Inside out of your computer memories

Hung-Wei Tseng
Outline

- Memory Hierarchy
  - The CPU-memory gap problems
  - Locality
- Cache organization
  - The structure of a cache
  - Hung-Wei’s secret formula of cache structures
The memory gap problem
Stored-program computer

```
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 ldl 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 $$$

```
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’s impact

• Considering that you have a processor with base CPI (including instruction fetch) of 1. The latency of DRAM is 100 cycles. If the application contains 20% memory operations, what’s the slowdown comparing with a perfect processor with CPI=1? (Choose the closest one)

A. 15%
B. 35%
C. 55%
D. 75%
E. 95%

average CPI = 1 + 0.2*100 = 21
slowdown = 1/21 = 4.76%
(95% performance drop)
The memory hierarchy in “inside out”

Islands (long-term memory)

Core Memory

Short-term Memory
Memory hierarchy

- CPU
- Main Memory
- Secondary Storage
- Cache

Access time:
- < 1ns
- < 1ns ~ 20 ns
- 50-60 ns
- 10,000,000 ns

Fastest, Most Expensive

$
Why building memory hierarchy would help?

- How many of the following descriptions about memory hierarchy/caching is/are correct?
  I. Existing programs can take advantage from memory hierarchy without any change.
  II. Memory hierarchy can capture frequently used data/instructions in faster/more expensive memory.
  III. Memory hierarchy can capture data/instructions that will be referenced in the near future in faster/more expensive memory.
  IV. Memory hierarchy exists because we cannot build large, fast memories.

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
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
Where in our code has locality?

- Which description about locality of arrays sum and A in the following code is the most accurate?

```c
for(i = 0; i< 100000; i++)
{
    sum[i%10] += A[i];
}
```

A. Access of A has temporal locality, sum has spatial locality
B. Both A and sum have temporal locality, and sum also has spatial locality
C. Access of A has spatial locality, sum has temporal locality
D. Both A and sum have spatial locality
E. Both A and sum have spatial locality, and sum also has temporal locality
### Demo revisited

- **Why the left performs a lot better than the right one?**

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

<table>
<thead>
<tr>
<th>Array_size = 1024, 0.048s</th>
<th>Array_size = 1024, 0.252s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5.25X faster)</td>
<td></td>
</tr>
</tbody>
</table>

A. The left one has fewer instruction counts
B. **The left one exploits spatial locality better**
C. The left one exploits temporal locality better
D. The left one exploits both spatial and temporal locality better
Demo revisited

```c
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];
    }
}
```

<table>
<thead>
<tr>
<th>Array_size = 1024, 0.048s</th>
<th>Array_size = 1024, 0.252s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5.25X faster)</td>
<td></td>
</tr>
</tbody>
</table>
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>Performance equation</th>
<th>ET=IC<em>CPI</em>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amdahl’s law</td>
<td>ET_after = ET_affected/Speedup + ET_unaffected</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>MIPS = No_FP_Ops/(ET*10^6)</td>
</tr>
<tr>
<td>Power consumption</td>
<td>P = aCV^2f</td>
</tr>
</tbody>
</table>

---

Tag: the address prefix of data in the cacheline/block

Cacheline/block: data with the same prefix in their addresses
Let’s make memory great again!

- **Spatial locality**
  - Each hash entry contains a block of data
  - We bring a “block” of data each time
  - Cache blocks are a power of 2 in size.
  - Usually between 16B-128Bs
  - Tag: help us identify what’s in the block

- **Temporal locality**
  - LRU-like polices keeps the most frequently used data
A simple cache: a block can go anywhere

- Assume each block contains 16B data
- A total of 4 blocks
- LRU

1. 0x4               0b00000100
2. 0x48             0b01001000
3. 0xC4             0b11000100
4. 0xFC             0b11111100
5. 0x12             0b00001100
6. 0x44             0b01000100
7. 0x68             0b01100100

- Too slow if the number of entries/blocks/cachelines is huge
Let’s make memory great again!

