Cache memories
There is a trade-off between speed and size of memories.

A memory component is either fast and small or big and slow (or a compromise somewhere in between).

Programmers of course want fast and big memories.

It should take one clock cycle to fetch from memory and it should be several gigabytes big.

Unfortunately, reading from memory can take one hundred clock cycles.

So, we need to solve a problem!
Two Observations

- When solving a computer engineering research problem we should always first try to make observations about how the system behaves and then exploit that knowledge.
- A first observation about memory usage is that programs usually access memory somewhat regularly, which is called **locality**.
  - **Temporal locality** means that if a program has accessed a particular location $A$ in memory, it is likely it will access $A$ again soon.
  - **Spatial locality** means that if a program has accessed a word at address $A$ in memory, it is likely it will soon access the word at address $A+1$. 

Jonas Skeppstedt (js@cs.lth.se)
Examples of Temporal Locality

- The instructions in a loop are accessed next iteration as well.
- The loop index variable is usually accessed frequently.
- The stack space is often accessed again when a new function is called (since that space is reused).
- An object is typically accessed for a while and then the program is done with it.
Examples of Spatial Locality

- The instructions are accessed one after the other — until there is a branch.
- Elements of an array are often accessed one after the other.
- Often several variables in an object is accessed and if these are put close together (by the programmer) then there will be spatial locality.
What is needed is a small memory on the same chip as the processor.

If we describe our **hardware** in C, then extending our machine.c could look like:

```c
typedef struct {
    int reg[32];
    int pc;
    struct {
        bool valid;
        int data;
        int address;
    } cache_array[8];
} cpu_t;
```
We have now a cache which can store eight popular words. The cache array contains eight pairs of **data** and **address**. There is also a boolean called **valid** which tells us whether the data and address are valid for that row. Recall that on paper we can write question marks but the machine can only write ones and zeros and cannot see whether they make sense so a separate bit must tell whether they mean anything useful. Suppose the compiler has decided that a global variable X should be put at the address 293, or 0x125, or 0001 0010 0101.
Using our Cache

- When the program (or CPU) wants to read variable X, it should check whether any of the eight rows has \texttt{valid = true} and \texttt{address = 293}
- If the CPU found one such row (or, let’s call it line), then the CPU can take the data from that line and avoid waiting for the slow memory! Great!
- We must call this event something: a \texttt{cache hit}
- It can save us 100 clock cycles.
In hardware all iterations are executed concurrently!!

The openmp directive is here to make you alert on that this is not a sequential loop.

case LD: found = false;
    address = source1 + constant;
    #pragma omp parallel for
    for (i = 0; i < 8; i++) {
        if (cache_array[i].valid &&
            cache_array[i].address == address)
            data = cache_array[i].data;
        found = true;
        break;
    }
}
Cache Replacement

- We need to select one line.
- If there is one line with `valid = false` then we select that one.
- Otherwise, for now, we take a random line (row).
- If the row we selected had valid data, we need to copy the old data contents to memory (otherwise it’s lost).
- Then we read our data from memory.
- Then we write our data into the selected row, set the address to our address and set valid to true.
if (!found) {
    i = select_row();

    if (cache_array[i].valid) // save old data to memory
        memory[cache_array[i].address] = cache_array[i].data;

    // read our data from memory
    data = memory[address];

    // save our data in the cache
    cache_array[i].data = data;
    cache_array[i].address = address;
    cache_array[i].valid = true;
}
Similar for a Store

case ST:
  found = false;
  address = source2 + constant;
  data = source1;
#pragma omp parallel for
for (i = 0; i < 8; i++) {
    if (cache_array[i].valid &&
        cache_array[i].address == address) {
        cache_array[i].data = data;
        found = true;
        break;
    }
}
if (found)
break;
i = select_row();
if (cache_array[i].valid)
    memory[cache_array[i].address] = cache_array[i].data;
cache_array[i].data = data;
cache_array[i].address = address;
cache_array[i].valid = true;

- Next time we want to read or write that variable it is likely that it will be found in the cache.
No, it doesn’t exist!

It only exists in the software model of the hardware.

