Helmut Seidl

Program Optimization

TU München

Winter 2011/12

Organization

Dates:	Lecture:	Monday, 12:30-14:00
		Wednesday, 12:30-14:00
	Tutorials:	Thursday, 12:30-14:00
		Kalmer Apinis: `apinis@in.tum.de`
	Material:	slides, recording :-)
		simulator environment
		Programmanalyse und Transformation
		Springer, 2010

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

    ...

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!
     }
```

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, ...

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()`

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

Optimization 1: $1 * R \Longrightarrow R$

Optimization 2: Reuse of subexpressions

$$A_1 == A_5 == A_6$$

$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$

$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

By this, we obtain:

```
$A_1 = A_0 + i;$
$R_1 = M[A_1];$
$A_2 = A_0 + j;$
$R_2 = M[A_2];$
if $(R_1 > R_2)$ {
        $t = R_2;$
        $M[A_2] = R_1;$
        $M[A_1] = t;$
        }
```

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!

$\Longrightarrow$ Availability of expressions

$\Longrightarrow$ Memoization

Problem: Identify repeated computations!

Example:

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

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 :-)

$\Longrightarrow$ We need:

$\rightarrow$ an operational semantics :-)

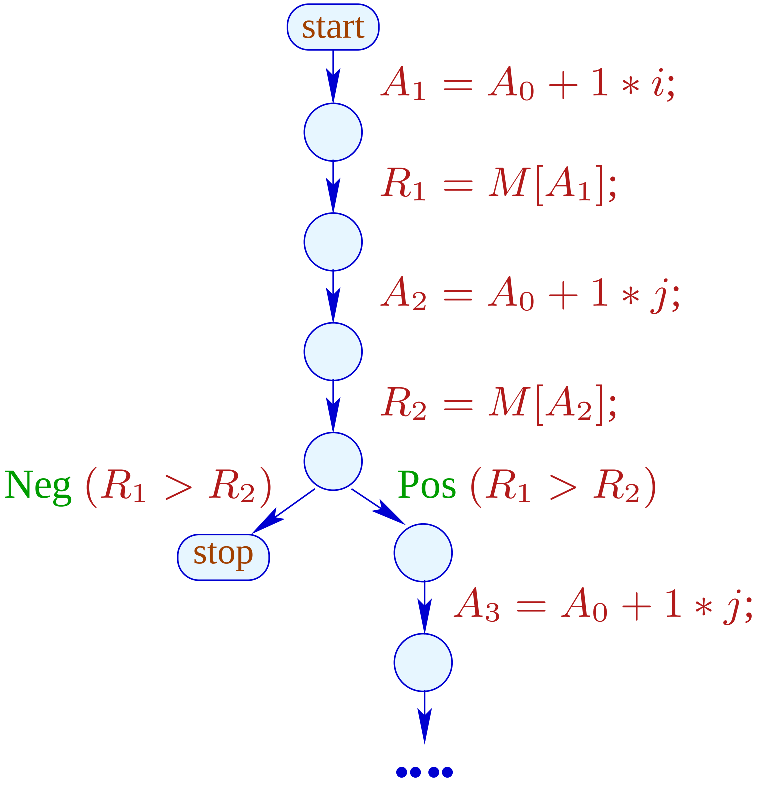
$\rightarrow$ a method which identifies at least some repeated computations ...

Background 1: An Operational Semantics

we choose a small-step operational approach.

Programs are represented as control-flow graphs.

In the example:



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:

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

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

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

of the state:

$$\llbracket ; \rrbracket\,(\rho,\mu) \quad = \quad (\rho,\mu)$$

$$\begin{array}{lcll}
\llbracket \mathrm{Pos}\,(e) \rrbracket\,(\rho,\mu) & = & (\rho,\mu) & \text{if } \llbracket e \rrbracket\,\rho \neq 0 \\
\llbracket \mathrm{Neg}\,(e) \rrbracket\,(\rho,\mu) & = & (\rho,\mu) & \text{if } \llbracket e \rrbracket\,\rho = 0
\end{array}$$

$$\begin{array}{lcll}
[\![;]\!]\,(\rho,\mu) & = & (\rho,\mu) & \\
\\
[\![\mathrm{Pos}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho \neq 0 \\
[\![\mathrm{Neg}\,(e)]\!]\,(\rho,\mu) & = & (\rho,\mu) & \text{if } [\![e]\!]\,\rho = 0
\end{array}$$

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

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

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

$$\llbracket ; \rrbracket\,(\rho, \mu) \quad = \quad (\rho, \mu)$$

$$\begin{array}{lcll} \llbracket \mathrm{Pos}\,(e) \rrbracket\,(\rho, \mu) & = & (\rho, \mu) & \text{if } \llbracket e \rrbracket\,\rho \neq 0 \\ \llbracket \mathrm{Neg}\,(e) \rrbracket\,(\rho, \mu) & = & (\rho, \mu) & \text{if } \llbracket e \rrbracket\,\rho = 0 \end{array}$$

// $\llbracket e \rrbracket$: evaluation of the expression e, e.g.

// $\llbracket x + y \rrbracket\,\{x \mapsto 7, y \mapsto -1\} = 6$

// $\llbracket !(x == 4) \rrbracket\,\{x \mapsto 5\} = 1$

$$\llbracket R = e; \rrbracket\,(\rho, \mu) \quad = \quad (\boxed{\rho \oplus \{R \mapsto \llbracket e \rrbracket\,\rho\}}, \mu)$$

// where “$\oplus$” modifies a mapping at a given argument

$$\begin{aligned} \llbracket R = M[e]; \rrbracket\,(\rho, \mu) &= (\boxed{\rho \oplus \{R \mapsto \mu(\llbracket e \rrbracket\,\rho))\}}, \mu) \\ \llbracket M[e_1] = e_2; \rrbracket\,(\rho, \mu) &= (\rho, \boxed{\mu \oplus \{\llbracket e_1 \rrbracket\,\rho \mapsto \llbracket e_2 \rrbracket\,\rho\}}) \end{aligned}$$

