Fundamentals of Computer Systems
Caches

Martha A. Kim

Columbia University

Fall 2014

Illustrations Copyright © 2007 Elsevier
Performance depends on which is slowest: the processor or the memory system.
Memory Speeds Haven’t Kept Up

Our single-cycle memory assumption has been wrong since 1980.

## Your Choice of Memories

<table>
<thead>
<tr>
<th></th>
<th>Fast</th>
<th>Cheap</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-Chip SRAM</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Commodity DRAM</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Supercomputer</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
An essential trick that makes big memory appear fast

<table>
<thead>
<tr>
<th>Technology</th>
<th>Cost ($/Gb)</th>
<th>Access Time (ns)</th>
<th>Density (Gb/cm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>30 000</td>
<td>0.5</td>
<td>0.00025</td>
</tr>
<tr>
<td>DRAM</td>
<td>10</td>
<td>100</td>
<td>1 – 16</td>
</tr>
<tr>
<td>Flash</td>
<td>2</td>
<td>300*</td>
<td>8 – 32</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>0.1</td>
<td>10 000 000</td>
<td>500 – 2000</td>
</tr>
</tbody>
</table>

*Read speed; writing much, much slower
A Modern Memory Hierarchy

---

A desktop machine:

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Tech.</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction*</td>
<td>64 K</td>
<td>SRAM</td>
</tr>
<tr>
<td>L1 Data*</td>
<td>64 K</td>
<td>SRAM</td>
</tr>
<tr>
<td>L2*</td>
<td>512 K</td>
<td>SRAM</td>
</tr>
<tr>
<td>L3</td>
<td>2 MB</td>
<td>SRAM</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB</td>
<td>DRAM</td>
</tr>
<tr>
<td>Disk</td>
<td>500 GB</td>
<td>Magnetic</td>
</tr>
</tbody>
</table>

* per core

AMD Phenom 9600
Quad-core
2.3 GHz
1.1–1.25 V
95 W
65 nm
A Simple Memory Hierarchy

First level: small, fast storage (typically SRAM)

Last level: large, slow storage (typically DRAM)

Can fit a subset of lower level in upper level, but which subset?
Locality Example: Substring Matching

Addresses accessed over time
aka “reference stream”

temporal locality: if needed X recently, likely to need X again soon

spatial locality: if need X, likely also need something near X
Cache

Highest levels of memory hierarchy

Fast: level 1 typically 1 cycle access time

With luck, supplies most data

Cache design questions:

What data does it hold?  Recently accessed

How is data found?  Simple address hash

What data is replaced?  Often the oldest
What Data is Held in the Cache?

Ideal cache: always correctly guesses what you want before you want it.

Real cache: never that smart

Caches Exploit

**Temporal Locality**
Copy newly accessed data into cache, replacing oldest if necessary

**Spatial Locality**
Copy nearby data into the cache at the same time
Specifically, always read and write a block, also called a line, at a time (e.g., 64 bytes), never a single byte.
Memory Performance

Hit: Data is found in the level of memory hierarchy

Miss: Data not found; will look in next level

Hit Rate = \frac{\text{Number of hits}}{\text{Number of accesses}}

Miss Rate = \frac{\text{Number of misses}}{\text{Number of accesses}}

Hit Rate + Miss Rate = 1

The expected access time $E_L$ for a memory level $L$ with latency $t_L$ and miss rate $M_L$:

$E_L = t_L + M_L \cdot E_{L+1}$
Memory Performance Example

Two-level hierarchy: Cache and main memory
Program executes 1000 loads & stores
750 of these are found in the cache

What’s the cache hit and miss rate?

Hit Rate = \frac{750}{1000} = 75\% 

Miss Rate = 1 - 0.75 = 25\%

If the cache takes 1 cycle and the main memory 100, 
What’s the expected access time?

Expected access time of main memory: \( E_1 = 100 \) cycles

Access time for the cache: \( t_0 = 1 \) cycle

Cache miss rate: \( M_0 = 0.25 \)

\( E_0 = t_0 + M_0 \cdot E_1 = 1 + 0.25 \cdot 100 = 26 \) cycles
Memory Performance Example

Two-level hierarchy: Cache and main memory
Program executes 1000 loads & stores
750 of these are found in the cache

What’s the cache hit and miss rate?

