memory woes:

physically separate memory makes memory accesses SLOW!

P and M

colocated?
very expensive!
or memory too small!
So, design the HW + SW to make this problem less bad.

Look at memory reference patterns. Design a special memory system.
The patterns come from the fetch + execute cycle:

1. fetch instruction
2. update PC
3. decode
4. get operands
5. do operation
6. put result(s) away

memory references.

They exhibit locality.
temporal locality

Recently referenced memory locations are likely to referenced again (soon!)

```
loop:     instr 1 @ A1
     instr 2 @ A2
     instr 3 @ A3
     jmp/b loop @ A4
```

Instruction stream references:

```
A1 A2 A3 A4 A1 A2 A3 A4 A1 A2 A3 ...
```

Note that the same memory location is repeatedly read (for the fetch).
spatial locality

Memory locations near to referenced locations are likely to also be referenced.

Array

Code must do something to each element of the array.

Must load each element.
The fetch of the code exhibits a high degree of spatial locality.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

I2 is next to I1.

If these instructions are not branches, then we fetch I1, I2, I3, etc.
Design a cache to attempt to hold copies of memory locations. Which locations?

Put cache on chip for speed but it will hold fewer bytes than main memory.
Intel 386 chip (1985 image)
Pentium II (1997 image)
P sends memory request to C.

- **hit**: requested location's copy *is* in the C
- **miss**: requested location's copy *is NOT* in the C. So, send the memory access to M.
Needed terminology:

\[
\text{miss ratio} = \frac{\# \text{ of misses}}{\text{total } \# \text{ of accesses}}
\]

\[
\text{hit ratio} = \frac{\# \text{ of hits}}{\text{total } \# \text{ of accesses}}
\]

or \[1 - \text{miss ratio}\]

You already assumed that \[\text{total } \# \text{ of accesses} = \# \text{ of misses} + \# \text{ of hits}\]
When a memory access causes a miss, place that location's bytes and its neighbors (spatial locality) into the cache. Keep the block of bytes there for as long as possible (temporal locality).

A statistic to measure how well this works:

\[
\text{Average Memory Access Time} = T_c + (\text{miss ratio})(T_m)
\]
Quick example:

\[ T_c = 1 \text{ nsec} \]
\[ T_m = 20 \text{ nsec} \]

hit ratio is .98 for measured program

\[ AMAT = 1 + (.02)(20) \]
\[ = 1.4 \text{ nsec} \]

Note: individual memory access takes either 1 nsec (hit) or 21 nsec (miss).
Design the cache such that it can quickly do a lookup: given the address, decide if that location is in the cache (hit) or not (miss).

Divide all of memory into fixed size blocks. Transfer a block on a miss. Keep the block in the cache until something else knocks it out.

What block size should we pick?
Let's play with an unrealistically small cache example. It will hold only 4 blocks.

A block lands in a block frame set (textbook)

2 bits of address are used to determine the frame #. LSBs identify byte/word within the block.
Each main memory block maps to a specific block frame.

2 bits of the address define this mapping.
Many main memory blocks map to the same block frame.

Only 1 can be in the block frame.

We have to quickly decide if the right one is in the frame.

The only thing we have to use is the address.

So... store the remainder of the address of a block with the block.

Called a tag.
Address as used by the cache for a lookup.

| tag | index# | byte w# block |

Bits of SRAM cannot identify whether a block from memory has or has not been placed in a block frame.

So, keep 1 bit per frame to identify if data is valid or not.
Completed diagram of the cache:

```
tag | index # | byte w/i block
```

```
\downarrow \text{valid}
```

```
tag | \text{data blocks}
00 |
01 |
10 |
11 |
```

© Karen Miller, 2011
This cache is called direct mapped or 1-way set associative or set associative, with a set size of 1

Each index # maps to exactly 1 block frame
Lookup algorithm:

(cache receives address)

use index to identify frame

if frame is valid

if frame's tag matches address' tag

  HIT

else

  MISS

else

  MISS

<table>
<thead>
<tr>
<th>Valid</th>
<th>tag</th>
<th>data blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
On a miss

- Send the memory request to main memory.
- Memory returns the entire block containing the needed byte/word.
- Place the block into the frame.
  - Set the tag bits
  - Mark the frame valid.

And, while doing this, extract the byte/word and return it to the processor, completing the memory access.
Types of misses:

1) compulsory
2) conflict
3) capacity
To reduce conflict misses
increase set associativity

2-way set associative
2 blocks per set (line)

4-way set associative
larger set size

😊 tends to lead to higher hit ratio (due to fewer conflict misses)

😢 amount of circuitry goes up, leading to increase in $T_c$
Implementing writes

1. write through
   change data in the cache, and send the write to main memory

   slow 😞, but very little circuitry 😊
2) **write back**

- at first, change data in the cache
- write to memory only when necessary

dirty bit is set on a write, to identify blocks to be written back to memory

when a program completes, all dirty blocks must be written to memory...
write back (continued)

- faster 😊
multiple stores to the same location result in only 1 main memory access
- more circuitry 😞
  - must maintain the dirty bit
  - *dirty miss*: a miss caused by a read or write to a block not in the cache, but the required block frame has its dirty bit set. So, there is a write of the dirty block, followed by a read of the requested block.
How about 2 separate caches?

**I-cache**
- for instructions only
- can be rather small, and still have excellent performance.

**D-cache**
- for data only
- needs to be fairly large

© Karen Miller, 2011
We can send memory accesses to the 2 caches independently... 😊 (increased parallelism)
Caching works so well at hiding latency, that we use many levels.