Code Placement, Code Motion

Compiler Construction Course

Winter Term 2009/2010

SAARLAND UNIVERSITY

COMPUTER SCIENCE

Why?

- Loop-invariant code motion
- Global value numbering destroys block membership
- Remove redundant computations

GVN Recap

- SSA GVN treats the program as a graph
- Nodes are computations $\equiv$ SSA values
- Edges are data dependences
- Graph can be seen as finite state automaton
- Minimized automaton merges multiple congruent SSA values

GVN Recap

- GVN destroys block membership
- Some nodes are pinned
  - Cannot be moved outside the block
  - They cannot be congruent to a node in a different block
  - (non-functional) Calls, Stores, ϕs
- All other nodes do not have side effects and are floating
- Need to place floating computations of minimized program
- Issues:
  - Correctness
  - Efficiency of placed code

A Simple Heuristic

Idea

1. Place nodes as early as possible
   - Earliest point: All operands have to dominate the node
   - Place all operands before
   - Placing a node as early as possible leaves most freedom for its users
   - Gives a correct placement

2. Modify placement and place nodes as late as possible
   - Reduces partial deadness of the computation (Efficiency)
   - Latest point:
     - A node has to dominate all its users
     - Lowest common dominator of all users
   - Might end up in a loop
   - Hence: search for latest node between earliest and latest with lowest loop nesting

Early Placement

- Perform DFS on the reversed SSA graph
- We assume, there is a unique data dependence source (in Firm, there is the `End` node)
- Place node *n* when returning from operands
- Each operand is either a pinned node or has then been placed

- All operands have to dominate the node to be placed
  - All operands lie on a branch in the dominance tree
  - Hence, there is a lowest one
  - This is the earliest block to place the node in
- Example on black board

Late Placement

- Inverse order as early placement
- Forward DFS on the SSA graph
- Place all users of a node first
- Then place the node
- Latest possible placement of the node is the lowest common dominator of all users
- Earliest dominates latest
- Node can be placed everywhere on the dominance branch between earliest and latest
- Search for the latest (lowest) block on that branch with the lowest loop nesting level
- Hoists loop-invariant computations out of loops
- Example on black board

Drawback

Definition

An variable v is dead along a path $P : \mathit{def}(v) \rightarrow^{+} \mathit{end}$, if P does not contain a use of v.
An variable v is fully (partially) dead if it is dead along every (some) path.

- The latest placement might still lead to a partial dead code
- Would need to duplicate computations
- Example on black board
- See `ir/opt/code_placement.c` in libFirm

Partial Redundancy Elimination

- GVN merges congruent computations
- Regardless of redundancy
- Sometimes it eliminates (partially) redundant computations
- Might create partial dead code

- PRE considers placement of computations
- to remove partially redundant computations
- Does not create partial dead code
- But has no concept of congruence
- Few SSA-based algorithms exist
- Here: First part of “Lazy Code Motion”

Redundancy of Computations

Definition

Consider a program point ℓ with a statement

$$\ell : z \leftarrow \tau(x_1, \ldots, x_n)$$

The computation $\tau(x_1, \ldots, x_n)$ is redundant along a path P to ℓ iff there exists $\ell' \in P$ in front of ℓ with

$$\ell' : z \leftarrow \tau(x_1, \ldots, x_n)$$

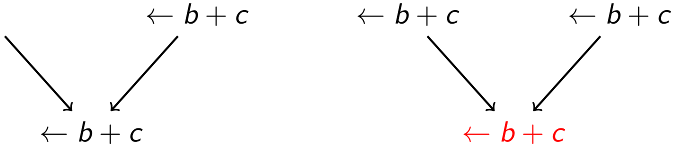
and no (re-)definition to the x_i.

Definition (full and partial redundancy)

A computation $\tau(x_1, \ldots, x_n)$ is fully (partially) redundant if every (some) path to ℓ contains $\tau(x_1, \ldots, x_n)$

Partial Redundant Computations

Example



- Left figure: $a + b$ is partially redundant on right path
- Right figure: Insertion of computation on left branch makes computation below fully redundant

Partial Redundant Computations

Loop-Invariant Code

$$a \leftarrow b + c$$

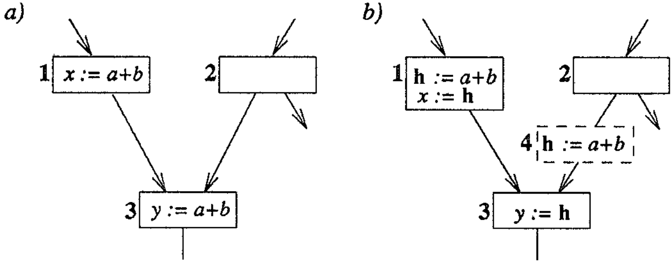
- Loop-invariant code is partial redundant

Code Placement

- Consider an expression $\tau(a, b)$
- A statement $z \leftarrow \tau(a, b)$ is a computation of $\tau(a, b)$
- Code Placement for an expression $\tau(a, b)$ comprises:
  - Insert statements of the form $t \leftarrow \tau(a, b)$ with a new temporary h
  - Rewrite some of the original computations of $\tau(a, b)$ to h

Critical Edges

- Redundancies cannot be removed safely in arbitrary graphs
- Moving $a + b$ from 3 to 2 might create new redundancies there
- This is because the edge $2 \to 3$ is critical



- We need to be able to put code on every edge
- Split every edge from blocks with multiple successors to blocks with multiple predecessors

Anticipability

Aka Down-Safety

- We want to find program points that make computations of t fully redundant
- A program point n is an anticipator of t if a computation of t lies on every path from n to *end*.