Recall: in hardware the loop is run in parallel.

In our case, there are eight so called comparators which compare the address requested with the address in its row and says ”here!” if the addresses are equal and the valid bit is true.
A look at our Cache

- Our cache does not exploit spatial locality, yet.
- Instructions may also be put in the cache.
- Hit rate is the fraction of hits in the cache.
- Let us test it on the factorial program.

<table>
<thead>
<tr>
<th># rows</th>
<th>reads</th>
<th>read hits</th>
<th>writes</th>
<th>write hits</th>
<th>hit rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>77</td>
<td>21</td>
<td>11</td>
<td>0</td>
<td>23.9 %</td>
</tr>
<tr>
<td>16</td>
<td>77</td>
<td>36</td>
<td>11</td>
<td>0</td>
<td>40.9 %</td>
</tr>
<tr>
<td>32</td>
<td>77</td>
<td>50</td>
<td>11</td>
<td>0</td>
<td>56.8 %</td>
</tr>
<tr>
<td>64</td>
<td>77</td>
<td>50</td>
<td>11</td>
<td>0</td>
<td>56.8 %</td>
</tr>
<tr>
<td>128</td>
<td>77</td>
<td>50</td>
<td>11</td>
<td>0</td>
<td>56.8 %</td>
</tr>
</tbody>
</table>
What can have happened?

- All writes always miss — the factorial program writes to new places on the stack.
- The reads benefit from a larger cache since then what fac(5) and fac(4) saved on the stack will remain in the cache when they later read the saved parameter and return values.
- The 27 read misses are due to instruction which are fetched for the first time.
Some Comments

- Our cache is very small, only a half KB. Usually the cache closest to the processor, called the L1 cache is 32 KB or 64 KB.
- Having 128 comparators might be feasible but hundreds of them is too much. That is a problem which we must address.
- It is much better to read and write multiple words from/to memory rather than only one at a time. This we will also address.
In our cache, a word can be put in any row in the cache.
That means every row must be checked to see if the address matches.
Our cache is called a **fully-associative cache**. These are expensive.
If there is a function (in hardware) which maps an address to a particular row, then we only need one comparator, since there is only one row to look in. That is called a **direct-mapped cache**.
In a direct-mapped cache we can use the least significant bits of the address as the function.
Why should we not use the most significant bits as the function instead???
The purpose is to avoid so many comparators.
In the C code the loop will disappear.
If the number of rows in the cache is a power of two we can do as follows instead of having a loop:

```c
i = address & (CACHE_ROWS - 1);
if (cache_array[i].valid
    && cache_array[i].address == address)
    data = cache_array[i].data;
// etc as in LD: above
```
In direct mapped cache, we can be unlucky and having two frequently used items which are mapped to the same row in the cache which will result in many cache misses when they replace each other.

A compromise is a so called **N-way associative cache**.

Now we group the cache rows into sets, where each set has N rows.

The number of sets, \( \text{CACHE\_SETS} = \frac{\text{CACHE\_ROWS}}{N} \).

An address is now mapped to a set and within its set, the address can be put in any of the N rows.

N comparators are needed now, and typical values of N is 2, 4 or 8.
i = address & (CACHE_SETS - 1);
for (j = 0; j < N; j++) {
    if (cache_array[i][j].valid
        && cache_array[i][j].address == address) {
        data = cache_array[i][j].data;
        found = true;
        break;
    }
}

- In a fully associative cache, CACHE_SETS = 1.
- In a direct-mapped cache, N = 1.
So far we have only transferred one word between the cache and the memory.

It is more efficient if multiple words, say 8, 16, or 32 words are transferred at a time.

Power.ludat.lth.se transfers 64 bytes at a time.

Assume instead that our cache block size is 8 words.

Then we can eg fetch eight instructions at a time.

Since we store 8 consecutive words from memory in a cache row, we only need, of course, to know the address of the first word in the block.
We can now view memory as an array, not of words, but of cache blocks.

When the cache block size is eight words (or \(2^3\) words) we get the cache block number of a word by dividing the word number by eight.

Alternatively we shift the word number, ie the address, to the right by three, ie, we throw the last three bits of the address away.

