1.8 Application: Loop-invariant Code

Example:

```c
for (i = 0; i < n; i++)
    a[i] = b + 3;
```

// The expression $b + 3$ is recomputed in every iteration

// This should be avoided
The Control-flow Graph:

\[ i = 0; \]
\[ \text{Neg}(i < n) \]
\[ y = b + 3; \]
\[ A_1 = A + i; \]
\[ M[A_1] = y; \]
\[ i = i + 1; \]
Warning: \( T = b + 3; \) may not be placed before the loop:

\[
\begin{align*}
0 & : i = 0; \\
1 & : T = b + 3; \\
7 & : \text{Neg}(i < n) \\
2 & : \text{Pos}(i < n) \\
3 & : y = T; \\
4 & : A_1 = A + i; \\
5 & : M[A_1] = y; \\
6 & : i = i + 1;
\end{align*}
\]

\[\Rightarrow \] There is no decent place for \( T = b + 3; \) :-(
Idea: Transform into a do-while-loop ...

\[ i = 0; \]

\[ i = i + 1; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ y = b + 3; \]
... now there is a place for $T = e; \quad :-)$

$T = b + 3$

$A_1 = A + i$

$M[A_1] = y$

$i = i + 1$

$y = T$

Neg($i < n$)

Pos($i < n$)

Neg($i < n$)

Pos($i < n$)
Application of T5 (PRE):

\[ i = 0; \]

\[ y = b + 3; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ i = i + 1; \]

<table>
<thead>
<tr>
<th></th>
<th>( \mathcal{A} )</th>
<th>( \mathcal{B} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( \emptyset )</td>
<td>( { b+3 } )</td>
</tr>
<tr>
<td>3</td>
<td>( { b+3 } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>( { b+3 } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>5</td>
<td>( { b+3 } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>6</td>
<td>( { b+3 } )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>7</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
Application of T5 (PRE):

\[ \begin{align*}
i &= 0; \\
A_1 &= A + i; \\
M[A_1] &= y; \\
i &= i + 1;
\end{align*} \]

\begin{align*}
\text{Pos}(i < n) &\quad \text{Neg}(i < n)
\end{align*}

\begin{align*}
0 &\quad 1 \\
1 &\quad Pos(i < n) \\
2 &\quad y = b + 3; \\
3 &\quad A_1 = A + i; \\
4 &\quad M[A_1] = y; \\
5 &\quad i = i + 1;
\end{align*}

\begin{align*}
\text{Pos}(i < n) &\quad \text{Neg}(i < n)
\end{align*}

\begin{tabular}{|c|c|}
\hline
 & \( A \) & \( B \) \\
\hline
0 & \( \emptyset \) & \( \emptyset \) \\
1 & \( \emptyset \) & \( \emptyset \) \\
2 & \( \emptyset \) & \( \{ b + 3 \} \) \\
3 & \( \{ b + 3 \} \) & \( \emptyset \) \\
4 & \( \{ b + 3 \} \) & \( \emptyset \) \\
5 & \( \{ b + 3 \} \) & \( \emptyset \) \\
6 & \( \{ b + 3 \} \) & \( \emptyset \) \\
7 & \( \emptyset \) & \( \emptyset \) \\
\hline
\end{tabular}
Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of the loop :

- This only works properly for **do-while**-loops :

- To optimize other loops, we transform them into **do-while**-loops before-hand:

\[
\text{while } (b) \text{ stmt } \implies \text{if } (b) \\
\quad \text{do stmt} \\
\quad \text{while } (b); \\
\implies \text{ Loop Rotation}
\]
Problem:

If we do not have the source program at hand, we must re-construct potential loop headers ;-

\[ u \Rightarrow v \text{ if every path } \pi: \text{start} \rightarrow^* v \text{ contains } u. \]

We write: \( u \Rightarrow v \).

