Memories (2)

Hung-Wei Tseng
Recap: von Neumann Architecture

Processor

Program

Instructions
- 0f00bb27
- 509cbd23
- 00005d24
- 0000bd24
- 2ca422a0
- 130020e4
- 00003d24
- 2ca4e2b3

Data
- 00c2e800
- 00000008
- 00c2f000
- 00000008
- 00c2f800
- 00000008
- 00c30000
- 00000008

Memory

Storage

Instructions
- 0f00bb27
- 509cbd23
- 00005d24
- 0000bd24
- 2ca422a0
- 130020e4
- 00003d24
- 2ca4e2b3

Data
- 00c2e800
- 00000008
- 00c2f000
- 00000008
- 00c2f800
- 00000008
- 00c30000
- 00000008
Recap: Performance gap between Processor/Memory

![Graph showing the performance gap between Processor and Memory over time from 1980 to 2015.]
Recap: Memory Hierarchy

Processor

- Core
- Registers

SRAM

- L1 $
- L2 $
- L3 $

DRAM

Storage

- TBs
- larger

Fastest:

- < 1ns
- a few ns
- tens of ns

Larger:

- tens of ns
Locality

• Spatial locality — application tends to visit nearby stuffs in the memory
  • Code — the current instruction, and then PC + 4
  • Data — the current element in an array, then the next
• Temporal locality — application revisit the same thing again and again
  • Code — loops, frequently invoked functions
  • Data — the same data can be read/write many times

Most of time, your program is just visiting a very small amount of data/instructions within a given window
To capture “spatial” locality, $ fetch a “block”

```
<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x1000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x2000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x3000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x4000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x5000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x6000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x7000</td>
<td>AAAA</td>
</tr>
<tr>
<td>0x8000</td>
<td>AAAA</td>
</tr>
</tbody>
</table>
```

Load 0x000A

“Logically” partition memory space into “blocks”
Recap: How to tell who is there?

The complexity of search the matching tag—$O(n)$—will be slow if our cache size grows!

Can we search things faster? —hash table! $O(1)$

0x404 not found, go to lower-level memory
Recap: Way-associative cache

memory address:

0x0  8  2  4

set  block

index  offset

block offset

set index

tag

data

data
tag

memory address:

0b0000100000100100

V D  tag  data

1 1  0x29  IIJJKKLLMMNNOOPP
1 1  0xDE  QQRRSSTTUUVVWWXX
1 1  0x10  YYZZAABBCDDDEEFF
1 1  0x60  IIJJKKLLMMNNOOPP
1 1  0x70  QQRRSSTTUUVVWWXX
0 1  0x10  QQRRSSTTUUVVWWXX
0 1  0x11  YYZZAABBCDDDEEFF

V D  tag  data

1 1  0x00  AABBCCDDEEGGFFHH
1 1  0x10  AABBCCDDEEGGFFHH
1 0  0xA1  QQRRSSTTUUVVWWXX
0 1  0x10  YYZZAABBCDDDEEFF
1 1  0x31  AABBCCDDEEGGFFHH
1 1  0x45  IIJJKKLLMMNNOOPP
0 1  0x41  QQRRSSTTUUVVWWXX
0 1  0x68  YYZZAABBCDDDEEFF

Set

hit?

0x1

hit?

=?

=?
Outline

• Architecting the cache (cont.)
• Cache/CPU interactions
• Simulate the cache!
C = 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 **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
Corollary of C = ABS

- number of bits in block offset — \( \lg(B) \)
- number of bits in set index: \( \lg(S) \)
- tag bits: address_length - \( \lg(S) \) - \( \lg(B) \)
  - address_length is 32 bits for 32-bit machine
- \((\text{address} / \text{block}_\text{size}) \mod S\) = set index
L1 data (D-L1) cache configuration of AMD Phenom II
- Size 64KB, 2-way set associativity, 64B block
- Assume 64-bit memory address

Which of the following is correct?
A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above
AMD Phenom II

- L1 data (D-L1) cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above
L1 data (D-L1) cache configuration of AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block
- Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

\[ C = \text{ABS} \]
\[ 64\text{KB} = 2 \times 64 \times S \]
\[ S = 512 \]
\[ \text{offset} = \log(64) = 6 \text{ bits} \]
\[ \text{index} = \log(512) = 9 \text{ bits} \]
\[ \text{tag} = 64 - \log(512) - \log(64) = 49 \text{ bits} \]
• L1 data (D-L1) cache configuration of Core i7
  • Size 32KB, 8-way set associativity, 64B block
  • Assume 64-bit memory address
  • Which of the following is NOT correct?
    A. Tag is 52 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets
• L1 data (D-L1) cache configuration of Core i7
  • Size 32KB, 8-way set associativity, 64B block
  • Assume 64-bit memory address
  • Which of the following is NOT correct?
    A. Tag is 52 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets
intel Core i7