The number of words in a cache block is called the BLOCK_SIZE.

Actually the BLOCK_SIZE is the number of bytes but we simplify the presentation and only consider words.
block_num = address / BLOCK_SIZE;

i = block_num & (CACHE_SETS - 1);

for (j = 0; j < N; j++) {
    if (cache_array[i][j].valid
        && cache_array[i][j].block_num == block_num) {

        k = address & (BLOCK_SIZE - 1);
        data = cache_array[i][j].data[k];
        found = true;
        break;
    }
}

• The data in a row is now an array of BLOCK_SIZE words.
In 1827 Fourier published a method for solving linear inequalities in the real case. This method is known as Fourier-Motzkin elimination and is used in compilers as an approximation.

If Fourier-Motzkin elimination finds that there is no real solution, then there certainly is no integer either. But if there is a real solution, there may or may not be an integer solution.

Fourier-Motzkin elimination is regarded as a time-consuming algorithm and to apply it so perhaps thousands of data dependence tests may make the compiler too slow. Therefore, it is used as a backup tests when other faster tests fail to prove independence.
An interesting question is how frequently Fourier-Motzkin elimination finds a real solution when there is no integer solution. Some special cases can be exploited.

For instance, if a variable $x_i$ must satisfy $2.2 \leq x_i \leq 2.8$ then there is no integer solution.

Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution.

It is easiest to understand Fourier-Motzkin elimination if we first look at an example.
Fourier-Motzkin Elimination

- Assume we wish to solve the following system linear inequalities.

\[
\begin{align*}
2x_1 & - 11x_2 \leq 3 \\
-3x_1 & + 2x_2 \leq -5 \\
x_1 & + 3x_2 \leq 4 \\
-2x_1 & \leq -3
\end{align*}
\]  

(1)

- We will first eliminate \( x_2 \) from the system, and then check whether the remaining inequalities can be satisfied. To eliminate \( x_2 \), we start out with sorting the rows with respect to the coefficients of \( x_2 \):

\[
\begin{align*}
-3x_1 & + 2x_2 \leq -5 \\
x_1 & + 3x_2 \leq 4 \\
2x_1 & - 11x_2 \leq 3 \\
-2x_1 & \leq -3
\end{align*}
\]  

(2)
First we want to have rows with positive coefficients of $x_2$, then negative, and lastly zero coefficients.

Next we divide each row by its coefficient (if it is nonzero) of $x_2$:

\[
\begin{align*}
\frac{-3}{2} x_1 & + x_2 \leq \frac{-5}{2} \\
\frac{1}{3} x_1 & + x_2 \leq \frac{4}{3} \\
\frac{2}{11} x_1 & - x_2 \geq \frac{3}{11}
\end{align*}
\]

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient. We can now rearrange the system to isolate $x_2$:

\[
\begin{align*}
x_2 & \leq \frac{3}{2} x_1 - \frac{5}{2} \\
x_2 & \leq -\frac{1}{3} x_1 + \frac{4}{3}
\end{align*}
\]
At this point, we make a record of the minimum and maximum values that $x_2$ can have, expressed as functions of $x_1$. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1) \quad (5)$$

where

$$b_2(x_1) = 11x_1 \quad (6)$$

$$B_2(x_1) = \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right)$$
To eliminate $x_2$ from the system, we simply combine the inequalities which had positive coefficients of $x_2$ with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$\begin{align*}
\frac{2}{11} x_1 - \frac{3}{11} x_1 & \leq \frac{3}{2} x_1 - \frac{5}{2} \\
\frac{2}{11} x_1 - \frac{3}{11} x_1 & \leq -\frac{1}{3} x_1 + \frac{4}{3}
\end{align*}$$

These are simplified and the inequality with the zero coefficient of $x_2$ is brought back:

$$\begin{align*}
-\frac{29}{22} x_1 & \leq -\frac{49}{22} \\
-\frac{17}{33} x_1 & \leq \frac{53}{33} \\
-2x_1 & \leq -3
\end{align*}$$
We can now repeat parts of the procedure above:

\[ x_1 \leq \frac{53}{17} \]
\[ x_1 \geq \frac{49}{29} \]
\[ x_1 \geq \frac{3}{2} \]  \hspace{1cm} (9)