“\( \Rightarrow \)” is reflexive, transitive and anti-symmetric :-)
Computation:

We collect the nodes along paths by means of the analysis:

\[ \mathcal{P} = 2^{\text{Nodes}}, \quad \sqsubseteq = \supseteq \]

\[ [(\_, \_, v)]^\# P = P \cup \{v\} \]

Then the set \( \mathcal{P}[v] \) of pre-dominators is given by:

\[ \mathcal{P}[v] = \bigcap \{[[\pi]]^\# \{\text{start}\} \mid \pi : \text{start} \rightarrow^* v\} \]
Since $[k]^2$ are distributive, the $\mathcal{P}[v]$ can be computed by means of fixpoint iteration :-)

Example:

\[
\begin{array}{c|c}
\mathcal{P} & \\
0 & \{0\} \\
1 & \{0, 1\} \\
2 & \{0, 1, 2\} \\
3 & \{0, 1, 2, 3\} \\
4 & \{0, 1, 2, 3, 4\} \\
5 & \{0, 1, 5\} \\
\end{array}
\]
The partial ordering “⇒” in the example:

<table>
<thead>
<tr>
<th></th>
<th>( \mathcal{P} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>2</td>
<td>{0, 1, 2}</td>
</tr>
<tr>
<td>3</td>
<td>{0, 1, 2, 3}</td>
</tr>
<tr>
<td>4</td>
<td>{0, 1, 2, 3, 4}</td>
</tr>
<tr>
<td>5</td>
<td>{0, 1, 5}</td>
</tr>
</tbody>
</table>
Apparently, the result is a tree :-) 

In fact, we have:

**Theorem:**

Every node \( v \) has at most one immediate pre-dominator.

**Proof:**

Assume: 

there are \( u_1 \neq u_2 \) which immediately pre-dominate \( v \).

If \( u_1 \Rightarrow u_2 \) then \( u_1 \) not immediate.

Consequently, \( u_1, u_2 \) are incomparable :-)

450
Now for every $\pi : start \rightarrow^* v$:

$$\pi = \pi_1 \pi_2 \quad \text{with} \quad \pi_1 : start \rightarrow^* u_1 \quad \pi_2 : u_1 \rightarrow^* v$$

If, however, $u_1, u_2$ are incomparable, then there is path: $\quad start \rightarrow^* v$

avoiding $u_2$:
Now for every $\pi : \text{start} \rightarrow^* v$:

$$
\pi = \pi_1 \pi_2 \quad \text{with} \quad \pi_1 : \text{start} \rightarrow^* u_1
\quad \pi_2 : u_1 \rightarrow^* v
$$

If, however, $u_1, u_2$ are incomparable, then there is path: $\text{start} \rightarrow^* v$ avoiding $u_2$:
Observation:

The loop head of a while-loop pre-dominates every node in the body.

A back edge from the exit $u$ to the loop head $v$ can be identified through

$$v \in \mathcal{P}[u]$$

Accordingly, we define:
Transformation 6:

\[ u_1 \not\in \mathcal{P}[u] \]
\[ u_2, v \in \mathcal{P}[u] \]

We duplicate the entry check to all back edges :-(
... in the Example:

\[ i = 0; \]

\[ \text{Neg}(i < n) \]

\[ \text{Pos}(i < n) \]

\[ y = b + 3; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ i = i + 1; \]
... in the Example:

\[ i = 0; \]

\[ \text{Neg}(i < n) \]

\[ \text{Pos}(i < n) \]

\[ y = b + 3; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ i = i + 1; \]
... in the Example:

```
i = 0;

Neg(i < n)  Pos(i < n)

A_1 = A + i;

M[A_1] = y;

i = i + 1;
```
... in the Example:

\[
\begin{align*}
\text{i} &= 0; \\
y &= b + 3; \\
A_1 &= A + i; \\
M[A_1] &= y; \\
i &= i + 1;
\end{align*}
\]
Warning:

There are unusual loops which cannot be rotated:

Pre-dominators:
... but also common ones which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated  :-(

... but also common ones which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated  :-(
... but also common ones which cannot be rotated:

Here, the complete block between back edge and conditional jump should be duplicated :-(

462
1.9 Eliminating Partially Dead Code

Example:

\[ x + 1 \quad \text{need only be computed along one path} \quad ;-( \]
Idea:

\[ T = x + 1; \]

\[ M[x] = T; \]

\[ T = x + 1; \]

\[ M[x] = T; \]
Problem:

- The definition $x = e; \ (x \not\in \text{Vars}_e)$ may only be moved to an edge where $e$ is safe ;-

