Memory hierarchy / Cache

Hung-Wei Tseng
Stored-program computer

instruction memory

120007a30: 0f00bb27 ldah gp,15(t12)
120007a34: 509cbd23 lda gp,-25520(gp)
120007a38: 00005d24 ldah t1,0(gp)
120007a3c: 0000bd24 ldah t4,0(gp)
120007a40: 2ca422a0 ld1 t0,-23508(t1)
120007a44: 130020e4 beq t0,120007a94
120007a48: 00003d24 ldah t0,0(gp)
120007a4c: 2ca4e2b3 stl zero,-23508(t1)
120007a50: 0004ff47 clr v0
120007a54: 28a4e5b3 stl zero,-23512(t4)
120007a58: 20a421a4 ldq t0,-23520(t0)
120007a5c: 0e0020e4 beq t0,120007a98
120007a60: 0204e147 mov t0,t1
120007a64: 0304ff47 clr t2
120007a68: 0500e0c3 br 120007a80
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 $$$
The memory hierarchy in “inside-out”

Islands (long-term memory)

Core Memory

Short-term Memory
Memory hierarchy

- **CPU**: Fastest, Most Expensive
  - Access time: < 1ns
- **Cache**: < 1ns ~ 20 ns
- **Main Memory**: 50-60ns
- **Secondary Storage**: 10,000,000ns

**Biggest**
Memory hierarchy

- **CPU**: Fastest, Most Expensive
- **Main Memory**: $\downarrow$
- **Secondary Storage**: $\downarrow$
  - **Cache**: Access time
    - $< 1\text{ ns}$
    - $< 1\text{ ns} \sim 20\text{ ns}$
    - $50-60\text{ ns}$
    - $10,000,000\text{ ns}$
Locality

- **Temporal Locality**
  - Referenced item tends to be referenced again soon.

- **Spatial Locality**
  - Items close by referenced item tends to be referenced soon.
    - example: consecutive instructions, arrays

- Let’s see how to build a memory hierarchy with “cache” that exploits “both” locality
Cache organization
Cache

- Like a cheat-sheet for the processor
- For a cheat-sheet, you may need to put
  - Most frequently asked concepts (temporal locality)
  - Problems, key points related to the frequently asked topics (spatial locality)
How do you make a cheatsheet?

- Go through your homework
- Write down the topic and content
- If running out of space: kick out the least recently used content

1. Performance equation
2. Amdahl’s law
3. MIPS
4. Power consumption
5. Performance equation 😊
6. Amdahl’s law 😊
7. MFLOPS

<table>
<thead>
<tr>
<th>Tag: the address prefix of data in the cacheline/block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance equation</td>
</tr>
<tr>
<td>Amdahl’s law</td>
</tr>
<tr>
<td>MFLOPS</td>
</tr>
<tr>
<td>Power consumption</td>
</tr>
</tbody>
</table>

Cacheline/block: data with the same prefix in their addresses
A simple cache: a block can go anywhere

- Assume each block contains 16B data
- A total of 4 blocks
- LRU
- Called fully-associative cache: block can go anywhere

- Too slow if the number of entries/blocks/cachelines is huge

<table>
<thead>
<tr>
<th></th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x4</td>
<td>0b000000100 0b00000000 - 0b00001111</td>
</tr>
<tr>
<td>2</td>
<td>0x48</td>
<td>0b01001000 0b01000000 - 0b01001111</td>
</tr>
<tr>
<td>3</td>
<td>0xC4</td>
<td>0b11000100 0b11110000 - 0b11111111</td>
</tr>
<tr>
<td>4</td>
<td>0xFC</td>
<td>0b11111100 0b11000000 - 0b11001111</td>
</tr>
<tr>
<td>5</td>
<td>0x12</td>
<td>0b00001100 0b00000000 - 0b00001111</td>
</tr>
<tr>
<td>6</td>
<td>0x44</td>
<td>0b01000100 0b01100000 - 0b01101111</td>
</tr>
<tr>
<td>7</td>
<td>0x68</td>
<td>0b01100100 0b11110000 - 0b11111111</td>
</tr>
</tbody>
</table>
Make cache a hash-like structure

- Performance is better than linear search
- Make cache a hardware hash table!
- The hash function takes memory addresses as inputs
- Each hash entry contains a block of data
  - caches operate on “blocks”
  - cache blocks are a power of 2 in size. Contains multiple words of memory
  - usually between 16B-128Bs
- Hit: requested data is in the table
- Miss: requested data is not in the table
- Tag: help us identify if it’s the requested data
Let cache exploit locality!

- Temporal locality
  - LRU-like polices keeps the most frequently used data
- Spatial locality
  - We bring a “block” of data each time
The structure of a cache

Set: cache blocks/lines sharing the same index

Tag: the high order address bits stored along with the data 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 and index in their address.

valid
tag
data

index offset

memory address: 1000 0000 0000 0000 0000 0001 0101 1000

valid
tag
data

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

22
How many bits in each field?

- **tag**
- **index**
- **offset**
- **valid**
- **data**

![Diagram showing how many bits in each field](image)

- \( \lg(\text{number of sets}) \)
- \( \lg(\text{block size}) \)

hit?
C = ABS

- **C**: Capacity
- **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 associative cache
Corollary of $C = \text{ABS}$

- offset bits: $\lg(B)$
- index bits: $\lg(S)$
- tag bits: $\text{address\_length} - \lg(S) - \lg(B)$
  - address\_length is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \mod S = \text{set index}$
Way-associative cache

Set: cache blocks/lines sharing the same index

memory address:
1000 0000 0000 0000 0000 0001 0101 1000

valid tag data

hit?

=?

valid tag data

hit?

=?
Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 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 0000 0001 0000
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>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0b100</td>
<td></td>
<td>1</td>
<td>0b110</td>
</tr>
</tbody>
</table>

**tag index**

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

30
Special case: a direct-mapped cache

- **Tag**: The high order address bits stored along with the data to identify the actual address of the cache line.
- **Offset**: The position of the requesting word in a cache block.
- **Block (cacheline)**: The basic unit of data storage in cache. Contains all data with the same tag and index in their address.
- **Hit**: The data was found in the cache.
- **Miss**: The data was not found in the cache.

### Example Memory Address
```
1000 0000 0000 0000 0000 0001 0101 1000
```

### Tag Index Offset Split
```
1000 0000 0000 0000 0000 0001 0101 1000
```

### Diagram Notes
- memory address: 
- valid
- tag
- index
- offset
- data
- hit? miss?
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1000010000
  - 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 1000 0000

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1000010000
  - 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 1000 0000

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1000010000
  - 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 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>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **tag**: 0b10
- **index**: 0000 0000

Miss

- **tag**: 0b10
- **index**: 0000 1000

Hit!

- **tag**: 0b10
- **index**: 0000 0000

Miss

- **tag**: 0b10
- **index**: 0001 0100

Hit!

- **tag**: 0b11
- **index**: 0001 0000

Miss

- **tag**: 0b10
- **index**: 0000 0000

Hit!

- **tag**: 0b10
- **index**: 0000 1000

Hit!

- **tag**: 0b10
- **index**: 0001 0000

Miss

- **tag**: 0b10
- **index**: 0001 0100

Hit!
Conflict in direct-mapped cache

If we have two frequently used cache blocks:

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 0001</td>
<td>0000</td>
<td>001</td>
<td>xx</td>
</tr>
<tr>
<td>1000 0000</td>
<td>0000</td>
<td>001</td>
<td>xx</td>
</tr>
</tbody>
</table>

If they are usually used back-to-back, one will kick out the other all the time.
Pros & cons of way associate caches

- Help alleviating the hash collision by having more blocks associating with each different index.
  - N-way associative: the block can be in N blocks of the cache
- Fully associative
  - The requested block can be anywhere in the cache
  - Or say N = the total number of cache blocks in the cache
- Performance issue: increasing associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
  - This limits associativity L1 caches to 2-8.
  - Larger, slower caches can be more associative
Way associativity and cache performance