Example:

$\llbracket x = x + 1; \rrbracket\,(\{x \mapsto 5\}, \mu) = (\rho, \mu)$ **where:**

$$\begin{aligned} \rho &= \{x \mapsto 5\} \oplus \{x \mapsto \llbracket x + 1 \rrbracket\,\{x \mapsto 5\}\} \\ &= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ &= \{x \mapsto 6\} \end{aligned}$$

A path $\pi = k_1 k_2 \dots k_m$ is a computation for the state s if:

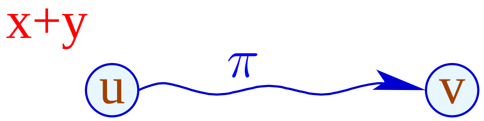
$$s \in def\,([\![k_m]\!] \circ \dots \circ [\![k_1]\!])$$

The result of the computation is:

$$[\![\pi]\!]\, s = ([\![k_m]\!] \circ \dots \circ [\![k_1]\!])\, s$$

Application:

Assume that we have computed the value of $x + y$ at program point u:



We perform a computation along path π and reach v where we evaluate again $x + y$ ...

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

More generally:

Assume that the values of the expressions $A = \{e_1, \ldots, e_r\}$ are available at u.

Idea:

If x and y have not been modified in π, then evaluation of $x + y$ at v must return the same value as evaluation at u :-)

We can check this property at every edge in π :-}

More generally:

Assume that the values of the expressions $A = \{e_1, \ldots, e_r\}$ are available at u.

Every edge k transforms this set into a set $[\![k]\!]^\sharp\, A$ of expressions whose values are available after execution of k ...

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on the label lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$

... which transformations can be composed to the effect of a path $\pi = k_1 \ldots k_r$:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on the label lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lclcl} [\![;]\!]^\sharp\, A & = & A & & \\ [\![Pos(e)]\!]^\sharp\, A & = & [\![Neg(e)]\!]^\sharp\, A & = & A \cup \{e\} \\ [\![x = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x & & \text{where} \end{array}$$

$Expr_x$ all expressions which contain x

$$\begin{aligned} [\![x = M[e];]\!]^\sharp\, A &= (A \cup \{e\}) \backslash Expr_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A &= A \cup \{e_1, e_2\} \end{aligned}$$

$$\begin{array}{lll} [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & A \cup \{e_1, e_2\} \end{array}$$

By that, every path can be analyzed :-)

A given program may admit several paths :-(

For any given input, another path may be chosen :-((

$$\begin{array}{lcl} [\![x = M[e];]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash \mathit{Expr}_x \\ [\![M[e_1] = e_2;]\!]^\sharp\, A & = & A \cup \{e_1, e_2\} \end{array}$$

By that, every path can be analyzed :-)

A given program may admit several paths :-(

For any given input, another path may be chosen :-((

$\Longrightarrow$ We require the set:

$$\mathcal{A}[v] \quad = \quad \bigcap \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : \mathit{start} \rightarrow^* v \}$$

Concretely:

→ We consider all paths π which reach v.

→ For every path π, we determine the set of expressions which are available along π.

→ Initially at program start, nothing is available :-)

→ We compute the intersection $\Longrightarrow$ safe information

Concretely:

- → We consider all paths π which reach v.
- → For every path π, we determine the set of expressions which are available along π.
- → Initially at program start, nothing is available :-)
- → We compute the intersection $\Longrightarrow$ safe information

How do we exploit this information ???

Transformation 1.1:

We provide novel registers T_e as storage for the e: