Lecture 8
Code Optimizations:
Partial Dead Code Elimination

Dead Code Elimination

Dead Code is code that is either never executed or, if it is executed, its result is never used by the program. Therefore dead code can be eliminated from the program.
Dead Code: More Examples

Values used only by dead statements.

\[
\begin{align*}
\alpha &= - \\
y &= \alpha + 1 \\
\text{Ready} \\
\text{write } y + 1
\end{align*}
\]

Partially Dead Code

Value computed by partially dead statement is sometimes used and sometimes not used.

\[
\begin{align*}
\alpha &= - \\
\alpha &= \alpha + 1 \\
\alpha &= y + 1
\end{align*}
\]

\(\alpha\) is partially dead here

\(\alpha\) is live here

\(\alpha\) is dead here

Write \(\alpha\)
Code Motion Req'd. for Elimination

![Diagram of code motion for elimination]

Sometimes Code Motion Requires Code Restructuring

![Diagram of code motion requiring code restructuring]
Removal of Code From Loops

Critical Edges
Second Order Effects

\[ a = c + d \] cannot be moved till \( x = a + b \) is moved

Algorithm

Repeat

Perform dead/faint variable analysis to identify sinking candidates
Perform delayability analysis for sinking candidates
Place statements in new positions

Until Program Stabilizes
Definitions

**Dead Code:** S: $x = t$ is **dead**, if $x$ is dead following S, i.e. every path from S to every right hand side occurrence of $x$ following S is preceded by a redefinition of $x$.

**Faint Code:** S: $x = t$ is **faint**, if $x$ is faint following S, i.e. every path from S to every right hand side occurrence of $x$ following S is either preceded by a redefinition of $x$ or is in a statement whose left hand side variable is faint.

Local Predicates

**USED**$_S(x)$ - $x$ is rhs variable of statement $s$

**RELV-USED**$_S(x)$ - $x$ is a rhs variable of relevant statement $s$; relevant statements are those that cannot be eliminated - conditionals like if $x < y$ and output statements like write $x$.

**ASSIGN-USED**$_S(x)$ - $x$ is a rhs variable of assignment statement $s$.

**MOD**$_S(x)$ - $x$ is lhs variable of statement $s$. 
Dead Variable Analysis

Bit vector analysis for all variables.

\[ N\text{-DEAD}_s = \neg USED_s \land (X\text{-DEAD}_s \lor MOD_s) \]

\[ X\text{-DEAD}_s = \bigwedge_{t \in Succ(s)} N\text{-DEAD}_t \]

Faint Variable Analysis

Not bit vector analysis.

\[ \forall x, \quad N\text{-FAINT}_s(x) : \neg RELV\text{-USED}_s(x) \]

\[ \land (X\text{-FAINT}_s(x) \lor MOD_s(x)) \]

\[ \land (X\text{-FAINT}_s(x) \lor \neg ASSIGN\text{-USED}_s(x)) \]

\[ X\text{-FAINT}_s(x) = \bigwedge_{t \in Succ(s)} N\text{-FAINT}_t(x) \]
Analysis of Equation

\[
N\text{-FAINT}_s(x) = \\
\text{REL-USED}_s(x) \lor (x\text{-FAINT}_s(x) \land \text{MOD}_s(x)) \lor (x\text{-FAINT}_s(x) \land \text{ASSIGN-USED}_s(x))
\]

\(N\text{-FAINT}_s(x)\) is false if any of the following conditions is true:

1. the statement \(s\) uses \(x\) and \(s\) cannot ever be marked faint, i.e.
   \[\text{REL-USED}_s(x) \text{ is } \text{true}\]

2. the statement \(s\) does not modify \(x\) and \(x\) is not faint on exit, i.e.
   \[\left(\overline{x\text{-FAINT}_s(x)} \land \overline{\text{MOD}_s(x)}\right)\]

3. \(x\) is used by statement \(s\) and lhs of \(s\) is not faint, i.e.
   \[\left(\overline{x\text{-FAINT}_s(x)} \land \overline{\text{ASSIGN-USED}_s(x)}\right)\]

Elimination

Following dead/faint variable analysis we can eliminate dead/faint code \(\rightarrow\) Eliminate \(n\): \(x = \ldots\) assignment \(x = \ldots\). If \(x\) is dead/faint at exit of \(n\).

The above elimination rule eliminates fully dead/faint code. To eliminate partially dead code, we develop delayability analysis \(\rightarrow\) using this analysis partially dead statement is moved to program points where it is fully dead or fully live.

Iterative application of dead/faint code elimination and delaying partially dead assignments leads to optimization.
Example

Delayability Analysis

**LOCDELEYED\(_n\)(\(\alpha\))**: \(\alpha\) is a sinking candidate that \(\in\) \(n\).

**LOCBLOCKED\(_n\)(\(\alpha\))**: sinking of \(\alpha\) is blocked by code in \(n\).

\[
N\text{-DELAYED}_n(\alpha) = \begin{cases} \text{false} & \text{if } n = S \\ \bigwedge_{m \in \text{pred}(n)} X\text{-DELAYED}_m(\alpha) & \text{otherwise} \end{cases}
\]

\[
X\text{-DELAYED}_n(\alpha) = \text{LOCDELEYED}_n(\alpha) \lor (N\text{-DELAYED}_n(\alpha) \land \text{LOCBLOCKED}_n(\alpha))
\]

\(N\text{-DELAYED}_n/X\text{-DELAYED}_n\) indicate whether \(\alpha\) can be delayed to the entry/exit point of \(n\).
Placing $\alpha$ at its new positions.

$$N-\text{INSERT}_n(\alpha) = N-\text{DELAYED}_{\neg n}(\alpha) \land \text{LOCIBLOCKED}_{\neg n}(\alpha)$$

$$X-\text{INSERT}_n(\alpha) = X-\text{DELAYED}_{\neg n} \land \left( \bigvee_{\neg n \in \text{succ}(n)} N-\text{DELAYED}_{\neg n}(\alpha) \right)$$

If $N-\text{INSERT}_n(\alpha)/X-\text{INSERT}_n(\alpha)$ is true then place $\alpha$ at $n$'s entry/exit point. Essentially we are moving $\alpha$ to the latest points to which it can be delayed.