![Graph showing the relationship between associativity and miss rate for different cache sizes. The y-axis represents the miss rate, ranging from 0% to 15%. The x-axis represents the associativity, ranging from One-way to Eight-way. The graph includes lines for cache sizes of 1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, and 128 KB, showing a decrease in miss rate as associativity increases.](image-url)
How cache works
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
What happens on a write? (Write Allocate, write through)

- Write hit?
  - Update in-place
  - Write to lower memory (Write-Through Policy)

- Write miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
What happens on a write? (No-Write Allocate, write through)

- Write hit?
  - Update in-place
  - Write to lower memory (Write-Through only)
    - write penalty (can be eliminated if there is a buffer)

- Write miss?
  - Write to the first lower memory hierarchy has the data
    - Penalty
What happens on a write?
(No-Write Allocate, write back)

• Write hit?
  • Update in-place

• Write miss?
  • Write to the first lower memory hierarchy has the data
  • Penalty

CPU

L1 $

write

L2 $
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
Evaluating cache performance
How to evaluate cache performance

• 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\text{Average}**: the average CPI of a memory instruction
  \[ \text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss\_rate}_{L1} \times \text{miss\_penalty}_{L1} \]

- **miss\_penalty\_L1**: CPI\text{accessing\_L2} + miss\_rate\_L2 \times miss\_penalty\_L2

- **miss\_penalty\_L2**: CPI\text{accessing\_L3} + miss\_rate\_L3 \times miss\_penalty\_L3

- **miss\_penalty\_L3**: CPI\text{accessing\_DRAM} + miss\_rate\_\text{DRAM} \times miss\_penalty\_\text{DRAM}

- If the problem (like those in your textbook) 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 = \( \text{hit	ext{\_}time}_{L1} + \text{miss	ext{\_}rate}_{L1} \times \text{AMAT}_{L2} \)
  - \( \text{AMAT}_{L2} = \text{hit	ext{\_}time}_{L2} + \text{miss	ext{\_}rate}_{L2} \times \text{AMAT}_{DRAM} \)
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
Cache simulation

• Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  • 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010

• $16 = 2^4$ : 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: $0x\underline{80000010}$
  • Index: $\underline{80}$
  • Tag: $00$
  • Offset: $0010$
Cache simulation

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8000000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8000000</td>
<td></td>
</tr>
</tbody>
</table>

0x80000000 miss: compulsory
0x80000008 hit!
0x80000010 miss: compulsory
0x80000018 hit!
0x300000010 miss: compulsory
0x80000000 hit!
0x80000008 hit!
0x80000010 miss: conflict
0x80000018 hit!
Cache simulation

• Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeat the following memory access sequence:
  • 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
  • $8 = 2^3$ : 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: $0b1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0000$
## Cache simulation

<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>0x1000000</td>
<td></td>
<td>1</td>
<td>0x600000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1000000</td>
<td></td>
<td>1</td>
<td>0x600000</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>

- **miss**: compulsory
- **hit**: 

- 0x80000000
- 0x80000008
- 0x80000010
- 0x80000018
- 0x30000010
- 0x80000000
- 0x80000008
- 0x80000010
- 0x80000018
• D-L1 Cache configuration of Athlon 64
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.
  • Cache performance for 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
      }
    • 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></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>load 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>load 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 conflict miss!
Improving 3Cs
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 $n=1024$?
Block algorithm for matrix multiplication

- Discover the cache miss rate
- valgrind --tool=cachegrind cmd
  - cachegrind is a tool profiling the cache performance
- Performance counter

```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];
        }
    }
}
```

```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
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>0</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x6</td>
<td>0</td>
</tr>
<tr>
<td>load 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>load c[1]</td>
<td>0x10004</td>
<td>0x2</td>
<td>0</td>
</tr>
</tbody>
</table>

100% miss rate due to 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 miss penalty of conflict misses
Prefetching

• Identify the access pattern and proactively fetch data before the application asks for.
  • Think about this code:
    
    ```
    for(i = 0; i < 1000000; i++) {
        sum += data[i];
    }
    ```
  
• 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

• Reduce compulsory misses
Write buffer

- Every write to lower memory will first write to a small SRAM buffer.
  - 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
- Help improve write through performance
- 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.
Q & A