Results:

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

Results:

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

Fischer, Rabin,1974 : $\text{PSAT} \geq \text{NTIME}(2^{2^{c \cdot n}})$

3.3 Improving the Memory Layout

Goal:

- Better utilization of caches

  $\Longrightarrow$ reduction of the number of cache misses

- Reduction of allocation/de-allocation costs

  $\Longrightarrow$ replacing heap allocation by stack allocation

  $\Longrightarrow$ support to free superfluous heap objects

- Reduction of access costs

  $\Longrightarrow$ 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:

$\rightarrow$ Reorganize the data accesses !

$\rightarrow$ Reorganize the data !

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

Example:

$$
\begin{array}{l}
\textsf{for } (j = 1; j < n; j{+}{+}) \\
\quad \textsf{for } (i = 1; i < m; i{+}{+}) \\
\quad\quad a[i][j] = a[i-1][j-1] + a[i][j];
\end{array}
$$

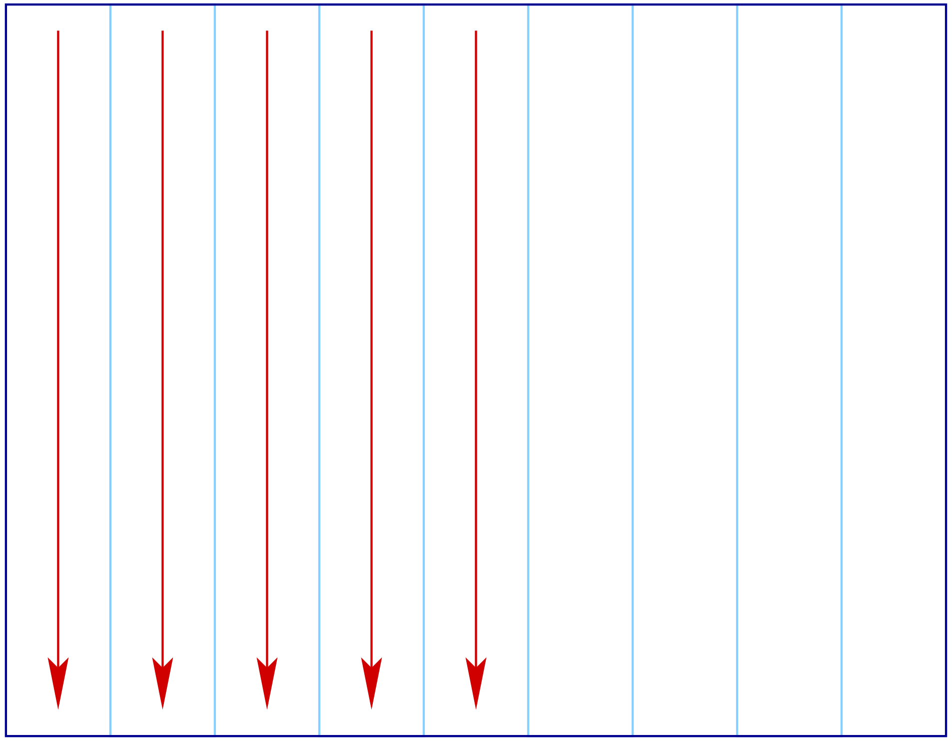
$\Longrightarrow$ At first, always iterate over the rows!

$\Longrightarrow$ Exchange the ordering of the iterations:

