80% ALU, 20% Loads and stores
  • L1 I-cache miss rate: 5%, hit time: 1 cycle
  • L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% of the replaced blocks are dirty.
  • L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% of the replaced blocks are dirty.
  • Main memory hit time: 100 cycles
  • What’s the average CPI?
    A. 0.77
    B. 2.6
    C. 3.37
    D. 4.1
    E. none of the above
Cache & Performance

- Application: 80% ALU, 20% Load/Store
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
- L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
- Main memory hit time: 100 cycles
- What’s the average CPI?

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss}_{\text{rate}} \times \text{miss}_{\text{penalty}}
\]

\[
= 1 + 100\% \times (5\% \times (10 + 20\% \times ((1 + 10\%) \times 100))) + 20\% \times (10\% \times (1 + 20\%) \times (10 + 20\% \times ((1 + 10\%) \times 100)))
\]

\[
= 3.368
\]
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
Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:

- 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- 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

Diagram:
- Tag: 25 bits
- Index: 3 bits
- Offset: 4 bits
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b100</td>
<td>1</td>
<td>0b100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10 0000 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b11 0001 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
</tbody>
</table>
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - `0b1000000000`, `0b1000001000`, `0b1000010000`, `0b1000010100`, `0b1100010000`

- \[ 16 = 2^4 \] : 4 bits are used for the index
- \[ 16 = 2^4 \] : 4 bits are used for the byte offset
- The tag is \[ 32 - (4 + 4) = 24 \] bits
- For example: `0b1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1000 0000`
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10 0000 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</td>
</tr>
<tr>
<td>0b11 0001 0000</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0b10 0000 0000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0000 1000</td>
<td>hit!</td>
</tr>
<tr>
<td>0b10 0001 0000</td>
<td>conflict miss</td>
</tr>
<tr>
<td>0b10 0001 0100</td>
<td>hit!</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.

Consider the following code

```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%
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>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x4</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x2</td>
<td>compulsory miss, evict 0x4</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x4</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x6</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x2</td>
<td>conflict miss, evict 0x4</td>
</tr>
</tbody>
</table>

... ... ... ... ... ...

<table>
<thead>
<tr>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x4</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x6</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x2</td>
<td>conflict miss, evict 0x4</td>
</tr>
</tbody>
</table>
• D-L1 Cache configuration of Core i7
  • Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, 32-bit OS?
  • Consider the following code?
    • 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. 33.33%
    C. 66.67%
    D. 68.75%
    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>...</td>
<td>...</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>hit</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>hit</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>hit</td>
</tr>
</tbody>
</table>

512/(64/4) = 32 compulsory misses each array
32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)
Improving 3Cs
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
## Conflict in direct-mapped cache

### If we have two frequently used cache blocks:

If they are usually used back-to-back, one will kick out the other all the time.
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!
Memory hierarchy and your code
#ifndef COL_MAJOR
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        for(j = 0; j < ARRAY_SIZE; j++)
        {
            c[i][j] = a[i][j]+b[i][j];
        }
    }
#else
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        for(i = 0; i < ARRAY_SIZE; i++)
        {
            c[i][j] = a[i][j]+b[i][j];
        }
    }
#endif
```c
#ifndef COL_MAJOR
for(i = 0; i < ARRAY_SIZE; i++)
{
    for(j = 0; j < ARRAY_SIZE; j++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
#else
for(j = 0; j < ARRAY_SIZE; j++)
{
    for(i = 0; i < ARRAY_SIZE; i++)
    {
        c[i][j] = a[i][j]+b[i][j];
    }
}
#endif
```
**Side-by-side comparison**

<table>
<thead>
<tr>
<th>i on row, j on col</th>
<th>i on col, j on row</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i = 0; i &lt; ARRAY_SIZE; i++) {</td>
<td></td>
</tr>
<tr>
<td>for(j = 0; j &lt; ARRAY_SIZE; j++) {</td>
<td></td>
</tr>
<tr>
<td>c[i][j] = a[i][j]+b[i][j];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>for(j = 0; j &lt; ARRAY_SIZE; j++) {</td>
<td></td>
</tr>
<tr>
<td>for(i = 0; i &lt; ARRAY_SIZE; i++) {</td>
<td></td>
</tr>
<tr>
<td>c[i][j] = a[i][j]+b[i][j];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- What type(s) of cache locality does (do) the left-hand side code better exploit than the right-hand side code?

  A. Spatial locality
  B. Temporal locality
  C. Both localities
  D. None of them
Demo revisited

- Why the left performs a lot better than t

<table>
<thead>
<tr>
<th>for(i = 0; i &lt; ARRAY_SIZE; i++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>for(j = 0; j &lt; ARRAY_SIZE; j++)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>c[i][j] = a[i][j] + b[i][j];</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for(j = 0; j &lt; ARRAY_SIZE; j++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
</tr>
<tr>
<td>for(i = 0; i &lt; ARRAY_SIZE; i++)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>c[i][j] = a[i][j] + b[i][j];</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Array_size = 1024, 0.048s
(5.25X faster)

Array_size = 1024, 0.252s

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

- for(j = 0; j < ARRAY_SIZE; j++)
  { for(i = 0; i < ARRAY_SIZE; i++)
    { c[i][j] = a[i][j] + b[i][j];
    }
  }
Considering your workload would like to calculate the average score of each homework, which data structure would deliver better performance?

A. Array of objects
B. Object of arrays

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

