Script generated by TTT

Title: Seidl: Programmoptimierung (14.10.2015)

Date: Wed Oct 14 10:03:43 CEST 2015

Duration: 90:01 min

Pages: 33

Helmut Seidl

Program Optimization

TU München

Winter 2015/16

Organization

Dates:

Lecture:	Wednesday, 10:00-1~~2~~1:30	
	Thursday, 8:30-10:00	
Tutorials:	Friday, 10:00-~~12:00~~	11:30
Material:	slides, recording	
	Moodle	
	Program Analysis and Transformation	
	Springer, 2012	

Grades:

- voluntary assignments
- written exam

0 Introduction

Observation 1 Intuitive programs often are inefficient.

Example

```
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
    }
}
```

Inefficiencies

- Addresses a[i], a[j] are computed three times
- Values a[i], a[j] are loaded twice

Improvement

- Use a pointer to traverse the array a;
- store the values of a[i], a[j]!


```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
          t = aj;
          *q = ai;
          *p = t;     // t can also be
        }             // eliminated!
     }
```



```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
          t = aj;
          *q = ai;
          *p = t;     // t can also be
        }             // eliminated!
     }
```


Observation 2

Higher programming languages (even C) abstract from hardware and efficiency.

It is up to the compiler to adapt intuitively written program to hardware.

Examples

... Filling of delay slots;

... Utilization of special instructions;

... Re-organization of memory accesses for better cache behavior;

... Removal of (useless) overflow/range checks.

0 Introduction

Observation 1 Intuitive programs often are inefficient.

Example

```
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
    }
}
```


Observation 2

Higher programming languages (even C) abstract from hardware and efficiency.

It is up to the compiler to adapt intuitively written program to hardware.

Examples

... Filling of delay slots;

... Utilization of special instructions;

... Re-organization of memory accesses for better cache behavior;

... Removal of (useless) overflow/range checks.

Observation 3

Programm-Improvements need not always be correct !

Example

`y = f() + f();` $\Longrightarrow$ `y = 2 * f();`

Idea: Save second evaluation of `f()` ...

Observation 3

Programm-Improvements need not always be correct !

Example

`y = f() + f();` $\Longrightarrow$ `y = 2 * f();`

Idea: Save the second evaluation of `f()` ???

Problem: The second evaluation may return a result different from the first; (e.g., because `f()` reads from the input

Observation 3

Programm-Improvements need not always be correct !

Example

```
y = f() + f();    ⟹    y = 2 * f();
```

Idea: Save the second evaluation of `f()` ???

Problem: The second evaluation may return a result different from the first; (e.g., because `f()` reads from the input

Consequences

$\Longrightarrow$ Optimizations have assumptions.

$\Longrightarrow$ The assumption must be:

- formalized,
- checked !

$\Longrightarrow$ It must be proven that the optimization is correct, i.e., preserves the semantics !!!

Observation 4

Optimization techniques depend on the programming language:

$\rightarrow$ which inefficiencies occur;

$\rightarrow$ how analyzable programs are;

$\rightarrow$ how difficult/impossible it is to prove correctness ...

Example Java

Unavoidable Inefficiencies

$*$ Array-bound checks;

$*$ Dynamic method invocation;

$*$ Bombastic object organization ...

Analyzability

$+$ no pointer arithmetic;

$+$ no pointer into the stack;

$-$ dynamic class loading;

$-$ reflection, exceptions, threads, ...

Correctness proofs

+ more or less well-defined semantics;
− features, features, features;
− libraries with changing behavior ...

... in this course:

a simple imperative programming language with

• variables	//	registers
• $R = e;$	//	assignments
• $R = M[e];$	//	loads
• $M[e_1] = e_2;$	//	stores
• if (e) s_1 else s_2	//	conditional branching
• goto L;	//	no loops

Remark

- For the beginning, we omit procedures ...
- External procedures are taken into account through a statement $f()$ for an unknown procedure f.
  $\Longrightarrow$ intra-procedural
  $\Longrightarrow$ kind of an intermediate language in which (almost) everything can be translated.

Example `swap()`

$$
\begin{array}{lllll}
0: & A_1 = A_0 + 1 * i; & & // & A_0 == \&a \\
1: & R_1 = M[A_1]; & & // & R_1 == a[i] \\
2: & A_2 = \boxed{A_0 + 1 * j;} & & & \\
3: & R_2 = \boxed{M[A_2];} & & // & R_2 == a[j] \\
4: & \text{if } (R_1 > R_2)\ \{ & & & \\
5: & \quad A_3 = \boxed{A_0 + 1 * j;} & & & \\
6: & \quad t = \boxed{M[A_3];} & & & \\
7: & \quad A_4 = A_0 + 1 * j; & & & \\
8: & \quad A_5 = A_0 + 1 * i; & & & \\
9: & \quad R_3 = M[A_5]; & & & \\
10: & \quad M[A_4] = R_3; & & & \\
11: & \quad A_6 = A_0 + 1 * i; & & & \\
12: & \quad M[A_6] = t; & & & \\
 & \quad \} & & &
\end{array}
$$

Optimization 1: $1 * R \Longrightarrow R$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

$$\begin{array}{lllll}
A_1 & = & A_0 + \boxed{1 *}\, i; & // & A_0 == \&a \\
R_1 & = & M[A_1]; & // & R_1 == a[i] \\
A_2 & = & A_0 + 1 * j; & & \\
R_2 & = & M[A_2]; & // & R_2 == a[j] \\
\text{if } (R_1 > R_2)\ \{ & & & & \\
\quad A_3 & = & A_0 + 1 * j; & & \\
\quad t & = & M[A_3]; & & \\
\quad A_4 & = & A_0 + 1 * j; & & \\
\quad A_5 & = & A_0 + 1 * i; & & \\
\quad R_3 & = & M[A_5]; & & \\
\quad M[A_4] & = & R_3; & & \\
\quad A_6 & = & A_0 + 1 * i; & & \\
\quad M[A_6] & = & t; & & \\
\quad \} & & & &
\end{array}$$

Optimization 1: $\boxed{1 * R \Longrightarrow R}$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

$$\begin{array}{lllll}
A_1 & = & A_0 + 1 * i; & // & A_0 == \&a \\
R_1 & = & M[A_1]; & // & R_1 == a[i] \\
A_2 & = & A_0 + 1 * j; & & \\
R_2 & = & M[A_2]; & // & R_2 == a[j] \\
\text{if } (R_1 > R_2)\ \{ & & & & \\
\quad A_3 & = & A_0 + 1 * j; & & \\
\quad t & = & M[A_3]; & & \\
\quad A_4 & = & A_0 + 1 * j; & & \\
\quad A_5 & = & A_0 + 1 * i; & & \\
\quad R_3 & = & M[A_5]; & & \\
\quad M[A_4] & = & R_3; & & \\
\quad A_6 & = & A_0 + 1 * i; & & \\
\quad M[A_6] & = & t; & & \\
\quad \} & & & &
\end{array}$$

Optimization 1: $1 * R \quad \Longrightarrow \quad R$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Optimization 1: $1 * R \quad \Longrightarrow \quad R$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

By this, we obtain:

$$
\begin{array}{lll}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\text{if } (R_1 > R_2) \ \{ & & \\
\quad \cancel{t} & \cancel{=} & \cancel{R_2;} \\
\quad M[A_2] & = & R_1; \\
\quad M[A_1] & = & \cancel{t}; \ R_2 \\
\quad \} & &
\end{array}
$$

Optimization 3: Contraction of chains of assignments

Gain

	before	after
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$=$	6	2

1 Removing superfluous computations

1.1 Repeated computations

Idea

If the same value is computed repeatedly, then

→ store it after the first computation;

→ replace every further computation through a look-up!

⟹ Availability of expressions

⟹ Memoization

Problem: Identify repeated computations!

Example

$$\begin{array}{llll} & z & = & 1; \\ & y & = & M[17]; \\ A: & x_1 & = & \boxed{y+z}; \\ & & \dots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Problem: Identify repeated computations!

Example

$$\begin{array}{llll} & z & = & 1; \\ & y & = & M[17]; \\ A: & x_1 & = & \boxed{y+z}; \\ & & \dots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Remark

B is a repeated computation of the value of $\boxed{y+z}$, if:

(1) A is always executed before B; and

(2) y and z at B have the same values as at A.

⟹ We need:

→ an operational semantics;

→ a method which identifies at least some repeated computations ...

Problem: Identify repeated computations!

Example

$$\begin{array}{rrcl} & z & = & 1; \\ & y & = & M[17]; \\ A: & x_1 & = & \boxed{y+z}; \\ & & \dots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Remark

B is a repeated computation of the value of $\boxed{y+z}$, if:

(1) A is always executed before B; and

(2) y and z at B have the same values as at A.

$\Longrightarrow$ We need:

$\rightarrow$ an operational semantics;

$\rightarrow$ a method which identifies at least some repeated computations ...