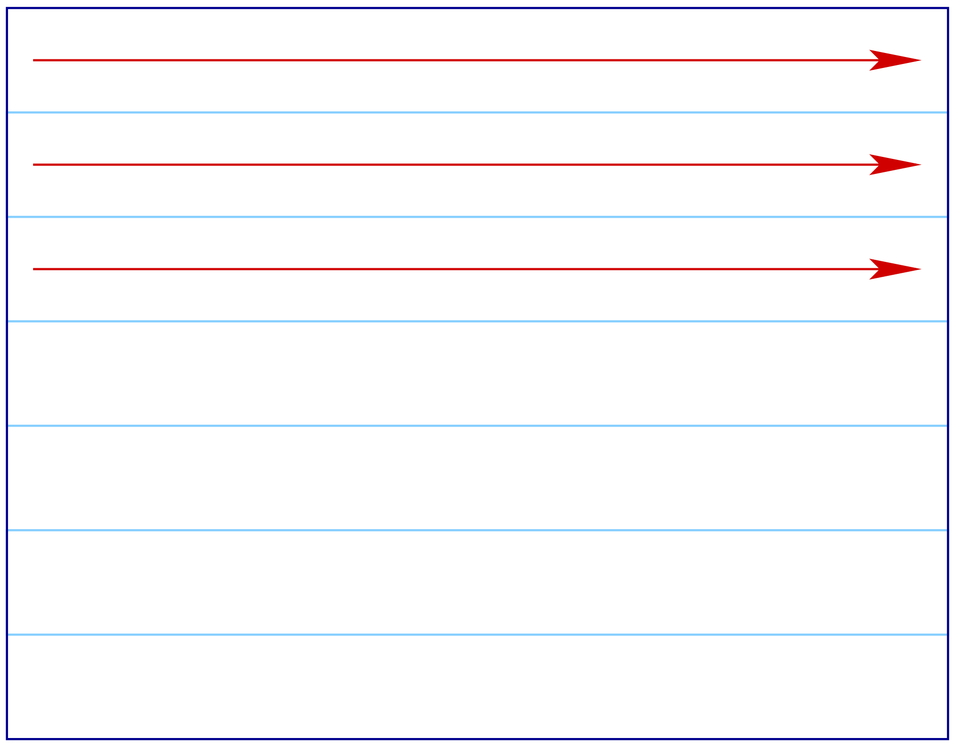
$$
\begin{array}{l}
\textsf{for} \;\; (i = 1; i < m; i{+}{+}) \\
\quad \textsf{for} \;\; (j = 1; j < n; j{+}{+}) \\
\qquad a[i][j] = a[i-1][j-1] + a[i][j];
\end{array}
$$

When is this permitted???

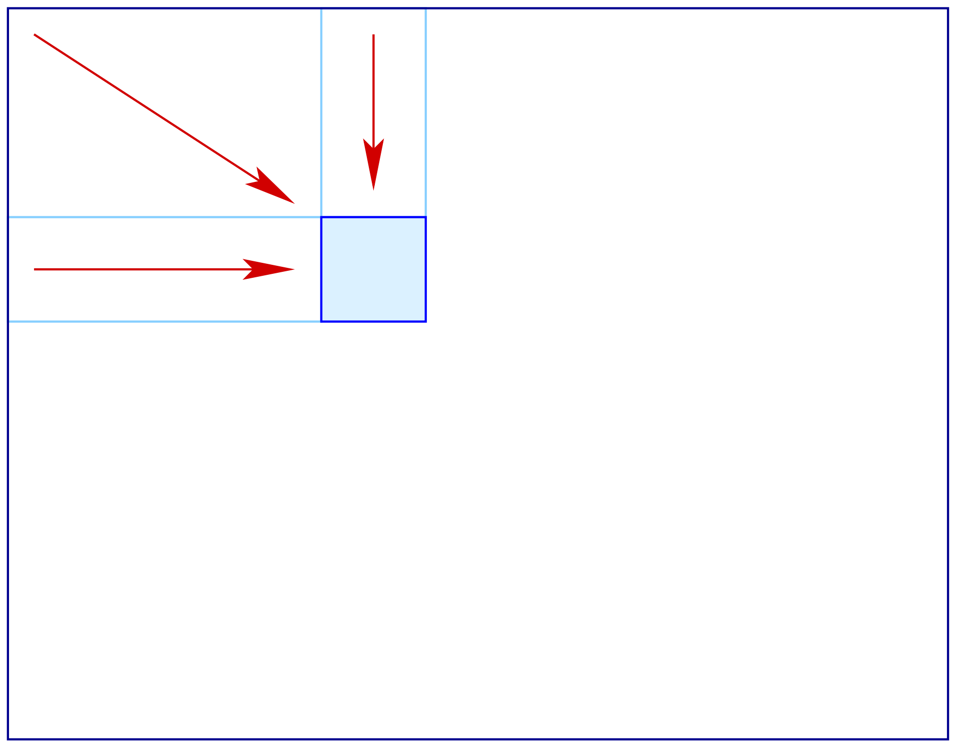
Iteration Scheme: before:



Iteration Scheme: after:



Iteration Scheme: allowed dependencies:



In our case, we must check that the following equation systems have no solution:

Write		Read
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_1	$\leq$	i_2
j_2	$\leq$	j_1
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_2	$\leq$	i_1
j_1	$\leq$	j_2

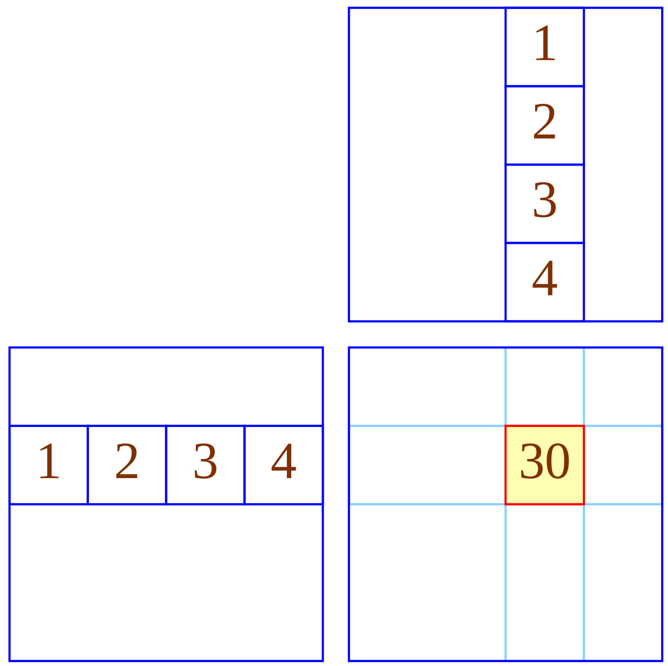
The first implies: $j_2 \leq j_2 - 1$ Hurra!

The second implies: $i_2 \leq i_2 - 1$ Hurra!

Example: Matrix-Matrix Multiplication

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\quad\quad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{array}
$$

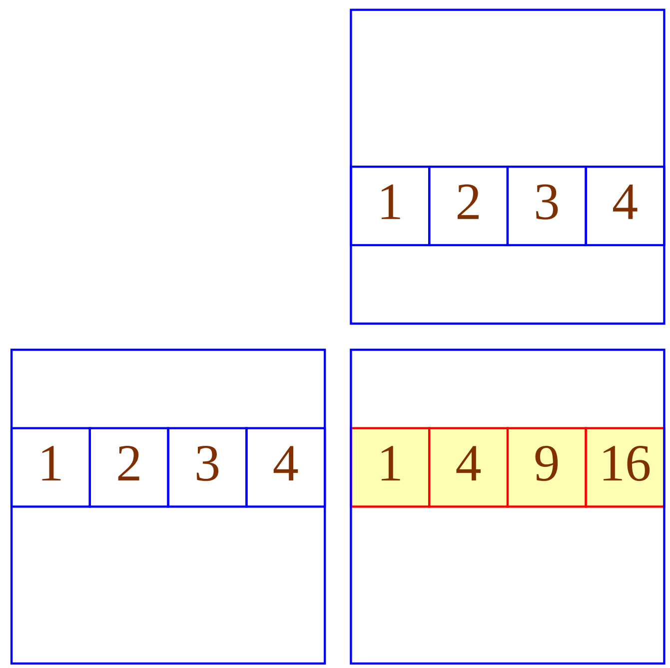
Over $b[][]$ the iteration is columnwise :-(


1
2
3
4
1
2
3
4
30

Exchange the two inner loops:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\quad\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];
\end{array}
$$

Is this permitted ???


1 2 3 4
1 2 3 4
1 4 9 16

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 ...

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\quad\quad c[i][j] = 0; \\
\quad\quad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\quad \}
\end{array}
$$

- Now, the two iterations can no longer be exchanged :-(
- The iteration over j, however, can be duplicated ...

