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:

```c
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:

\[ y = f() + f(); \quad \implies \quad y = 2 \times f(); \]

Idea: Save second evaluation of \( f() \)  ...
Observation 3:

Programm-Improvements need not always be correct :-(

Example:

\[ y = f() + f(); \implies y = 2 * f(); \]

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:

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

Optimization 1: \[ 1 \times R \rightarrow R \]

Optimization 2: Reuse of subexpressions

\begin{align*}
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
\end{align*}
By this, we obtain:

\[ A_1 = A_0 + i; \]
\[ R_1 = M[A_1]; \]
\[ A_2 = A_0 + j; \]
\[ R_2 = M[A_2]; \]
\[ \text{if } (R_1 > R_2) \{ \]
\[ t = R_2; \]
\[ M[A_2] = R_1; \]
\[ M[A_1] = t; \]
\[ \} \]
**Optimization 3:** Contraction of chains of assignments :-)

**Gain:**

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>
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:

\[ z = 1; \]
\[ y = M[17]; \]
\[ A: \quad x_1 = y + z; \]
\[ \quad \ldots \]
\[ B: \quad x_2 = y + z; \]
Note:

$B$ is a repeated computation of the value of $y + z$, 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:

\[ A_1 = A_0 + 1 \times i; \]
\[ R_1 = M[A_1]; \]
\[ A_2 = A_0 + 1 \times j; \]
\[ R_2 = M[A_2]; \]
\[ \text{Neg (} R_1 > R_2 \text{)} \]
\[ \text{Pos (} R_1 > R_2 \text{)} \]
\[ A_3 = A_0 + 1 \times j; \]
Thereby, represent:

<table>
<thead>
<tr>
<th>vertex</th>
<th>program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>programm start</td>
</tr>
<tr>
<td>stop</td>
<td>program exit</td>
</tr>
<tr>
<td>edge</td>
<td>step of computation</td>
</tr>
</tbody>
</table>

Edge Labelings:
- Test : Pos $(e)$ or Neg $(e)$
- Assignment : $R = e$
- Load : $R = M[e]$
- Store : $M[e_1] = e_2$
- Nop : $; _26$
Thereby, represent:

<table>
<thead>
<tr>
<th>vertex</th>
<th>program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>programm start</td>
</tr>
<tr>
<td>stop</td>
<td>program exit</td>
</tr>
<tr>
<td>edge</td>
<td>step of computation</td>
</tr>
</tbody>
</table>

Edge Labelings:

- **Test**: Pos \((e)\) or Neg \((-e)\)
- **Assignment**: \(R = e\);
- **Load**: \(R = M[e]\);
- **Store**: \(M[e_1] = e_2\);
- **Nop**: ;

27
Computations follow paths.

Computations transform the current state

\[ s = (\rho, \mu) \]

where:

| \( \rho : \text{Vars} \rightarrow \text{int} \) | contents of registers |
| \( \mu : \mathbb{N} \rightarrow \text{int} \) | contents of storage |

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

\[ [k] = [lab] \]

of the state:
\[ (;) (\rho, \mu) = (\rho, \mu) \]

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

\[ [\text{Neg} (e)] (\rho, \mu) = (\rho, \mu) \quad \text{if } [e] \rho = 0 \]
\[
[;] (\rho, \mu) = (\rho, \mu)
\]

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

\[
[\text{Neg } (e)] (\rho, \mu) = (\rho, \mu) \quad \text{if } [e] \rho = 0
\]

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

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

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

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

// \[ [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:

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

\begin{align*}
\rho &= \{x \mapsto 5\} \oplus \{x \mapsto [x + 1] \{x \mapsto 5\}\} \\
    &= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\
    &= \{x \mapsto 6\}
\end{align*}
A path $\pi = k_1 k_2 \ldots k_m$ is a computation for the state $s$ if:

$$s \in \text{def} \left([k_m] \circ \ldots \circ [k_1]\right)$$

The result of the computation is:

$$[\pi] s = \left([k_m] \circ \ldots \circ [k_1]\right) s$$

Application:

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

$$x + y$$

We perform a computation along path $\pi$ and reach $v$ where we evaluate again $x + y$ ...
Idea:

If \( x \) and \( y \) have not been modified in \( \pi \), 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 \( \pi \) :-}
Idea:

If $x$ and $y$ have not been modified in $\pi$, 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 $\pi$ :-}

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 $\pi$, 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 $\pi$ :-}

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 $[k]^{\#} A$ of expressions whose values are available after execution of $k$ ...
... which transformations can be composed to the effect of a path \( \pi = k_1 \ldots k_r \):

\[
[\pi]^{\#} = [k_r]^{\#} \circ \ldots \circ [k_1]^{\#}
\]
... which transformations can be composed to the effect of a path \( \pi = k_1 \ldots k_r \):

\[
[\pi]^\# = [k_r]^\# \circ \ldots \circ [k_1]^\#
\]

The effect \([k]^\#\) of an edge \( k = (u, \text{lab}, v) \) only depends on the label \( \text{lab} \), i.e., \([k]^\# = [\text{lab}]^\#\)
... which transformations can be composed to the effect of a path
\( \pi = k_1 \ldots k_r \):

\[
[\pi]# = [k_r]# \circ \ldots \circ [k_1]#
\]

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

\[
[;]# A = A
\]

\[
[Pos(e)]# A = [Neg(e)]# A = A \cup \{e\}
\]

\[
[x = e;]# A = (A \cup \{e\})\setminus Expr_x \quad \text{where}
\]

\(Expr_x\) all expressions which contain \(x\)
\[ [x = M[e];] \# A \quad = \quad (A \cup \{e\}) \setminus \text{Expr}_x \]
\[ [M[e_1] = e_2;] \# A \quad = \quad A \cup \{e_1, e_2\} \]
\[ x = M[e]; \] # A = (A \cup \{e\}) \setminus \text{Expr}_x \\
[M[e_1] = e_2;] # A = A \cup \{e_1, e_2\}

By that, every path can be analyzed \:-)

A given program may admit several paths \:-(

For any given input, another path may be chosen \:-((
\[ [x = M[e];] A = (A \cup \{e\}) \setminus Expr_x \]
\[ [M[e_1] = e_2;] A = A \cup \{e_1, e_2\} \]

By that, every path can be analyzed  

A given program may admit several paths  

For any given input, another path may be chosen  

We require the set:

\[ \mathcal{A}[v] = \bigcap \{[[\pi]] \setminus \emptyset \mid \pi : start \to^* v \} \]
Concretely:

→ We consider all paths $\pi$ which reach $v$.

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

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

→ We compute the intersection $\implies$ safe information
Concretely:

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

How do we exploit this information ???
Transformation 1.1:

We provide novel registers $T_e$ as storage for the $e$:

\[
\begin{align*}
    u &\xrightarrow{x = e;} v \\
    u &\xrightarrow{T_e = e;} v \\
\end{align*}
\]