Inside out of your computer memories (III)

Hung-Wei Tseng
Why memory hierarchy?

The access time of DDR3-1600 DRAM is around 50ns, 100x to the cycle time of a 2GHz processor!

SRAM is as fast as the processor, but $$$

CPU

main memory

```assembly
lw $t2, 0($a0)
add $t3, $t2, $a1
addi $a0, $a0, 4
subi $a1, $a1, 1
bne $a1, LOOP
lw $t2, 0($a0)
add $t3, $t2, $a1
```
Memory hierarchy

- CPU: Fastest, Most Expensive, Access time < 1ns
- Cache: Access time < 1ns ~ 20 ns
- Main Memory: Access time 50-60 ns
- Secondary Storage: Access time 10,000,000 ns
- Biggest
The structure of a cache

**Set:** cache blocks/lines sharing the same index. A cache is called N-way set associative cache if N blocks share the same set/index (this one is a 2-way set cache)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 0</td>
<td>1000 0001 0000 1000 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 1</td>
<td>1000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

**Block / Cacheline:** The basic unit of data storage in cache. Contains all data with the same tag/prefix and index in their memory addresses

**Tag:**
the high order address bits stored along with the data in a block to identify the actual address of the cache line.

**valid:** if the data is meaningful

**dirty:** if the block is modified
Accessing the cache

memory address: \(0x8\ 0\ 0\ 0\ 0\ 0\ 1\ 5\ 8\)

hit? miss?

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
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
- Offset bits: \(\log_2(B)\)
- Index bits: \(\log_2(S)\)
- Tag bits: address_length - \(\log_2(S)\) - \(\log_2(B)\)
What happens on a read?

- **Read hit**
  - hit time
- **Read miss?**
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
      - Data with the same “block address” will be fetch
    - Miss penalty
What happens on a write? (Write Allocate, write back)

- Write hit?
  - Update in-place
  - Set dirty bit (Write-Back Policy)

- Write miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back to lower memory hierarchy if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
**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>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>
</tbody>
</table>

\[
32*3/(512*3) = 1/16 = 6.25\% \text{ (93.75\% hit rate!)}
\]
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?

CPI_{Average} = CPI_{base} + miss\_rate \times miss\_penalty

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

\[
= 3.368
\]
Outline

- Cause of misses: 3Cs
- Improving 3Cs
- Other optimizations
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 2-way cache

- 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`, `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 0000 0001 0000`

![Diagram illustrating the tag, index, and offset parts of a memory address.](image)
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>0</td>
<td>0b110</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>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 1000 0000

- Tag: 0b1000 0000 0000 0000 0000 0000 0000
- Index: 0b1000
- Offset: 0b0000
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>index</th>
<th>data</th>
<th>tag</th>
<th>index</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
<td>0000</td>
<td>0b10</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></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>
<td></td>
</tr>
<tr>
<td>4</td>
<td></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>
<td></td>
</tr>
<tr>
<td>6</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- compulsory miss
- hit!
- conflict miss
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.
  • 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 “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>0</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x2</td>
<td>0</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x4</td>
<td>0</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x2</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>0x4</td>
<td>0</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x2</td>
<td>0</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>0x20000</td>
<td>0x20</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
<td>compulsory 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>0x10004</td>
<td>0x10</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>0x1003C</td>
<td>0x10</td>
<td>0</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?

- Increasing associativity can reduce conflict misses
- Increasing associativity can reduce hit time
- Increasing block size can increase the miss penalty
- Increasing block size can reduce compulsory misses

A. 0
B. 1
C. 2
D. 3
E. 4

Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You need to fetch more data for each miss

You bring more into the cache when a miss occurs
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!
Live demo: 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];
        }
    }
}
```

CSE101 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

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-matrixes in your cache
Block algorithm for matrix multiplication

• 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

```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
Split Data & Instruction caches

- 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!
Victim cache

- 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
Characteristic of memory accesses

\[
\text{for}(i = 0; i < 1000000; i++) \{ \\
\quad D[i] = \text{rand}(); \\
\}
\]
Prefetching

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

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
Write buffer

• Every write to lower memory will first write to a small SRAM buffer.
  • sw does not incur data hazards, but the pipeline has to stall if the write misses
  • The write buffer will continue writing data to lower-level memory
  • The processor/higher-level memory can response as soon as the data is written to write buffer.
• Help reduce miss penalty
• Write merge
  • Since application has locality, it’s highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.
Announcement

• Reading quizzes due this Thursday