```
struct grades {
    int *id;
    double **homework;
    double *average;
};
```
## Array of structures or structure of arrays

<table>
<thead>
<tr>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct grades</td>
<td>struct grades</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int id;</td>
</tr>
<tr>
<td></td>
<td>double *homework;</td>
</tr>
<tr>
<td></td>
<td>double average;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>struct grades</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int *id;</td>
</tr>
<tr>
<td></td>
<td>double **homework;</td>
</tr>
<tr>
<td></td>
<td>double *average;</td>
</tr>
<tr>
<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[total_number_students][j];
    }
    gradesheet.homework[i][total_number_students] /= total_number_students;
}
```
What data structure is performing better

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

- Considering your workload would like to calculate the average score of each homework, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays

What if we want to calculate average scores for students?
If you’re designing an in-memory database system, will you be using:

- **column-store** — stores data tables column by column

<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>

- **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;

select Lastname, Firstname from table
Case study: Matrix Multiplication

- 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 $n=512$, it takes about 1 sec
How long is it take when $n=1024$?
Matrix Multiplication

- 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];
        }
    }
}
```

- If each dimension of your matrix is 1024
  - Each row takes 1024*8 bytes = 8KB
  - The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 4 rows/columns of each matrix!
  - You need the same row when j increase!

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

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

You only need to hold these sub-matrices in your cache.
Connecting architecture and software design now!

- Block Algorithm for Matrix Multiplication
- What value of n makes the block algorithm work the best?
- If the demo machine has an L1 D-cache with 64KB, 2-way, 64B blocks, array_size is 1024, each word is “8 bytes”

A. 16
B. 32
C. 64
D. 128
E. 256

```cpp
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];
        }
    }
}
```
Connecting architecture and software design now!

- Block Algorithm for Matrix Multiplication
- What value of \( n \) makes the block algorithm works the best?
- If the demo machine has an L1 D-cache with 64KB, 2-way, 64B blocks, array_size is 1024, each word is “8 bytes”

A. 16
B. 32
C. 64
D. 128
E. 256

```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];
        }
    }
}
```
Other cache optimizations
Different area of memory
• Different access patterns
  • instruction accesses have lots of spatial locality
  • instruction accesses are predictable to the extent that branches are predictable
  • data accesses are less predictable
• Instruction accesses may interfere with data accesses
• Avoiding structural hazards in the pipeline
• Writes to I-cache are rare
Revisit: Athlon 64

```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>load a[0]</td>
<td>0x20000</td>
<td>0x4</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x2</td>
<td>compulsory miss, evict 0x4</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x4</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x6</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x2</td>
<td>conflict miss, evict 0x4</td>
</tr>
</tbody>
</table>

100% miss rate due to a majority of conflict miss!
A small cache that captures the evicted blocks
- Can be built as fully associative since it’s small
- Consult when there is a miss
- Athlon has an 8-entry victim cache
- Reduce the miss penalty of conflict misses

**Victim cache**

![Diagram of victim cache](image_url)
Characteristic of memory accesses

```c
for(i = 0; i < 1000000; i++) {
    D[i] = rand();
}
```

![Diagram showing memory accesses and cache misses]
Prefetching

```
for(i = 0; i < 1000000; i++) {
    D[i] = rand();
    // prefetch D[i+8] if i % 8 == 0
}
```

![Diagram of CPU, L1, L2 cache with prefetching operations](image-url)
Prefetching

- Identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
  - Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction
- Hardware prefetch:
  - The processor can keep track the distance between misses. If there is a pattern, fetch miss_data_address+distance for a miss
- Software prefetching
  - Load data into $zero
  - Using prefetch instructions