Hit Rate = \( \frac{750}{1000} = 75\% \)
Miss Rate = \( 1 - 0.75 = 25\% \)

If the cache takes 1 cycle and the main memory 100,

What’s the expected access time?
Memory Performance Example

Two-level hierarchy: Cache and main memory

Program executes 1000 loads & stores

750 of these are found in the cache

What’s the cache hit and miss rate?

$$\text{Hit Rate} = \frac{750}{1000} = 75\%$$
$$\text{Miss Rate} = 1 - 0.75 = 25\%$$

If the cache takes 1 cycle and the main memory 100,

What’s the expected access time?

Expected access time of main memory: $$E_1 = 100$$ cycles

Access time for the cache: $$t_0 = 1$$ cycle

Cache miss rate: $$M_0 = 0.25$$

$$E_0 = t_0 + M_0 \cdot E_1 = 1 + 0.25 \cdot 100 = 26$$ cycles
This simple cache has

- **8 sets**
- **1 block per set**
- **4 bytes per block**

To simplify answering “is this data in the cache?,” each byte is mapped to exactly one set.
Direct-Mapped Cache Hardware

Address bits:
0–1: byte within block
2–4: set number
5–31: block “tag”

Cache hit if
in the set of the address,
- block is valid (V=1)
- tag (address bits 5–31) matches
Direct-Mapped Cache Behavior

A dumb loop:
repeat 5 times
load from 0x4;
load from 0xC;
load from 0x8.

```
li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0xC($0)
lw $t3, 0x8($0)
addiu $t0, $t0, -1
j l1
done:
```

Assuming the cache starts empty, what’s the miss rate?

Cache when reading 0x4 last time

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...0C]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...08]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...04]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)

When two recently accessed addresses map to the same cache block,
Direct-Mapped Cache Behavior

A dumb loop:
repeat 5 times
load from 0x4;
load from 0xC;
load from 0x8.

```
li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0xC($0)
lw $t3, 0x8($0)
addiu $t0, $t0, -1
j l1
done:
```

Cache when reading 0x4 last time

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...0C]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...08]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...04]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When two recently accessed addresses map to the same cache block,

```
Assuming the cache starts empty, what’s the miss rate?

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>M</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>C</td>
<td>8</td>
<td>4</td>
<td>C</td>
<td>8</td>
<td>4</td>
<td>C</td>
<td>8</td>
<td>4</td>
<td>C</td>
<td>8</td>
<td>4</td>
<td>C</td>
<td>8</td>
</tr>
</tbody>
</table>
```

3/15 = 0.2 = 20%
**Direct-Mapped Cache: Conflict**

A dumber loop:
- repeat 5 times
- load from 0x4;
- load from 0x24

```assembly
li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addiu $t0, $t0, -1
j l1
done:
```

**Cache State**

Assuming the cache starts empty, what’s the miss rate?

These are *conflict misses*
Direct-Mapped Cache: Conflict

A dumber loop:
repeat 5 times
load from 0x4;
load from 0x24

li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addiu $t0, $t0, -1
j l1
done:

Assuming the cache starts empty, what’s the miss rate?

<table>
<thead>
<tr>
<th></th>
<th>Set 7 (111)</th>
<th>Set 6 (110)</th>
<th>Set 5 (101)</th>
<th>Set 4 (100)</th>
<th>Set 3 (011)</th>
<th>Set 2 (010)</th>
<th>Set 1 (001)</th>
<th>Set 0 (000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tag</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
<td>mem[0x00...24]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>001</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Byte Offset</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These are conflict misses
No Way! Yes Way! 2-Way Set Associative Cache

Memory Address

Tag Set Offset

Way 1 Way 0

V Tag Data V Tag Data

Set 3 Set 2 Set 1 Set 0

Data

Hit

1
0

Hit 1

V = 00

32 32

28 28

Way 1 Way 0

Set 3 Set 2 Set 1 Set 0

Data
2-Way Set Associative Behavior

li $t0, 5  
l1: beq $t0, $0, done  
lw $t1, 0x4($0)  
lw $t2, 0x24($0)  
addiu $t0, $t0, -1  
j l1

done:

Assuming the cache starts empty, what’s the miss rate?

\[
\begin{array}{cccccccccccc}
4 & 24 & 4 & 24 & 4 & 24 & 4 & 24 & 4 & 24 & 4 & 24 \\
M & M & H & H & H & H & H & H & H & H & H & H \\
\end{array}
\]

\[
\frac{2}{10} = 0.2 = 20\%
\]

Associativity reduces conflict misses

<table>
<thead>
<tr>
<th>Way 1</th>
<th></th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
<td>Data</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3  
Set 2  
Set 1  
Set 0
An Eight-way Fully Associative Cache

![Diagram of an eight-way fully associative cache]

No conflict misses: only compulsory or capacity misses

Either very expensive or slow because of all the associativity
Exploiting Spatial Locality: Larger Blocks

0x8000 0009C:

Memory Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>100...100</td>
<td>1</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

Memory Address

- 2 sets
- 1 block per set (Direct Mapped)
- 4 words per block
The dumb loop:
repeat 5 times
load from 0x4;
load from 0xC;
load from 0x8.

li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0xC($0)
lw $t3, 0x8($0)
addiu $t0, $t0, -1
j l1
done:

Assuming the cache starts empty, what’s the miss rate?
Direct-Mapped Cache Behavior w/ 4-word block

The dumb loop:
repeat 5 times
load from 0x4;
load from 0xC;
load from 0x8.

```
li $t0, 5
l1: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0xC($0)
lw $t3, 0x8($0)
addiu $t0, $t0, -1
j l1
```

done:

Assuming the cache starts empty, what’s the miss rate?

```
<table>
<thead>
<tr>
<th>00...00</th>
<th>mem[0x00...0C]</th>
<th>mem[0x00...08]</th>
<th>mem[0x00...04]</th>
<th>mem[0x00...00]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem[0x00...0C]</td>
<td>mem[0x00...08]</td>
<td>mem[0x00...04]</td>
<td>mem[0x00...00]</td>
</tr>
</tbody>
</table>
```

```
M H H H H H H H H H H H H H
1/15 = 0.0666 = 6.7%
```

Larger blocks reduce compulsory misses by exploiting spatial locality
Stephen’s Desktop Machine Revisited

AMD Phenom 9600
Quad-core
2.3 GHz
1.1–1.25 V
95 W
65 nm

On-chip caches:

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Sets</th>
<th>Ways</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1I*</td>
<td>64 K</td>
<td>512</td>
<td>2-way</td>
<td>64-byte</td>
</tr>
<tr>
<td>L1D*</td>
<td>64 K</td>
<td>512</td>
<td>2-way</td>
<td>64-byte</td>
</tr>
<tr>
<td>L2*</td>
<td>512 K</td>
<td>512</td>
<td>16-way</td>
<td>64-byte</td>
</tr>
<tr>
<td>L3</td>
<td>2 MB</td>
<td>1024</td>
<td>32-way</td>
<td>64-byte</td>
</tr>
</tbody>
</table>

* per core
<table>
<thead>
<tr>
<th>Chip</th>
<th>Year</th>
<th>Freq. (MHz)</th>
<th>L1 Data</th>
<th>L1 Instr</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>80386</td>
<td>1985</td>
<td>16–25</td>
<td>off-chip</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>80486</td>
<td>1989</td>
<td>25–100</td>
<td>8K</td>
<td>unified</td>
<td>off-chip</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60–300</td>
<td>8K</td>
<td>8K</td>
<td>off-chip</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>150–200</td>
<td>8K</td>
<td>8K</td>
<td>256K–1M (MCM)</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>233–450</td>
<td>16K</td>
<td>16K</td>
<td>256K–512K (Cartridge)</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>450–1400</td>
<td>16K</td>
<td>16K</td>
<td>256K–512K</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>1400–3730</td>
<td>8–16K</td>
<td>12k op trace cache</td>
<td>256K–2M</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2003</td>
<td>900–2130</td>
<td>32K</td>
<td>32K</td>
<td>1M–2M</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>2005</td>
<td>1500–3000</td>
<td>32K per core</td>
<td>32K per core</td>
<td>2M–6M</td>
</tr>
</tbody>
</table>