Inside out of your computer memories (III)

Hung-Wei Tseng
Why memory hierarchy?

CPU

main memory

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 $$$
Memory hierarchy

- **CPU**
  - Fastest, Most Expensive
  - Access time: $< 1\text{ns}$

- **Main Memory**
  - Access time: $< 1\text{ns} - 20\text{ns}$

- **Secondary Storage**
  - Access time: $50-60\text{ns}$

- **Cache**
  - Access time: $10,000,000\text{ns}$
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>
<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>
<td></td>
<td></td>
<td>1 1</td>
<td>1000 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

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

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

**Valid:** if the data is meaningful

**Dirty:** if the block is modified
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

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

---

Hit? miss?
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: $\lg(B)$
- Index bits: $\lg(S)$
- Tag bits: address\_length - $\lg(S)$ - $\lg(B)$
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
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>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>
```

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

• Homework #4 due next Monday
• Reading quiz due next Monday
Today’s CSE141

• Causes of cache misses (cont.)
• Optimizing cache performance
• Evaluating application performance considering cache misses
• 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%
```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>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000</td>
<td>0x4</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x30000</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x2</td>
<td>0</td>
<td>compulsory miss, evict 0x4</td>
</tr>
<tr>
<td>0x20004</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>0x30004</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>0x10004</td>
<td>0x2</td>
<td>0</td>
<td>conflict miss, evict 0x4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x2003C</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>0x3003C</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>0x1003C</td>
<td>0x2</td>
<td>0</td>
<td>conflict miss, evict 0x4</td>
</tr>
</tbody>
</table>
intel Core i7

• 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?

  ```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. 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></th>
<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>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
<td>hit</td>
</tr>
<tr>
<td>store c[15]</td>
<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). May hurt miss rate since we have fewer sets/blocks
  • 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

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

- If each dimension of your matrix is 1024
  - Each row takes 1024*8 bytes = 8KB
  - The L1 cache 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!
Block algorithm for matrix multiplication

- Discover the cache miss rate
  - `valgrind --tool=cachegrind cmd`
  - cachegrind is a tool profiling the cache performance
- Performance counter
Block algorithm for 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];
        }
    }
}
```

You only need to hold these sub-matrixes in your cache.

```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];
        }
    }
}
```
Block algorithm for matrix multiplication

- 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];
        }
    }
}
```
Performance evaluation considering cache
Performance evaluation considering cache

- If the load/store instruction hits in L1 cache where the hit time is usually the same as a CPU cycle
  - The CPI of this instruction is the base CPI
- If the load/store instruction misses in L1, we need to access L2
  - The CPI of this instruction needs to include the cycles of accessing L2
- If the load/store instruction misses in both L1 and L2, we need to go to lower memory hierarchy (L3 or DRAM)
  - The CPI of this instruction needs to include the cycles of accessing L2, L3, DRAM
How to evaluate cache performance

- $CPI_{Average}$: the average CPI of a memory instruction

$$CPI_{Average} = CPI_{base} + \text{miss\_rate}_{L1} \times \text{miss\_penalty}_{L1}$$

$\text{miss\_penalty}_{L1} = CPI_{accessing\_L2} + \text{miss\_rate}_{L2} \times \text{miss\_penalty}_{L2}$

$\text{miss\_penalty}_{L2} = CPI_{accessing\_L3} + \text{miss\_rate}_{L3} \times \text{miss\_penalty}_{L3}$

$\text{miss\_penalty}_{L3} = CPI_{accessing\_DRAM} + \text{miss\_rate}_{DRAM} \times \text{miss\_penalty}_{DRAM}$

- If the problem is asking for **average memory access time**, transform the CPI values into/from time by multiplying with CPU cycle time!
Average memory access time

• Average Memory Access Time (AMAT) = Hit Time + Miss rate * Miss penalty
  • Miss penalty = AMAT of the lower memory hierarchy
  • AMAT = hit\_time_{L1} + miss\_rate_{L1} * AMAT_{L2}
    • AMAT_{L2} = hit\_time_{L2} + miss\_rate_{L2} * AMAT_{DRAM}
Cache & Performance

- 5-stage MIPS processor.
  - 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
  - Assume the program is read only (nothing dirty)
  - What’s the average CPI?

A. 1.1
B. 1.6
C. 2.1
D. 3.1
E. none of the above

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

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

\[ = 3.1 \]
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?

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

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

\[
= 3.1
\]
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?

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

\begin{align*}
    & = 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
\end{align*}
Other cache optimizations
Multi-layer caches

- Speed of L1 matches the processor
- Caches data/code as many as possible in L2/L3 to avoid DRAM accesses
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></th>
<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>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>0</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x2</td>
<td>0</td>
<td>compulsory miss, evict 0x4</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x4</td>
<td>0</td>
<td>conflict miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x6</td>
<td>0</td>
<td>conflict miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x2</td>
<td>0</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