- The definition must still be available for uses of $x$ ;-

We define an analysis which maximally delays computations:

$$
\begin{align*}
[;]^\# D & = D \\
[x = e;]^\# D & = \begin{cases} 
D \setminus (\text{Use}_e \cup \text{Def}_x) \cup \{x = e;\} & \text{if } x \not\in \text{Vars}_e \\
D \setminus (\text{Use}_e \cup \text{Def}_x) & \text{if } x \in \text{Vars}_e
\end{cases}
\end{align*}
$$
... where:

\[ \text{Use}_e = \{ y = e'; \mid y \in \text{Vars}_e \} \]
\[ \text{Def}_x = \{ y = e'; \mid y \equiv x \lor x \in \text{Vars}_{e'} \} \]
... where:

\[
Use_e = \{ y = e'; \mid y \in \text{Vars}_e \}
\]

\[
Def_x = \{ y = e'; \mid y \equiv x \lor x \in \text{Vars}_{e'} \}
\]

For the remaining edges, we define:

\[
[x = M[e];]^{\#} D = D \setminus (Use_e \cup Def_x)
\]

\[
[M[e_1] = e_2;]^{\#} D = D \setminus (Use_{e_1} \cup Use_{e_2})
\]

\[
[\text{Pos}(e)]^{\#} D = [\text{Neg}(e)]^{\#} D = D \setminus Use_e
\]
Warning:

We may move \( y = e; \) beyond a join only if \( y = e; \) can be delayed along all joining edges:

\[
T = x + 1; \\
x = M[T];
\]

Here, \( T = x + 1; \) cannot be moved beyond \( 1 \) !!!
We conclude:

- The partial ordering of the lattice for delayability is given by “≥”.
- At program start: $D_0 = \emptyset$.

Therefore, the sets $D[u]$ of at $u$ delayable assignments can be computed by solving a system of constraints.

- We delay only assignments $a$ where $a a$ has the same effect as $a$ alone.

- The extra insertions render the original assignments as assignments to dead variables ...
Transformation 7:

\[ u \rightarrow \begin{array}{c} \text{lab} \\ \downarrow \end{array} v \]

\[ a \in \mathcal{D}[u] \setminus \left[ \text{lab} \right] \#(\mathcal{D}[u]) \]

\[ u \rightarrow \begin{array}{c} \text{lab} \\ \downarrow \end{array} \]

\[ a \in \left[ \text{lab} \right] \#(\mathcal{D}[u]) \setminus \mathcal{D}[v] \]
Note:

Transformation T7 is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation T2.

In the example, the partially dead code is eliminated:
\[ T = x + 1; \]

\[ M[x] = T; \]

<table>
<thead>
<tr>
<th></th>
<th>( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( { T = x + 1; } )</td>
</tr>
<tr>
<td>2</td>
<td>( { T = x + 1; } )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
M[x] &= T; \\
T &= x + 1; \quad 0 \\
T &= x + 1; \quad 1 \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>(D)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>1</td>
<td>({T = x + 1;})</td>
</tr>
<tr>
<td>2</td>
<td>({T = x + 1;})</td>
</tr>
<tr>
<td>3</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>4</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
\[ M[x] = T; \]

\[ T = x + 1; \]

\[
\begin{array}{c|c}
\text{L} & \\
\hline
0 & \{x\} \\
1 & \{x\} \\
2 & \{x\} \\
2' & \{x, T\} \\
3 & \emptyset \\
4 & \emptyset \\
\end{array}
\]
Remarks:

- After $T7$, all original assignments $y = e; \text{ with } y \notin Vars_e$ are assignments to dead variables and thus can always be eliminated :-)
- By this, it can be proven that the transformation is guaranteed to be non-degradating efficiency of the code :-))
- Similar to the elimination of partial redundancies, the transformation can be repeated :-}
Conclusion:

→ The design of a meaningful optimization is non-trivial.
→ Many transformations are advantageous only in connection with other optimizations :-) 
→ The ordering of applied optimizations matters !!
→ Some optimizations can be iterated !!!
... a meaningful ordering:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T4</td>
<td>Constant Propagation</td>
</tr>
<tr>
<td></td>
<td>Interval Analysis</td>
</tr>
<tr>
<td></td>
<td>Alias Analysis</td>
</tr>
<tr>
<td>T6</td>
<td>Loop Rotation</td>
</tr>
<tr>
<td>T1, T3, T2</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>T2</td>
<td>Dead Variables</td>
</tr>
<tr>
<td>T7, T2</td>
<td>Partially Dead Code</td>
</tr>
<tr>
<td>T5, T3, T2</td>
<td>Partially Redundant Code</td>
</tr>
</tbody>
</table>
2 Replacing Expensive Operations by Cheaper Ones

2.1 Reduction of Strength

(1) Evaluation of Polynomials

\[ f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0 \]

<table>
<thead>
<tr>
<th></th>
<th>Multiplications</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>( \frac{1}{2}n(n+1) )</td>
<td>( n )</td>
</tr>
<tr>
<td>re-use</td>
<td>( 2n - 1 )</td>
<td>( n )</td>
</tr>
<tr>
<td>Horner-Scheme</td>
<td>( n )</td>
<td>( n )</td>
</tr>
</tbody>
</table>
Idea:

\[ f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2})\ldots) \cdot x + a_0 \]

(2) Tabulation of a polynomial \( f(x) \) of degree \( n \):

\[ \rightarrow \] To recompute \( f(x) \) for every argument \( x \) is too expensive \(-\)

\[ \rightarrow \] Luckily, the \( n \)-th differences are constant !!!
**Example:** \[ f(x) = 3x^3 - 5x^2 + 4x + 13 \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( f(n) )</th>
<th>( \Delta )</th>
<th>( \Delta^2 )</th>
<th>( \Delta^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>2</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>10</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the \( n \)-th difference is always

\[
\Delta^n_h(f) = n! \cdot a_n \cdot h^n \quad (h \text{ step width})
\]
Costs:

- $n$ times evaluation of $f$;
- $\frac{1}{2} \cdot (n - 1) \cdot n$ subtractions to determine the $\Delta^k$;
- $n$ additions for every further value

\[ \Rightarrow \]

Number of multiplications only depends on $n$ :-))
Simple Case: \[ f(x) = a_1 \cdot x + a_0 \]

- ... naturally occurs in many numerical loops \(\because\)
- The **first** differences are already constant:
  \[ f(x + h) - f(x) = a_1 \cdot h \]
- Instead of the sequence: \( y_i = f(x_0 + i \cdot h), \ i \geq 0 \)
  we compute:
  \[ y_0 = f(x_0), \ \Delta = a_1 \cdot h \]
  \[ y_i = y_{i-1} + \Delta, \ i > 0 \]
Example:

\[
\text{for } (i = i_0; i < n; i = i + h) \{ \\
\quad A = A_0 + b \cdot i; \\
\quad M[A] = \ldots; \\
\}
\]
... or, after loop rotation:

\[ i = i_0; \]
\[
\text{if } (i < n) \text{ do } \{
    A = A_0 + b \cdot i; \\
    M[A] = \ldots; \\
    i = i + h;
\} \text{ while } (i < n); \]
... and reduction of strength:

\[ i = i_0; \]

\[ \text{if } (i < n) \{ \]
\[ \Delta = b \cdot h; \]
\[ A = A_0 + b \cdot i_0; \]
\[ \text{do } \{ \]
\[ M[A] = \ldots; \]
\[ i = i + h; \]
\[ A = A + \Delta; \]
\[ \} \text{ while } (i < n); \]
\[ \} \]
Warning:

- The values \( b, h, A_0 \) must not change their values during the loop.
- \( i, A \) may be modified at exactly one position in the loop :-(
- One may try to eliminate the variable \( i \) altogether:
  
  \[ \rightarrow \quad i \text{ may not be used elsewhere.} \]
  \[ \rightarrow \quad \text{The initialization must be transformed into:} \]
  \[ A = A_0 + b \cdot i_0 . \]
  \[ \rightarrow \quad \text{The loop condition } i < n \text{ must be transformed into:} \]
  \[ A < N \quad \text{for} \quad N = A_0 + b \cdot n . \]
  \[ \rightarrow \quad b \text{ must always be different from zero} !!! \]