Memory hierarchy / Cache

Hung-Wei Tseng
Outline

• Why Cache/Memory Hierarchy
  • CPU/Memory Gap

• Cache organization
  • Why cache works: Locality
  • How cache works: Basic operations of cache
Memory gap
Stored-program computer

Processor

PC

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 t0,120007a88
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 $$$
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)
Memory hierarchy

- **CPU**: Fastest, Most Expensive
  - Access time: < 1ns

- **Cache**: $< 1ns ~ 20 ns

- **Main Memory**: 50-60ns

- **Secondary Storage**: 10,000,000ns

- **Biggest**
Cache organization
What is Cache?

• Cache is 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
Accessing cache

Block (cacheline): The basic unit of data storage in cache. Contains all data with the same tag and index in their address block / cacheline

memory address: 1000 0000 0000 0000 0000 0001 0101 1000

tag index offset

valid tag data

1 1000 0000 0000 0000 0000

Offset: The position of the requesting word in a cache block
Tag: the high order address bits stored along with the data to identify the actual address of the cache line.

Hit: The data was found in the cache
Miss: The data was not found in the cache
Why we build memory hierarchy?

- 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

Fastest, Most Expensive

• 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
Where is 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?

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

```
Array_size = 1024, 0.048s
(5.25X faster)
```

```
Array_size = 1024, 0.252s
```
**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];
    }
}
```

<table>
<thead>
<tr>
<th>Array_size</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.048s</td>
<td>5.25X</td>
</tr>
</tbody>
</table>

```c
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</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.252s</td>
</tr>
</tbody>
</table>
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
Basic organization of cache

If we have two frequently used cache blocks:

Which one should be in the cache?
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?

=?

block / cacheline

1000 0000 0000 0000 0000 0001 0101 1000

1000 0000 0000 0000 0001 0000 1000 0000

1000 0000 0000 0000 0000 0000 0000 0000

valid tag data

hit?

=?
Way associativity

- 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
- Increased 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 cache size and miss rate across different associativities.](image-url)
How many bits in each field?

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

Block / cacheline

Valid  Tag  Data

Valid  Tag  Data
C = ABS

- **C**: Capacity
- **A**: Way-Associativity
  - How many blocks in a set
  - 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
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
- $(\text{address} / \text{block\_size}) \mod S = \text{set index}$
Athlon 64

• L1 data (D-L1) cache configuration of Athlon 64
  • Size 64KB, 2-way set associativity, 64B block
  • Assume 32-bit memory address

Which of the following is correct?

A. Tag is 17 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

\[ C = \text{ABS} \]
\[ 64\text{KB} = 2 \times 64 \times S \]
\[ S = 512 \]
\[ \text{offset} = \lg(64) = 6 \text{ bits} \]
\[ \text{index} = \lg(512) = 9 \text{ bits} \]
\[ \text{tag} = 32 - \lg(512) - \lg(64) = 17 \text{ bits} \]
Core 2

- L1 data (D-L1) cache configuration of Core 2 Duo
  - Size 32KB, 8-way set associativity, 64B block
  - Assume 32-bit memory address
  - Which of the following is NOT correct?
    A. Tag is 20 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets

C = ABS

32KB = 8 * 64 * S

S = 64

offset = lg(64) = 6 bits

index = lg(64) = 6 bits

tag = 32 - lg(64) - lg(64) = 20 bits