Script generated by TTT

Title: Seidl: Programmoptimierung (16.10.2013)

Date: Wed Oct 16 08:30:34 CEST 2013

Duration: 88:14 min

Pages: 32

Helmut Seidl

Program Optimization

TU München

Winter 2013/14

Organization

Dates:

Lecture:	Monday, 14:00-15:30
	Wednesday, 8:30-10:00
Tutorials:	Tuesday/Wednesday, 10:00-12:00
	Stefan Schulze Friehlinghaus: `schulzef@in.tum.de`
Material:	slides, recording :-)
	Moodle
	Program Analysis and Transformation
	Springer, 2012

Grades:

- Bonus for homeworks
- written exam

Proposed Content:

1. Avoiding redundant computations
   - → available expressions
   - → constant propagation/array-bound checks
   - → code motion
2. Replacing expensive with cheaper computations
   - → peep hole optimization
   - → inlining
   - → reduction of strength

     ...

Proposed Content:

1. Avoiding redundant computations
   - → available expressions
   - → constant propagation/array-bound checks
   - → code motion
2. Replacing expensive with cheaper computations
   - → peep hole optimization
   - → inlining
   - → reduction of strength

     ...

3. Exploiting Hardware
   - → Instruction selection
   - → Register allocation
   - → Scheduling
   - → Memory management

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!
}
```


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;
    }
}
```



```
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.

Observation 3:

Programm-Improvements need not always be correct :-(

Example:

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

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

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:

→ which inefficiencies occur;
→ how analyzable programs are;
→ 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, ...

· weak memory models

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 :-)

Note:

- 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()`

... 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 :-)

Note:

- 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()`

17

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

$$
\begin{array}{llll}
A_1 & = & A_0 + \cancel{1} * i; & // \quad A_0 == \&a \\
R_1 & = & M[A_1]; & // \quad R_1 == a[i] \\
A_2 & = & A_0 + \cancel{1} * j; & \\
R_2 & = & M[A_2]; & // \quad R_2 == a[j] \\
\multicolumn{4}{l}{\text{if } (R_1 > R_2)\ \{} \\
 & A_3 & = & A_0 + 1 * j; \\
 & t & = & M[A_3]; \\
 & A_4 & = & A_0 + \cancel{1} * j; \\
 & A_5 & = & A_0 + \cancel{1} * i; \\
 & R_3 & = & M[A_5]; \\
 & M[A_4] & = & R_3; \\
 & A_6 & = & A_0 + \cancel{1} * i; \\
 & M[A_6] & = & t; \\
 & \} & &
\end{array}
$$

$$
\begin{array}{llll}
A_1 & = & A_0 + 1 * i; & // \quad A_0 == \&a \\
R_1 & = & M[A_1]; & // \quad R_1 == a[i] \\
A_2 & = & A_0 + 1 * j; & \\
R_2 & = & M[A_2]; & // \quad R_2 == a[j] \\
\multicolumn{4}{l}{\text{if } (R_1 > R_2)\ \{} \\
 & A_3 & = & A_0 + 1 * j; \\
 & t & = & M[A_3]; \\
 & A_4 & = & A_0 + 1 * j; \\
 & A_5 & = & A_0 + 1 * i; \\
 & R_3 & = & M[A_5]; \\
 & M[A_4] & = & R_3; \\
 & A_6 & = & A_0 + 1 * i; \\
 & M[A_6] & = & t; \\
 & \} & &
\end{array}
$$

Optimization 1: $1 * R \implies R$

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: $1 * R \implies R$

Optimization 2: Reuse of subexpressions

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

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

$$R_1 == R_3$$

$$
\begin{array}{lcll}
A_1 & = & A_0 + 1 * i; & // \quad A_0 == \&a \\
R_1 & = & M[A_1]; & // \quad R_1 == a[i] \\
A_2 & = & A_0 + 1 * j; & \\
R_2 & = & M[A_2]; & // \quad R_2 == a[j] \\
\multicolumn{4}{l}{\text{if } (R_1 > R_2) \ \{} \\
\quad A_3 & = & A_0 + 1 * j; & \\
\quad t & = & M[A_3]; & \\
\quad A_4 & = & A_0 + 1 * j; & \\
\quad A_5 & = & A_0 + 1 * i; & \\
\quad R_3 & = & M[A_5]; & \\
\quad M[A_4] & = & R_3; & \\
\quad A_6 & = & A_0 + 1 * i; & \\
\quad M[A_6] & = & t; & \\
\quad \} & & &
\end{array}
$$

By this, we obtain:

$$
\begin{array}{lcl}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\multicolumn{3}{l}{\text{if } (R_1 > R_2) \ \{} \\
\quad \cancel{t} & \cancel{=} & \cancel{R_2;} \\
\quad M[A_2] & = & R_1; \\
\quad M[A_1] & = & \cancel{t}\ R_2 \\
\quad \} & &
\end{array}
$$

Optimization 3: Contraction of chains of assignments :-)

Gain:

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

redundant

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!

$\Longrightarrow$ Availability of expressions

$\Longrightarrow$ Memoization

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!

$\Longrightarrow$ Availability of expressions

$\Longrightarrow$ Memoization