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><strong>On-Chip SRAM</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Commodity DRAM</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Supercomputer</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Memory Hierarchy

Fundamental trick to making a 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

**My 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
Temporal Locality

What path do your eyes take when you read this?

Did you look at the drawings more than once?

Euclid’s Elements
Spatial Locality

If you need something, you may also need something nearby
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?
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: \( M_0 = 0.25 \)

\[
E_0 = t_0 + M_0 \cdot E_1 = 1 + 0.25 \cdot 100 = 26 \text{ cycles}
\]
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 at a time (e.g., 64 bytes), never a single byte.
This simple cache has

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

To simplify answering “is this memory 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:
```

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

```
Cache when reading 0x4 last time
```

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

```
V Tag Data
0
0
0
0
0
0
00
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)
```

4 C 8 4 C 8 4 C 8 4 C 8
M M M H H H H H H H H H H H
3 / 15 = 0.2 = 20%
### 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**

---

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

<table>
<thead>
<tr>
<th>Cache State</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>M M M M M M M M M M M M M M M M</td>
<td>3/15 = 0.2 = 20%</td>
</tr>
</tbody>
</table>

---

### Memory Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...00</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

---

### Data Tag

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

---

### Tag Set

<table>
<thead>
<tr>
<th>Tag Set</th>
<th>Byte</th>
<th>Offset</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 7</td>
<td>(111)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 6</td>
<td>(110)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 5</td>
<td>(101)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 4</td>
<td>(100)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>(011)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td>(010)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>(001)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 0</td>
<td>(000)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

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

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>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>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...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache State

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

4 24 4 24 4 24 4 24 4 24 4 24
M M M M M M M M M M

10/10 = 1 = 100% Oops

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Tag</th>
<th>Set Offset</th>
<th>Byte Offset</th>
<th>Data Tag</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 1
- V
- Tag
- Data

Way 0
- V
- Tag
- Data

Set 3
Set 2
Set 1
Set 0

- Hit 1
- Hit 0
- Hit

- 28
- 32
- 28
- 32
- 28
- 32

Data
2-Way Set Associative Behavior

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

```
4 24 4 24 4 24 4 24 4 24
M M H H H H H H H H
```

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

Associativity reduces conflict misses

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

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

#### Diagram:

- **Tag**: 27 bits
- **Set**: 2 bits
- **Offset**: 11 bits
- **Byte**: 3 bits

**Memory Address**

**Data**

- **Set 1**: Hit
- **Set 0**: Miss

- **1 block per set (Direct Mapped)**
- **4 words per block**
**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
```

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

```
Cache when reading 0xC
```

Larger blocks reduce compulsory misses by exploiting spatial locality
**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?

```
   4 C 8 4 C 8 4 C 8 4 C 8 4 C 8
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
## Intel On-Chip Caches

<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 unified</td>
<td></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>