Automata for Basic Predicates:

$x = 5$
Automata for Basic Predicates:

\[ x + x = y \]
Automata for Basic Predicates:

\[ x+y = z \]
Results:

Ferrante, Rackoff, 1973: \[ \text{PSAT} \leq \text{DSpace}(2^{2^c n}) \]
Results:

Ferrante, Rackoff, 1973: \( \text{PSAT} \leq \text{DSpace}(2^{2c \cdot n}) \)

Fischer, Rabin, 1974: \( \text{PSAT} \geq \text{NTime}(2^{2c \cdot n}) \)
3.3 Improving the Memory Layout

Goal:

• Better utilization of caches
  \[\Rightarrow\] reduction of the number of cache misses

• Reduction of allocation/de-allocation costs
  \[\Rightarrow\] replacing heap allocation by stack allocation
  \[\Rightarrow\] support to free superfluous heap objects

• Reduction of access costs
  \[\Rightarrow\] short-circuiting indirection chains (Unboxing)
1. **Cache Optimization:**

**Idea:** *local memory access*

- Loading from memory fetches not just one byte but fills a complete cache line.
- Access to neighbored cells become cheaper.
- If all data of an inner loop fits into the cache, the iteration becomes maximally memory-efficient ...
Possible Solutions:

→ Reorganize the data accesses!
→ Reorganize the data!

Such optimizations can be made fully automatic only for arrays :-(

Example:

\[
\text{for } (j = 1; j < n; j++) \\
\quad \text{for } (i = 1; i < m; i++) \\
\quad \quad a[i][j] = a[i - 1][j - 1] + a[i][j];
\]
At first, always iterate over the rows!

Exchange the ordering of the iterations:

\[
\text{for } (i = 1; i < m; i++)
\]
\[
\text{for } (j = 1; j < n; j++)
\]
\[
a[i][j] = a[i - 1][j - 1] + a[i][j];
\]

When is this permitted???
Iteration Scheme: before:
Iteration Scheme: after:
Iteration Scheme: allowed dependencies:
In our case, we must check that the following equation systems have no solution:

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>((i_1, j_1) = (i_2 - 1, j_2 - 1))</td>
<td>((i_1, j_1) = (i_2 - 1, j_2 - 1))</td>
</tr>
<tr>
<td>(i_1 \leq i_2)</td>
<td>(i_1 \leq i_2)</td>
</tr>
<tr>
<td>(j_2 \leq j_1)</td>
<td>(j_1 \leq j_2)</td>
</tr>
</tbody>
</table>

The first implies: \(j_2 \leq j_2 - 1\)  Hurra!
The second implies: \(i_2 \leq i_2 - 1\)  Hurra!
Example: Matrix-Matrix Multiplication

\[
\begin{align*}
\text{for } (i = 0; i < N; i++) \\
&\quad \text{for } (j = 0; j < M; j++) \\
&\quad \quad \text{for } (k = 0; k < K; k++) \\
&\quad \quad \quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{align*}
\]

Over \( b[][] \) the iteration is \textit{columnwise} :-(

754
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

755
Exchange the two inner loops:

\[
\text{for } (i = 0; i < N; i++) \\
\quad \text{for } (k = 0; k < K; k++) \\
\quad \quad \text{for } (j = 0; j < M; j++) \\
\quad \quad \quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\]

Is this permitted ???
Discussion:

- Correctness follows as before :-(
- A similar idea can also be used for the implementation of multiplication for row compressed matrices :-(
- Sometimes, the program must be massaged such that the transformation becomes applicable :-(
- Matrix-matrix multiplication perhaps requires initialization of the result matrix first ...
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        { 
            c[i][j] = 0;
            for (k = 0; k < K; k++)
                c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
        }

• Now, the two iterations can no longer be exchanged  :-(

• The iteration over j, however, can be duplicated ...
for \( (i = 0; i < N; i++) \) {
  for \( (j = 0; j < M; j++) \) \( c[i][j] = 0; \)
  for \( (j = 0; j < M; j++) \)
    for \( (k = 0; k < K; k++) \) \( c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \)
}

**Correctness:**

\( \implies \) The read entries (here: no) may not be modified in the remaining body of the loop !!!

\( \implies \) The ordering of the write accesses to a memory cell may not be changed :)
We obtain:

\[
\text{for } (i = 0; i < N; i++) \{ \\
\text{for } (j = 0; j < M; j++) \ c[i][j] = 0; \\
\text{for } (k = 0; k < K; k++) \\
\hspace{1em} \text{for } (j = 0; j < M; j++) \\
\hspace{2em} c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\]

Discussion:

- Instead of fusing several loops, we now have distributed the loops :-(
- Accordingly, conditionals may be moved out of the loop \( \Rightarrow \) if-distribution ...
Warning:

Instead of using this transformation, the inner loop could also be optimized as follows:

\[
\text{for } (i = 0; i < N; i++) \\
\quad \text{for } (j = 0; j < M; j++) \\
\qquad t = 0; \\
\qquad \text{for } (k = 0; k < K; k++) \\
\qquad \quad t = t + a[i][k] \cdot b[k][j]; \\
\qquad c[i][j] = t;
\]
Idea:

If we find heavily used array elements \( a[e_1] \ldots [e_r] \) whose index expressions stay constant within the inner loop, we could instead also provide auxiliary registers :-) 

Warning:

The latter optimization prohibits the former and vice versa ...
Discussion:

- so far, the optimizations are concerned with iterations over arrays.
- Cache-aware organization of other data-structures is possible, but in general not fully automatic ...

Example: **Stacks**
Advantage:

+ The implementation is simple  :-)  
+ The operations push / pop require constant time  :-)  
+ The data-structure may grow arbitrarily  :-)  

Disadvantage:

— The individual list objects may be arbitrarily dispersed over the memory  :-(  

Alternative:

Advantage:

+ The implementation is also simple :-) 
+ The operations push / pop still require constant time :-) 
+ The data are consequitively allocated; stack oscillations are typically small

⇒⇒⇒更好的Cache behavior !!!
Disadvantage:

- The data-structure is bounded  

Improvement:

- If the array is full, replace it with another of double size !!!
- If the array drops empty to a quarter, halve the array again !!!

⇒ The extra amortized costs are constant  
⇒ The implementation is no longer so trivial
Discussion:

→ The same idea also works for queues :-) 

→ Other data-structures are attempted to organize blockwise.

Problem: how can accesses be organized such that they refer mostly to the same block ???

⇒ Algorithms for external data
2. Stack Allocation instead of Heap Allocation

Problem:

- Programming languages such as Java allocate all data-structures in the heap — even if they are only used within the current method :-(
- If no reference to these data survives the call, we want to allocate these on the stack :-) 

⇒ Escape Analysis
Idea:

Determine points-to information.
Determine if a created object is possibly reachable from the outside...

Example: Our Pointer Language

\[
x = \text{new}();
y = \text{new}();
x[A] = y;
z = y;
\text{ret} = z;
\]

... could be a possible method body ;-)
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`; or
- are reachable from global variables.

... in the Example:

```
x = new();
y = new();
x[A] = y;
z = y;
ret = [z];
```
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as \texttt{ret}; or
- are \texttt{reachable} from global variables.

... in the Example:

\begin{verbatim}
  x = new();
  y = new();
  x[A] = y;
  z = y;
  ret = z;
\end{verbatim}
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`; or
- are reachable from global variables.

... in the Example:

```c
x = new();
y = new();
x[A] = y;
z = y;
ret = z;
```
Accessible from the outside world are memory blocks which:

- are assigned to a global variable such as `ret`; or
- are reachable from global variables.

... in the Example:

\[
\begin{align*}
\text{x} &= \text{new}(); \\
\text{y} &= \text{new}(); \\
\text{x}[A] &= \text{y}; \\
\text{z} &= \text{y}; \\
\text{ret} &= \text{z};
\end{align*}
\]
We conclude:

- The objects which have been allocated by the first \texttt{new()} may never escape.
- They can be allocated on the stack :-)

Warning:

This is only \texttt{meaningful} if only few such objects are allocated during a method call :-(

If a local \texttt{new()} occurs within a loop, we still may allocate the objects in the heap ;-)
Extension: Procedures

- We require an interprocedural points-to analysis :-)
- We know the whole program, we can, e.g., merge the control-flow graphs of all procedures into one and compute the points-to information for this.
- **Warning:** If we always use the same global variables \( y_1, y_2, \ldots \) for (the simulation of) parameter passing, the computed information is necessarily imprecise :-(
- If the whole program is not known, we must assume that each reference which is known to a procedure escapes :-((