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+})\ \ \{ \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ c[i][j] = 0; \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Correctness:

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

$\Longrightarrow$ The ordering of the write accesses to a memory cell may not be changed :-)

We obtain:

$$
\begin{array}{l}
\text{for } (i = 0; i < N; i{+}{+}) \ \{ \\
\qquad \text{for } (j = 0; j < M; j{+}{+}) \ \ c[i][j] = 0; \\
\qquad \text{for } (k = 0; k < K; k{+}{+}) \\
\qquad\qquad \text{for } (j = 0; j < M; j{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Discussion:

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

Warning:

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

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\qquad\qquad t = 0; \\
\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad t = t + a[i][k] \cdot b[k][j]; \\
\qquad\qquad c[i][j] = t; \\
\qquad \}
\end{array}
$$

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 :-)

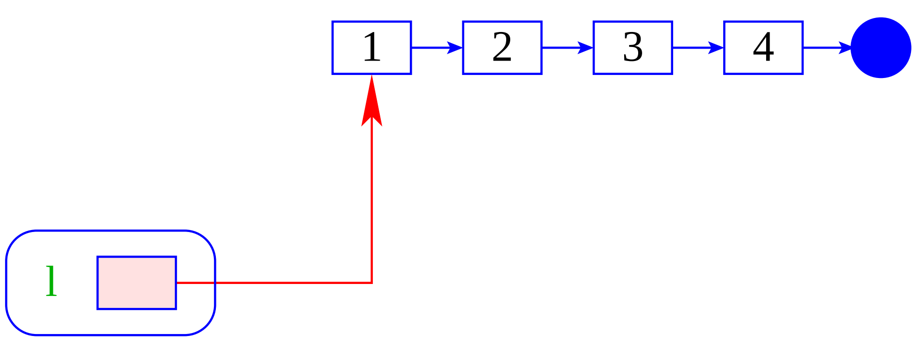
Warning:

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



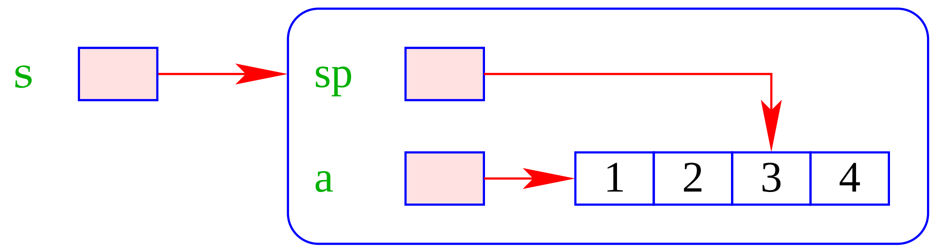
Advantage:

- $+$ 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 consequtively allocated; stack oscillations are typically small

$\Longrightarrow$ better 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 !!!

$\Longrightarrow$ The extra amortized costs are constant :-)

$\Longrightarrow$ The implementation is no longer so trivial :-}

Discussion:

$\rightarrow$ The same idea also works for queues :-)

$\rightarrow$ Other data-structures are attempted to organize blockwise.

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

$\Longrightarrow$ 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 :-)

$\Longrightarrow$ Escape Analysis

Idea:

Determine points-to information.

Determine if a created object is possibly reachable from the out side ...

Example: Our Pointer Language

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

... 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:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x[A] = y;$$
$$z = y;$$
$$\mathsf{ret} = \boxed{z};$$

Accessible from the outside world are memory blocks which:

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

... in the Example:

$$x = \textsf{new}();$$
$$y = \textsf{new}();$$
$$x[A] = y;$$
$$z = \boxed{y};$$
$$\textsf{ret} = \boxed{z};$$

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:

$$
\begin{aligned}
& x = \mathsf{new}(); \\
& y = \mathsf{new}(); \\
& \boxed{x[A]} = y; \\
& z = \boxed{y}; \\
& \mathsf{ret} = \boxed{z};
\end{aligned}
$$

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:

$$
\begin{aligned}
x &= \text{new}(); \\
y &= \boxed{\text{new}()}; \\
\boxed{x[A]} &= y; \\
z &= \boxed{y}; \\
\text{ret} &= \boxed{z};
\end{aligned}
$$

We conclude:

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

Warning:

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

If a local `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 :-((

3.4 Wrap-Up

We have considered various optimizations for improving hardware utilization.

Arrangement of the Optimizations:

- First, global restructuring of procedures/functions and of loops for better memory behavior ;-)
- Then local restructuring for better utilization of the instruction set and the processor parallelism :-)
- Then register allocation and finally,
- Peephole optimization for the final kick ...

Procedures:	Tail Recursion + Inlining Stack Allocation
Loops:	Iteration Reordering $\longrightarrow$ if-Distribution $\longrightarrow$ for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Selection Instruction Scheduling with $\longrightarrow$ Loop Unrolling $\longrightarrow$ Loop Fusion
Instructions:	Register Allocation Peephole Optimization