1.14 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:

0
\[ i = 0; \]

1
\[ \text{Neg}(i < n) \]
\[ y = b + 3; \]
\[ \text{Pos}(i < n) \]

2

3
\[ A_1 = A + i; \]

4
\[ M[A_1] = y; \]

5
\[ i = i + 1; \]

6
Warning: \( T = b + 3; \) may not be placed before the loop:

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

\[\Rightarrow\] There is no safe place to move \( T = b + 3; \) to!
Idea: Transform into a do-while-loop ...
... now there is a place for  \( T = e; \)

\[ T = b + 3; \]

\[ i = 0; \]

\[ y = T; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ i = i + 1; \]

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

\[ \text{Pos}(i < n) \]
Application of PRE:

\[ i = 0; \]
\[ y = b + 3; \]
\[ A_1 = A + i; \]
\[ M[A_1] = y; \]
\[ i = i + 1; \]

\[
\begin{array}{c|cc}
\text{Pos}(i < n) & \mathcal{A} & \mathcal{B} \\
\hline
0 & \emptyset & \emptyset \\
1 & \emptyset & \emptyset \\
2 & \emptyset & \{b + 3\} \\
3 & \{b + 3\} & \emptyset \\
4 & \{b + 3\} & \emptyset \\
5 & \{b + 3\} & \emptyset \\
6 & \{b + 3\} & \emptyset \\
7 & \emptyset & \emptyset \\
\end{array}
\]
Application of PRE:

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

\[\begin{array}{|c|c|c|}
\hline
\text{Pos}(i < n) & \mathcal{A} & \mathcal{B} \\
\hline
0 & \emptyset & \emptyset \\
1 & \emptyset & \emptyset \\
2 & \emptyset & \{b + 3\} \\
3 & \{b + 3\} & \emptyset \\
4 & \{b + 3\} & \emptyset \\
5 & \{b + 3\} & \emptyset \\
6 & \{b + 3\} & \emptyset \\
7 & \emptyset & \emptyset \\
\hline
\end{array}\]
Conclusion:

- Elimination of partial redundancies may move loop-invariant code out of a loop
- This only works properly for do-while-loops
- To optimize other loops, we transform them into do-while-loops before-hand:

```plaintext
while (b) stmt  =>  if (b)
               do stmt
               while (b);
```

=> Loop Rotation
Problem:

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

\[ u \preceq v, \text{ if every path } \pi : \text{start} \to^* v \text{ contains } u. \text{ We write: } u \Rightarrow v. \]

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

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

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

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

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

\[ \mathcal{P}[v] = \bigcap \{[[\pi]]^\# \{\text{start}\} \mid \pi : \text{start} \to^* v\} \]
Since $[k]^d$ are distributive, the $\mathcal{P}[v]$ can be computed by means of fixed-point iteration.

**Example:**

![Diagram with nodes and arrows connecting them to illustrate the fixed-point iteration process.]

<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>
The partial ordering “⇒” in the example:

<table>
<thead>
<tr>
<th></th>
<th>( 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 exist $u_1 \neq u_2$ that immediately pre-dominate $v$.

If $u_1 \Rightarrow u_2$ then $u_1$ not immediate.

Consequently, $u_1, u_2$ are incomparable
Now for every $\pi: start \rightarrow^* v$:

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

$$\pi_2: u_1 \rightarrow^* v$$

If $u_1, u_2$ are incomparable, then there is a path: $start \rightarrow^* v$ bypassing $u_2$:
Now for every $\pi : \text{start} \rightarrow^* v$:

$$\pi = \pi_1 \pi_2$$

with

$$\pi_1 : \text{start} \rightarrow^* u_1$$

$$\pi_2 : u_1 \rightarrow^* v$$

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 \texttt{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 LR:

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

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

\[
i = 0; \\
0, 1, 7 \\
y = b + 3; \\
A_1 = A + i; \\
M[A_1] = y; \\
i = i + 1; \\
0, 1, 2, 3, 4, 5, 6
\]
... in the Example:

\[ i = 0; \]

\[ y = b + 3; \]

\[ A_1 = A + i; \]

\[ M[A_1] = y; \]

\[ i = i + 1; \]

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

\[ \text{Pos}(i < n) \]
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
1.15 Eliminating Partially Dead Code

Example:

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

\( x + 1 \) needs only be computed along one path
Idea:

\[ T = x + 1; \]

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

- The definition $x = e; \ (x \not\in Vars_e)$ 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 that maximally delays computations:

\[
\begin{align*}
[;]\# D &= D \\
[x = e;]\# D &= \begin{cases} 
D \setminus (Use_e \cup Def_x) \cup \{x = e;\} & \text{if } x \not\in Vars_e \\
D \setminus (Use_e \cup Def_x) & \text{if } x \in Vars_e
\end{cases}
\end{align*}
\]

... where:

\[
\begin{align*}
Use_e &= \{y = e'; | y \in Vars_e\} \\
Def_x &= \{y = e'; | y \equiv x \lor x \in Vars_{e'}\}
\end{align*}
\]
For the remaining edges, we define:

\[ [x = M[e];] \# D = D \backslash (Use_e \cup Def_x) \]
\[ [M[e_1] = e_2;] \# D = D \backslash (Use_{e_1} \cup Use_{e_2}) \]
\[ [\text{Pos}(e)] \# D = [\text{Neg}(e)] \# D = D \backslash Use_e \]
Warning:

We may move $y = e$ beyond a join only if $y = e$ can be delayed along all joining edges:

$T = x + 1$;

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 assignments delayable at \( u \) 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 PDE:

\[ v \text{ lab lab} \in D \]

\[ a \in D[u] \backslash \llbracket \text{lab} \rrbracket^\sharp(D[u]) \]

\[ v \]

\[ v_1 \]

\[ v_2 \]

\[ Pos(e) \]

\[ Neg(e) \]

\[ u \]

\[ a \in \llbracket \text{lab} \rrbracket^\sharp(D[u]) \backslash D[v] \]

\[ a \in \llbracket \text{lab} \rrbracket^\sharp(D[u]) \backslash D[v] \]

\[ u \]

\[ u \]

\[ u \]

\[ u \]

\[ a \in D[u] \backslash \llbracket \text{Pos}(e) \rrbracket^\sharp(D[u]) \]

\[ a \in \llbracket \text{Pos}(e) \rrbracket^\sharp(D[u]) \backslash D[v_1] \]

\[ a \in \llbracket \text{Pos}(e) \rrbracket^\sharp(D[u]) \backslash D[v_2] \]

\[ v_1 \]

\[ v_2 \]
Note:

Transformation PDE is only meaningful, if we subsequently eliminate assignments to dead variables by means of transformation DE. In the example, the partially dead code is eliminated:
\[ T = x + 1; \]
\[ M[x] = T; \]
\( M[x] = T; \)

\[
\begin{align*}
T &= x + 1; \\
T &= x + 1; \\
M[x] &= T;
\end{align*}
\]

\[\begin{array}{|c|c|}
\hline
D \setminus \emptyset & \emptyset \\
1 & \{T = x + 1;\} \\
2 & \{T = x + 1;\} \\
3 & \emptyset \\
4 & \emptyset \\
\hline
\end{array}\]
\[ M[x] = T; \]
\[ T = x + 1; \]

<table>
<thead>
<tr>
<th></th>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{x}</td>
</tr>
<tr>
<td>1</td>
<td>{x}</td>
</tr>
<tr>
<td>2</td>
<td>{x}</td>
</tr>
<tr>
<td>2'</td>
<td>{x, T}</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
Remarks:

- After $PDE$, 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 proved that the transformation does not deteriorate the efficiency of the code.
- Similar to the elimination of partial redundancies, repeated application of the transformation may be profitable.
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 should be applied repeatedly.
... a meaningful ordering:

<table>
<thead>
<tr>
<th>CP</th>
<th>Constant Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Interval Analysis</td>
</tr>
<tr>
<td></td>
<td>Alias Analysis</td>
</tr>
<tr>
<td>LR</td>
<td>Loop Rotation</td>
</tr>
<tr>
<td>RE</td>
<td>Available Expressions</td>
</tr>
<tr>
<td>DE</td>
<td>Dead Variables</td>
</tr>
<tr>
<td>PDE</td>
<td>Partially Dead Code</td>
</tr>
<tr>
<td>PRE</td>
<td>Partially Redundant Code</td>
</tr>
</tbody>
</table>