Anticipability

Aka Down-Safety

- We want to find program points that make computations of t fully redundant
- A program point n is an anticipator of t if a computation of t lies on every path from n to end.
- This is expressed by following data-flow equation of a backward flow problem

$$A_\bullet(\ell) = \bigcap_{s \in succ(\ell)} A_\circ(s)$$

$$A_\circ(\ell) = \mathsf{UEExpr}(\ell) \cup \left(A_\bullet(\ell) \cap \overline{\mathsf{ExprKill}(\ell)}\right)$$

- $\mathsf{UEExpr}(\ell)$ are the upward exposed expressions of ℓ:
  All variables used before defined in ℓ
- $\mathsf{ExprKill}(\ell)$ is the set of all variables killed in ℓ:
  All variables defined in ℓ

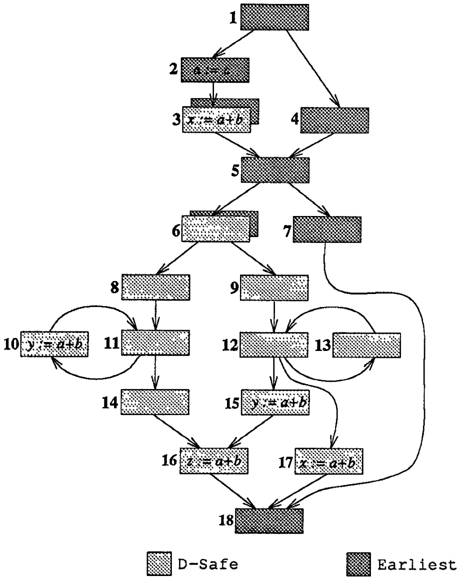
Earliestness

- A placement of t at a node n is earliest if there exists a path from r to n such that no node on P prior to n
  - anticipates t at n
  - or does not produce the same value when evaluating t at m
- Can also be cast as a flow problem:

$$E_{\circ}(\ell) = \bigcup_{p \in pred(\ell)} E_{\bullet}(p)$$

$$E_{\bullet}(\ell) = \mathit{ExprKill}(\ell) \cup \left(\overline{A_{\circ}(\ell)} \cap E_{\circ}(\ell)\right)$$

Example

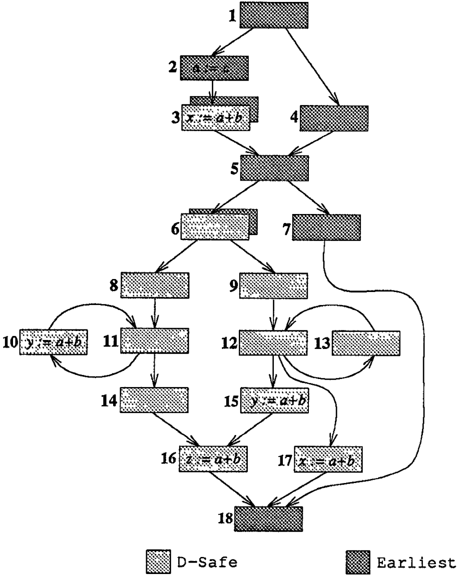

1
2
3
x := a+b
4
5
6
7
8
9
10
y := a+b
11
12
13
14
15
y := a+b
16
z := a+b
17
x := a+b
18
D-Safe
Earliest

The Transformation

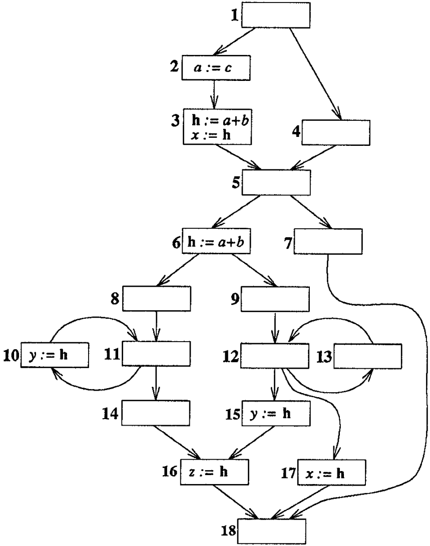
- For every expression $t \equiv \tau(a, b)$, compute E and A.
- Insert $h \leftarrow t$ at the beginning of every n with $t \in A_\circ(n)$ and $t \in E_\circ(n)$
- Replace every original computation of t by h

- This placement is computationally optimal!
- Every other down-safe placement has at least as many computations of t on every possible control flow path from r to e
- Proof sketch: Look at paths from computation points to uses and show that they do not contain redundant computations

Example



1
2 $a := c$
3 $x := a+b$
4
5
6
7
8
9
10 $y := a+b$
11
12
13
14
15 $y := a+b$
16 $z := a+b$
17 $x := a+b$
18
D-Safe
Earliest



1
2 $a := c$
3 $\mathbf{h} := a+b$
$x := \mathbf{h}$
4
5
6 $\mathbf{h} := a+b$
7
8
9
10 $y := \mathbf{h}$
11
12
13
14
15 $y := \mathbf{h}$
16 $z := \mathbf{h}$
17 $x := \mathbf{h}$
18

Literature

Jens Knoop, Oliver Rüthing, and Bernhard Steffen.
Lazy code motion.
In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 224–234, New York, NY, USA, 1992. ACM.

E. Morel and C. Renvoise.
Global optimization by suppression of partial redundancies.
Commun. ACM, 22(2):96–103, 1979.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck.
Global value numbers and redundant computations.
In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.