Script generated by TTT

Title: Seidl: Programmoptimierung (16.01.2013)

Date: Wed Jan 16 08:30:35 CET 2013

Duration: 88:52 min

Pages: 37

We obtain:

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

Discussion:

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

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

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

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

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 :-)

Warning:

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

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++) {
        t = 0;
        for (k = 0; k < K; k++)
            t = t + a[i][k] · b[k][j];
        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 ...

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 :-)

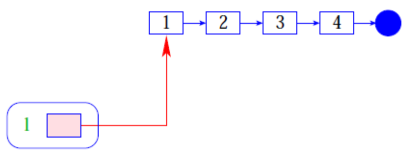
Warning:

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



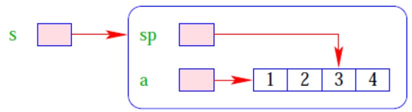
Advantage:

+ 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:

→ 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 ???

$\Longrightarrow$ 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 ret; or
- are reachable from global variables.

... in the Example:

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

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:

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

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 → if-Distribution → for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Selection Instruction Scheduling with → Loop Unrolling → Loop Fusion
Instructions:	Register Allocation Peephole Optimization

4 Optimization of Functional Programs

Example:

$$\text{let rec } \mathsf{fac}\ x \ = \ \text{if } x \leq 1 \text{ then } 1$$
$$\text{else } x \cdot \mathsf{fac}\ (x-1)$$

- There are no basic blocks :-(
- There are no loops :-(
- Virtually all functions are recursive :-((

let fac x =
    let rec work a x =
        if x = 1 then a
        else work (a*x) (x-1)
    in work 1 x

4 Optimization of Functional Programs

Example:

$$\text{let rec } \mathsf{fac}\ x \ = \ \text{if } x \leq 1 \text{ then } 1$$
$$\text{else } x \cdot \mathsf{fac}\ (x-1)$$

- There are no basic blocks :-(
- There are no loops :-(
- Virtually all functions are recursive :-((

Strategies for Optimization:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)


```
let fac x =
  let rec work a x =
    if x = 1 then a
    else work (a*x) (x-1)
  in work 1 x
```

Strategies for Optimization:

$\Longrightarrow$ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

$\Longrightarrow$ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

780

let fac x =
  let rec work a x =
    if x = 1 then a
    else work (a*x) (x-1)
  in work 1 x

Strategies for Optimization:

⟹ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

⟹ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{aligned} &\textbf{let}\ \ \mathsf{max}\ (x, y) &&= \ \textbf{if}\ x > y\ \textbf{then}\ x \\ & && \quad\ \textbf{else}\ y \\ &\textbf{let}\ \ \mathsf{abs}\ z &&= \ \mathsf{max}\ (z, -z) \end{aligned}$$

As result of the optimization we expect ...

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{aligned} &\textbf{let}\ \ \mathsf{max}\ (x, y) &&= \ \textbf{if}\ x > y\ \textbf{then}\ x \\ & && \quad\ \textbf{else}\ y \\ &\textbf{let}\ \ \mathsf{abs}\ z &&= \ \mathsf{max}\ (z, -z) \end{aligned}$$

As result of the optimization we expect ...

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{aligned} &\textbf{let}\ \ \mathsf{max}\ (x, y) &&= \ \textbf{if}\ x > y\ \textbf{then}\ x \\ & && \quad\ \textbf{else}\ y \\ &\textbf{let}\ \ \mathsf{abs}\ z &&= \ \mathsf{max}\ (z, -z) \end{aligned}$$

As result of the optimization we expect ...

$$
\begin{aligned}
&\text{let } \mathsf{max}\ (x, y) &=\ & \text{if } x > y \text{ then } x \\
& & & \text{else } y \\
&\text{let } \mathsf{abs}\ z &=\ & \quad \text{let } \ x = z \\
& & & \text{in let } \ y = -z \\
& & & \text{in } \quad \text{if } x > y \text{ then } x \\
& & & \qquad\ \ \text{else } y
\end{aligned}
$$

Discussion:

For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$
\begin{aligned}
&\text{let } \mathsf{max}\ (x, y) &=\ & \text{if } x > y \text{ then } x \\
& & & \text{else } y \\
&\text{let } \mathsf{abs}\ z &=\ & \mathsf{max}\ (z, -z)
\end{aligned}
$$

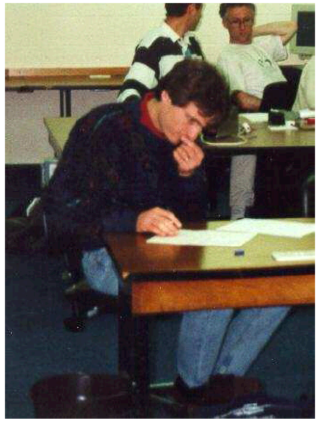
As result of the optimization we expect ...

$$
\begin{aligned}
&\text{let } \mathsf{max}\ (x, y) &=\ & \text{if } x > y \text{ then } x \\
& & & \text{else } y \\
&\text{let } \mathsf{abs}\ z &=\ & \quad \text{let } \ x = z \\
& & & \text{in let } \ y = -z \\
& & & \text{in } \quad \text{if } x > y \text{ then } x \\
& & & \qquad\ \ \text{else } y
\end{aligned}
$$

Discussion:

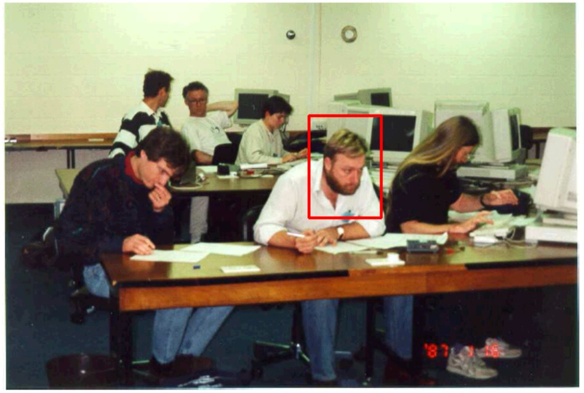
For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!



Nevin Heintze in the Australian team
of the Prolog-Programming-Contest, 1998

The complete picture:



4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lcl}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \ldots e_k \mid \mathbf{fun}\, x \to e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\
 & & \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0 \mid \\
 & & \mathbf{match}\ e_0\ \mathbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_k = e_k\ \mathbf{in}\ e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.