We find that

\[
\begin{align*}
b_1() &= \max(49/29, 3/2) = 49/29 \\
B_1() &= 53/17
\end{align*}
\]  \hspace{1cm} (10)

The solution to the system is \[ \frac{49}{29} \leq x_1 \leq \frac{53}{17} \] and \[ b_2(x_1) \leq B_2(x_1) \] for each value of \( x_1 \).
procedure fourier_motzkin_elimination (x, A, c)
    r ← m, s ← n, T ← A, q ← c
    while (1) {
        n₁ ← number of inequalities with positive \( t_{rj} \)
        n₂ ← n₁ + number of inequalities with negative \( t_{rj} \)
        Sort the inequalities so that the \( n_1 \) with \( t_{rj} > 0 \) come first,
        then the \( n_2 - n_1 \) with \( t_{rj} < 0 \) come next,
        and the ones with \( t_{rj} = 0 \) come last.
        for (i = 1; i ≤ r - 1; i ← i + 1)
            for (j = 1; i ≤ n₂; j ← j + 1)
                \( t_{ij} \) ← \( t_{ij} \) / \( t_{rj} \)
        for (j = 1; i ≤ n₂; j ← j + 1)
            \( q_j \) ← \( q_j \) / \( t_{rj} \)
        if (n₂ > n₁)
            \( b_r(x_1, x_2, ..., x_{r-1}) = \max_{n_1 + 1 \leq j \leq n_2} \left( -\sum_{i=1}^{r-1} t_{ij}x_i + q_i \right) \)
        else
            \( b_r \) ← −∞
        if (n₁ > 0)
            \( j_r(x_1, x_2, ..., x_{r-1}) = \min_{n_1 + 1 \leq j \leq n_2} \left( -\sum_{i=1}^{r-1} t_{ij}x_i + q_i \right) \)
        else
            \( B_r \) ← ∞
        if (r = 1)
            return make_solution()
/* We will now eliminate \( x_r \). */

\[
s' \leftarrow s - n_2 + n_1(n_2 - n_1)
\]

if \((s' = 0)\) {
    /* We have not discovered any inconsistency and */
    /* we have no more inequalities to check. */
    /* The system has a solution. */

    The solution set consists of all real vectors \((x_1, x_2, \ldots, x_m)\),
    where \(x_{r-1}, x_{r-2}, \ldots, x_1\) are chosen arbitrarily, and
    \(x_m, x_{m-1}, \ldots, x_r\) must satisfy
    \[
        b_i(x_1, x_2, \ldots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \ldots, x_{i-1}) \quad \text{for} \quad r \leq i \leq m.
    \]
    return solution set.
}

/* There are now \( s' \) inequalities in \( r - 1 \) variables. */

The new system of inequalities is made of two parts:

\[
    \sum_{i=1}^{r-1} (t_{ik} - t_{il})x_i \leq q_k - q_j \quad \text{for} \quad 1 \leq k \leq n_1, n_1 + 1 \leq j \leq n_2
\]

\[
    \sum_{i=1}^{r-1} t_{ij}x_i \leq q_j \quad \text{for} \quad n_2 + 1 \leq j \leq s
\]

and becomes by setting \( r = r \leftarrow 1 \) and \( s \leftarrow s' \):

\[
    \sum_{i=1}^{r-1} t_{ij}x_i \leq q_j \quad \text{for} \quad 1 \leq j \leq s
\]

} end

function make_solution()

/* We have come to the last variable \( x_1 \). */

if \((b_1 > B_1 \quad \text{or} \quad \text{there is a} \quad q_j < 0 \quad \text{for} \quad n_2 + 1 \leq j \leq s))\)
    return there is no solution

The solution set consists of all real vectors \((x_1, x_2, \ldots, x_m)\),
    such that \(b_i(x_1, x_2, \ldots, x_m) \leq x_i \leq B_i(x_1, x_2, \ldots, x_m) \quad \text{for} \quad 1 \leq i \leq m.

return solution set.

end