Lazy Code Motion

Andrew Lenharth
September, 2010
Outline

- Introduction
  - Goals
  - Constraints
  - History

- Algorithm
  - Local Properties
  - Dataflow Properties
  - Code Modification
An expression $E$ at point $P$ is redundant if on every path to $P$, $E$ is evaluated and not killed.

B + C is NOT redundant

$... := B + C$

$... ::= B + C$

$B := X + Y$

$... := B + C$

B + C is redundant

$... := B + C$
An expression $E$ is partially redundant at point $P$ if on some paths to $P$, $E$ is evaluated and not killed.

$B + C$ is partially redundant.
Constraints

- **Computationally optimal**
  - Don’t introduce new computations on paths
- **Lifetime optimal**
  - Minimize extra register pressure

\[
\ldots := B + C
\]

\[
\ldots := B + C
\]

\[
\ldots := B + C
\]

\[
\ldots := B + C
\]

\[
\ldots := B + C
\]
History

- **Lazy Code Motion**, Knoop, et al. (1992)
- **A variation of Knoop, Ruthing, and Steffen’s Lazy Code Motion**, Drechsler and Stadel (1993)
- **Partial Redundancy elimination in SSA Form**, Kennedy, et al. (1999)
- **Lazy Strength Reduction**, Knoop, et al. (1993)
Algorithm Outline

- Pull up computations until they are computationally optimal
- Push them down to minimize lifetimes
  - Lifetime Optimal
  - Stay computationally optimal
Preliminary: Representation

- Graph of basic blocks
- Nodes are BBs
- Edges are control flow
- Each expression will be assigned a position in a bit vector
Local Predicates

- **DEExpr**: E is defined and its arguments are not killed
  - E = E at end of block
- **UEExpr**: E is defined and its arguments are not
  - E = E at start of block
- **ExprKill**: E has arguments defined
  - E at start of block may not equal E at end of block
Availability

$\text{AVAILIN}(n) = \bigcap_{m \in \text{preds}(n)} \text{AVAILOUT}(m)$, \hspace{1cm} n \neq n_0

$\text{AVAILOUT}(m) = \text{DEExpr}(m) \cup (\text{AVAILIN}(m) \cap \text{ExprKill}(m))$

- **AvailOut**: E produces the same value at end of n as in the original placement
- **AvailIn**: E is produced on every path from START to n without being killed
  - May be killed in the path, as long as later produced

**Solving**:
- Initialize AvailIn(n) to all expressions
- Use empty set at start node
Anticipability

\begin{align*}
\text{AntOut}(n) &= \bigcap_{m \in \text{succs}(n)} \text{AntIn}(m), \\
\text{AntIn}(m) &= \text{UEExpr}(m) \cup (\text{AntOut}(m) \cap \text{ExprKill}(m))
\end{align*}

- **AntIn**: E produces the same value at the start of n as in the original placement
- **AntOut**: E is produced on every path from n to END before it is killed
  - May be killed, as long as produced first

**Solving:**
- Initialize AntOut to all expressions
- use empty set at exit node

\[
m = \ldots
\]
\[
\ldots
\]
\[
\text{DEExpr}
\]
\[
\text{AvailIn}
\]
\[
\text{AntIn}
\]
\[
\text{UEExpr}
\]
\[
\text{AvailOut}
\]
\[
\text{AntOut}
\]
\[
\text{ExprKill}
\]
**Move Up**

\[
\text{EARLIEST}(i,j) = \text{ANTIN}(j) \cap \text{AVAILOUT}(i) \cap (\text{EXPRKILL}(i) \cup \text{ANTOUT}(i))
\]

\[
\text{EARLIEST}(n_0,j) = \text{ANTIN}(j) \cap \text{AVAILOUT}(n_0)
\]

- **Earliest**(i,j): How far up E can be placed and cover only uses
  - AntIn(j): can place in j
  - !AvailOut(i): not already produced in i
  - ExprKill(i): can’t move past a kill
  - !AntOut(i): or not all outedges need it
Move Down

\[
L_{\text{LaterIn}}(j) = \bigcap_{i \in \text{pred}(j)} L_{\text{Later}}(i,j), \quad j \neq n_0
\]

\[
L_{\text{Later}}(i,j) = L_{\text{Earliest}}(i,j) \cup \left( L_{\text{LaterIn}}(i) \cap U_{\text{EEExpr}}(i) \right)
\]

- LaterIn: Paths with Earliest placements leading to them which do not evaluate E
- Later(i,j): push forward Earliest until a kill or a production
- Solving:
  - Initialize LaterIn to empty set
**Placement**

\[ \text{INSERT}(i,j) = \text{LATER}(i,j) \cap \text{LATERIN}(j) \]

\[ \text{DELETE}(k) = \text{UEEXPR}(k) \cap \text{LATERIN}(k), \, k \neq n_0 \]

- **Insert** \((i,j)\): Latest edge \(E\) can be placed on
- **Delete**: delete first instance of \(E\) in block
m = ... \\
T = l - m := T \\
m = ... \\
T = l - m := 1 - m
Critical Edges

For Insert(i, j)

- Succs(i) = 1: end of I
- Succs(i) > 1, preds(j) = 1: start of j
- Succs(i) > 1, preds(j) > 1: new block
Breaking Critical Edges

Critical edges prevent safe code motion

A critical edge exists between N and M when:
- M has more than one predecessor
- N has more than one successor

Solution: insert a node

Hoisted code would be visible in unintended paths
Perfect?

Any problems?
Profile Insensitive

- Single rarely executed path can limit code motion
  - Also prevents some loop invariant code movement
- Limited sense of Optimality
  - Code size could be reduced
  - Limits scheduling
Not Register Optimal

- If operands are still live, may cause spill with increased register pressure
  - Operation may be cheaper than spilling
  - Rematerialization may handle this

- Lazy placement may keep operands live
  - Earlier placement may reduce register pressure