• Spatial locality
  • Each hash entry contains a block of data
  • We bring a “block” of data each time
  • Cache blocks are a power of 2 in size.
  • Usually between 16B-128Bs
  • Tag: help us identify what’s in the block

• Temporal locality
  • LRU-like polices keeps the most frequently used data

• Performance needs to be better than linear search
  • Make cache a hardware hash table!
  • The hash function takes memory addresses as inputs
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)

**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

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

<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 1000 0001 0000 1000 0000</td>
<td></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 1000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Accessing the cache

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

memory address:

<table>
<thead>
<tr>
<th>valid</th>
<th>dirty</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1000 0000 0000 0000 0000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1000 0001 0000 1000 0000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>valid</th>
<th>dirty</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

tag
index
offset

1000 0000 0000 0000 0000
0001 0101 1000
Accessing the cache

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

memory address:

\[
\begin{array}{cccccccc}
\text{tag} & \text{index} & \text{offset} \\
1000 & 0000 & 0000 & 0000 & 0000 & & & \\
0001 & 0101 & 1000 & & & & & \\
\end{array}
\]

valid | dirty | tag | data
---|---|---|---
--- | --- | 1 | 0 1000 0001 0000 1000 0000
--- | --- | 1 | 1 1000 0000 0000 0000 0000

memory address:

\[
\begin{array}{cccccccc}
\text{valid} & \text{dirty} & \text{tag} & \text{data} \\
\text{valid} & \text{dirty} & \text{tag} & \text{data} \\
1 & 0 & 1000 0001 0000 1000 0000 & \\
1 & 1 & 1000 0000 0000 0000 0000 & \\
\end{array}
\]
Accessing the cache

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

memory address:

1000 0000 0000 0000 0000 0001 0101 1000

1000 0000 0000 0000 0000

valid dirty tag data

valid dirty tag data

=?

=?

memory address: 0x8 0 0 0 0 0 1 5 8
Accessing the cache

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

memory address: 1000 0000 0000 0000 0000 0001 0101 1000

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

=?

=?
Accessing the cache

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

memory address: 1000 0000 0000 0000 0000

hit? miss?

hit? miss?
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?

Valid dirty tag data

Valid dirty tag data

memory address: 1000 0000 0000 0000 0000

memory address: 0001 0101 1000

Offset:

1000 0000 0000 0000 0000 0001 0101 1000

1000 0000 0000 0000 0000 0001 0101 1000
How many bits in each field?

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

Diagram:
- Tag
- Index
- Offset
- Valid
- Dirty
- Block / cacheline

Hit?

23
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 associative cache
Corollary of $C = \text{ABS}$

- Offset bits: $\log(B)$
- Index bits: $\log(S)$
- Tag bits: $\text{address\_length} - \log(S) - \log(B)$
  - $\text{address\_length}$ is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \mod S = \text{set index}$
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

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
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

- \[ C = \text{ABS} \]
  - \[ 64\text{KB} = 2 \times 64 \times S \]
  - \[ S = 512 \]
  - \[ \text{offset} = \log(64) = 6 \text{ bits} \]
  - \[ \text{index} = \log(512) = 9 \text{ bits} \]
  - \[ \text{tag} = 64 - \log(512) - \log(64) = 49 \text{ bits} \]
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 46 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets

\[
\begin{align*}
C &= \text{ABS} \\
32\text{KB} &= 8 \times 64 \times S \\
S &= 64 \\
\text{offset} &= \log(64) = 6 \text{ bits} \\
\text{index} &= \log(64) = 6 \text{ bits} \\
\text{tag} &= 64 - \log(64) - \log(64) = 52 \text{ bits}
\end{align*}
\]
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
• Slower
  • 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
Multi-layer caches

- Speed of L1 matches the processor
- Caches data/code as many as possible in L2/L3 to avoid DRAM accesses
# Array of structures or structure of arrays

<table>
<thead>
<tr>
<th></th>
<th>Array of objects</th>
<th>object of arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>struct grades</code></td>
<td><code>struct grades</code></td>
</tr>
<tr>
<td></td>
<td>`{</td>
<td>`{</td>
</tr>
<tr>
<td></td>
<td>int id;</td>
<td>int *id;</td>
</tr>
<tr>
<td></td>
<td>double *homework;</td>
<td>double **homework;</td>
</tr>
<tr>
<td></td>
<td>double average;</td>
<td>double *average;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>};</td>
</tr>
<tr>
<td>average of</td>
<td>each student</td>
<td></td>
</tr>
<tr>
<td>average of</td>
<td>each homework</td>
<td></td>
</tr>
</tbody>
</table>
Announcement

• Pick up your midterm
• Homework #4 due next Tuesday
  • We are not using the problems from the textbook
  • We have programming homework this time