Organization

**Dates:**

**Lecture:** Monday, 12:30-14:00
Thursday, 10:15-11:45

**Tutorials:** Thursday, 16:30-18:00

Vesal Vojdani: vojdanig@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:

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

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 ;-)

16
Note:

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

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

18
Optimization 1: \[ 1 \times R \quad \Rightarrow \quad R \]

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:

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