Script generated by TTT

Title: Seidl: Programmoptimierung (22.10.2012)

Date: Mon Oct 22 14:02:05 CEST 2012

Duration: 90:47 min

Pages: 43

Helmut Seidl

Program Optimization

TU München

Winter 2012/13

Organization

Dates: Lecture: Monday, 14:00-15:30
Wednesday, 8:30-10:00

Tutorials: Tuesday/Wednesday, 10:00-12:00
Kalmer Apinis: apinis@in.tum.de

Material: slides, recording :-)
Moodle
Program Analysis and Transformation
Springer, 2012

Grades:

- Bonus for homeworks
- written exam

Organization

Dates:

Lecture: Monday, 14:00-15:30
Wednesday, 8:30-10:00

Tutorials: Tuesday/Wednesday, 10:00-12:00
Kalmer Apinis: apinis@in.tum.de

Material: slides, recording :-)
Moodle
Program Analysis and Transformation
Springer, 2012

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!
     }
```



```
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!
     }
```

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]`!

Observation 3:

Programm-Improvements need not always be correct :-(

Example:

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

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

Consequences:

⟹ Optimizations have assumptions.

⟹ The assumption must be:

- formalized,
- checked :-)

⟹ 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

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 :-)


```
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!
}
```

\+ more or less spec of semantics

Correctness proofs:

\+ more or less well-defined semantics;

– features, features, features;

– libraries with changing behavior ...

Correctness proofs:

\+ more or less well-defined semantics;

– features, features, features;

– libraries with changing behavior ...

15

... 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()`

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

Optimization 1: $1 * R \Longrightarrow R$

Optimization 2: Reuse of subexpressions

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

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

$$R_1 == R_3$$

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

Optimization 1: $1 * R \implies R$

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

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

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

$$R_1 == R_3$$

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

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]`!

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]`!

Optimization 3: Contraction of chains of assignments :-)

Gain:

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

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:

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

Note:

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 :-)

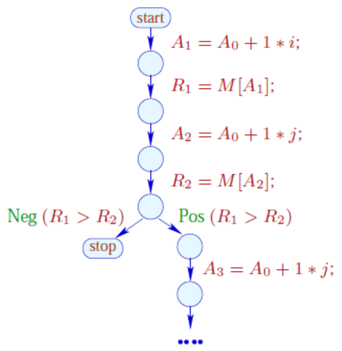
→ a method which identifies at least some repeated computations ...

Background 1: An Operational Semantics

we choose a small-step operational approach.

Programs are represented as control-flow graphs.

In the example:

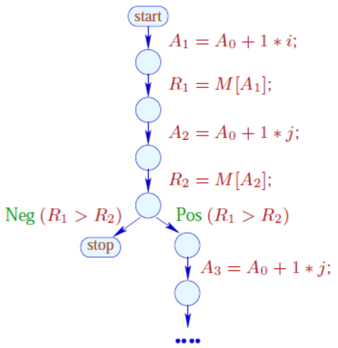


Background 1: An Operational Semantics

we choose a small-step operational approach.

Programs are represented as control-flow graphs.

In the example:



Thereby, represent:

vertex	program point
start	programm start
stop	program exit
edge	step of computation

Thereby, represent:

vertex	program point
start	programm start
stop	program exit
edge	step of computation

Edge Labelings:

Test : Pos (e) or Neg (e)

Assignment : $R = e;$

Load : $R = M[e];$

Store : $M[e_1] = e_2;$

Nop : ;

Computations follow paths.

Computations transform the current state

$$s = (\rho, \mu)$$

where:

$\rho : \mathit{Vars} \to \mathbf{int}$	contents of registers
$\mu : \mathbb{N} \to \mathbf{int}$	contents of storage

Every edge $k = (u, lab, v)$ defines a partial transformation

$$[\![k]\!] = [\![lab]\!]$$

of the state:

$$[\![;]\!]\,(\rho,\mu) = (\rho,\mu)$$

$$[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho \neq 0$$
$$[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho = 0$$

$$[\![;]\!]\,(\rho,\mu) = (\rho,\mu)$$

$$[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho \neq 0$$
$$[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho = 0$$

// $[\![e]\!]$: evaluation of the expression e, e.g.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![;]\!]\,(\rho,\mu) = (\rho,\mu)$$

$$[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho \neq 0$$
$$[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) = (\rho,\mu) \quad \text{if } [\![e]\!]\,\rho = 0$$

// $[\![e]\!]$: evaluation of the expression e, e.g.

// $[\![x+y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

$$[\![R = e;]\!]\,(\rho,\mu) = (\rho \oplus \{R \mapsto [\![e]\!]\,\rho\}, \mu)$$

// where "$\oplus$" modifies a mapping at a given argument

$$[\![R = M[e];]\!]\,(\rho,\mu) = (\rho \oplus \{R \mapsto \mu([\![e]\!]\,\rho))\}, \mu)$$

$$[\![M[e_1] = e_2;]\!]\,(\rho,\mu) = (\rho, \mu \oplus \{[\![e_1]\!]\,\rho \mapsto [\![e_2]\!]\,\rho\})$$

Example:

$[\![x = x + 1;]\!]\,(\{x \mapsto 5\}, \mu) = (\rho, \mu)$ where:

$$\begin{aligned} \rho &= \{x \mapsto 5\} \oplus \{x \mapsto [\![x+1]\!]\,\{x \mapsto 5\}\} \\ &= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ &= \{x \mapsto 6\} \end{aligned}$$