- L1 data (D-L1) cache configuration of Core i7
- Size 32KB, 8-way set associativity, 64B block
- Assume 64-bit memory address
- Which of the following is NOT correct?
  A. Tag is 52 bits
  B. Index is 6 bits
  C. Offset is 6 bits
  D. The cache has 128 sets

\[
\begin{align*}
S &= 64 \\
\text{offset} &= \log(64) = 6 \text{ bits} \\
\text{index} &= \log(64) = 6 \text{ bits} \\
\text{tag} &= 64 - \log(64) - \log(64) = 52 \text{ bits}
\end{align*}
\]

\[
C = \text{ABS}
\]

\[
32\text{KB} = 8 \times 64 \times S
\]

\[
S = 64
\]
Put everything all together: How cache interacts with CPU
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
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
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\textsubscript{Average}**: the average CPI of a memory instruction

\[
CPI_{\text{average}} = CPI_{\text{base}} + \text{miss\_rate}_{L1} \times \text{miss\_penalty}_{L1}
\]
\[
\text{miss\_penalty}_{L1} = CPI_{\text{accessing\_L2}} + \text{miss\_rate}_{L2} \times \text{miss\_penalty}_{L2}
\]
\[
\text{miss\_penalty}_{L2} = CPI_{\text{accessing\_L3}} + \text{miss\_rate}_{L3} \times \text{miss\_penalty}_{L3}
\]
\[
\text{miss\_penalty}_{L3} = CPI_{\text{accessing\_DRAM}} + \text{miss\_rate}_{\text{DRAM}} \times \text{miss\_penalty}_{\text{DRAM}}
\]

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

- 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?

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss\_rate} \times \text{miss\_penalty}
\]

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

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

\[
= 3.368
\]
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
Simulate the 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:

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

\[ C = A B S \]

- $S = \frac{256}{(16 \times 1)} = 16$

\[ \lg(16) = 4 : 4 \text{ bits are used for the index} \]

\[ \lg(16) = 4 : 4 \text{ bits are used for the byte offset} \]

- The tag is $48 - (4 + 4) = 40$ bits
- For example: $0b1000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1000 \ 0000$
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>0</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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

  - $C = A B S$
  - $S = 256 / (16 \times 2) = 8$
  - 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 0000 0001 0000

  

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

  - $S = \frac{256}{(16 \times 2)} = 8$
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>0b10</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0b10</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0b10</td>
</tr>
</tbody>
</table>

Tag Index:

- 0b10 0000 0000 0000 miss
- 0b10 0000 1000 0000 hit!
- 0b10 0001 0000 0000 miss
- 0b10 0001 0100 0000 hit!
- 0b11 0001 0000 0000 miss
- 0b10 0000 0000 0000 hit!
- 0b10 0000 1000 0000 hit!
- 0b10 0001 0000 0000 hit!
- 0b10 0001 0100 0000 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
}
```

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] + 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%
**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%

\[
\begin{align*}
C &= \text{ABS} \\
64\text{KB} &= 2 \times 64 \times S \\
S &= 512 \\
\text{offset} &= \log(64) = 6 \text{ bits} \\
\text{index} &= \log(512) = 9 \text{ bits} \\
\text{tag} &= 64 - \log(512) - \log(64) = 49 \text{ bits}
\end{align*}
\]
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>load a[0] 0x20000</th>
<th>0b10 0000 0000 0000 0000</th>
<th>tag</th>
<th>index</th>
<th>offset</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load b[0] 0x30000</td>
<td>0b11 0000 0000 0000 0000</td>
<td></td>
<td>0</td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>store c[0] 0x10000</td>
<td>0b01 0000 0000 0000 0000</td>
<td></td>
<td>0</td>
<td></td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[1] 0x20004</td>
<td>0b10 0000 0000 0000 0100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>load b[1] 0x30004</td>
<td>0b11 0000 0000 0000 0100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>store c[1] 0x10004</td>
<td>0b01 0000 0000 0000 0100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[15] 0x2003C</td>
<td>0b10 0000 0000 0011 1100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x6</td>
</tr>
<tr>
<td>load b[15] 0x3003C</td>
<td>0b11 0000 0000 0011 1100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x2</td>
</tr>
<tr>
<td>store c[15] 0x1003C</td>
<td>0b01 0000 0000 0011 1100</td>
<td></td>
<td>0</td>
<td>0</td>
<td>miss, evict 0x4</td>
</tr>
<tr>
<td>load a[16] 0x20040</td>
<td>0b10 0000 0000 0100 0000</td>
<td></td>
<td>0</td>
<td>1</td>
<td>miss</td>
</tr>
<tr>
<td>load b[16] 0x30040</td>
<td>0b11 0000 0000 0100 0000</td>
<td></td>
<td>0</td>
<td>1</td>
<td>miss</td>
</tr>
<tr>
<td>store c[16] 0x10040</td>
<td>0b01 0000 0000 0100 0000</td>
<td></td>
<td>0</td>
<td>1</td>
<td>miss, evict 0x4</td>
</tr>
</tbody>
</table>

- $C = \text{ABS}$
- $64KB = 2 \times 64 \times S$
- $S = 512$
- offset = lg(64) = 6 bits
- index = lg(512) = 9 bits
- tag = the rest bits

$100\%$ miss rate!
• 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%**
• 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
}
```

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 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
}
```

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 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
}
```

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

A. 6.25%
B. 56.25%
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>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*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
}
```

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

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

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