Announcements

• No lecture next week
  – Traveling to SysML conference (Stanford, CA)

• Assignment 2 is out (due March 8)

• Midterm is March 1\textsuperscript{st} (during the class)
Refreshing: Finding Loops
What is a Loop?

• **Goals:**
  – Define a loop in graph-theoretic terms (control flow graph)
  – Not sensitive to input syntax
  – A uniform treatment for all loops: DO, while, goto’s

• **Not every cycle is a “loop” from an optimization perspective**

• **Intuitive properties of a loop**
  – single entry point
  – edges must form at least a cycle
Formal Definitions

• Dominators
  – Node \(d\) dominates node \(n\) in a graph \((d \ dom \ n)\) if every path from the start node to \(n\) goes through \(d\)

  ![Diagram of dominators]

  – Dominators can be organized as a tree
    • \(a \rightarrow b\) in the dominator tree iff \(a\) immediately dominates \(b\)
Natural Loops

• Definitions
  – Single entry-point: header
    • a header dominates all nodes in the loop
  – A back edge is an arc whose head dominates its tail (tail -> head)
    • a back edge must be a part of at least one loop
  – The natural loop of a back edge is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
Algorithm to Find Natural Loops

• Find the dominator relations in a flow graph

• Identify the back edges

• Find the natural loop associated with the back edge
Finding Back Edges

- **Depth-first spanning tree**
  - Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

- **Categorizing edges in graph**
  - Advancing (A) edges: from ancestor to proper descendant
  - Cross (C) edges: from right to left
  - Retreating (R) edges: from descendant to ancestor (not necessarily proper)
Back Edges

• Definition
  – Back edge: t->h, h dominates t

• Relationships between graph edges and back edges

• Algorithm
  – Perform a depth first search
  – For each retreating edge t->h, check if h is in t’s dominator list

• Most programs (all structured code, and most GOTO programs) have reducible flow graphs
  – retreating edges = back edges

\[ \text{A nonreducible flow graph} \]
Constructing Natural Loops

• The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.

• **Algorithm**
  • delete $h$ from the flow graph
  • find those nodes that can reach $t$
    (those nodes plus $h$ form the natural loop of $t \rightarrow h$)
Inner Loops

• If two loops do not have the same header:
  – they are either disjoint, or
  – one is entirely contained (nested within) the other
  • inner loop: one that contains no other loop.

• If two loops share the same header:
  – Hard to tell which is the inner loop
  – Combine as one
Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop
Finding Loops: Summary

• Define loops in graph theoretic terms

• Definitions and algorithms for:
  – Dominators
  – Back edges
  – Natural loops
Loop-Invariant Computation and Code Motion

• A loop-invariant computation:
  – a computation whose value does not change as long as control stays within the loop

• Code motion:
  – to move a statement within a loop to the preheader of the loop

B, C defined outside of the loop

Function of loop inv

yes

yes

constant

One def inside loop, and one outside

D = A + 1

E = 3

F = A + 2

A = B + C

header

yes
Algorithm

• **Observations**
  – Loop invariant
    • operands are defined outside loop or invariant themselves
  – Code motion
    • not all loop invariant instructions can be moved to preheader

• **Algorithm**
  – Find invariant expressions
  – Conditions for code motion
  – Code transformation
Detecting Loop Invariant Computation

• Compute reaching definitions

• Mark INVARIANT if all the definitions of B and C that reach a statement A=B+C are outside the loop
  – constant B, C?

• Repeat: Mark INVARIANT if
  – all reaching definitions of B are outside the loop, or
  – there is exactly one reaching definition for B, and it is from a loop-invariant statement inside the loop
  – similarly for C

until no changes to set of loop-invariant statements occur.
Example

A = B + C

E = 2
D = A + 1
F = E + 2

E = 3
Example

A = B + C

E = 2

D = A + 1
F = E + 2

E = 3

constant

yes

all reaching defs of B, C are outside loop

yes

constant

yes

one reaching def of A, from loop-inv. statement inside loop

yes

>1 reaching defs of E from inside loop

no
Conditions for Code Motion

- **Correctness**: Movement does not change semantics of program
- **Performance**: Code is not slowed down

- **Basic idea**: defines once and for all
  - control flow: once?
    Code dominates all exists
  - other definitions: for all?
    No other definition
  - other uses: for all?
    Dominates use or no other reaching defs to use
Given: a set of nodes in a loop

- **Compute reaching definitions**
- **Compute loop invariant computation**
- **Compute dominators**
- **Find the exits of the loop (i.e. nodes with successor outside loop)**
- **Candidate statement for code motion:**
  - loop invariant
  - in blocks that dominate all the exits of the loop
  - assign to variable not assigned to elsewhere in the loop
  - in blocks that dominate all blocks in the loop that use the variable assigned
- **Perform a depth-first search of the blocks**
  - Move candidate to preheader if all the invariant operations it depends upon have been moved
Examples

Which statements can be moved to loop preheader?

Only E=3: only statement dominating all exits

A=B+C
D=A+1

(Although E=2, E=3 are invariant, neither is only def of E)

defines once and for all
More Aggressive Optimizations

• Gamble on: most loops get executed
  – Can we relax constraint of dominating all exits?

  \[
  \begin{align*}
  A &= B + C \\
  E &= A + D \\
  D &= \ldots
  \end{align*}
  \]

  Can relax if destination not live after loop
  & can compute in preheader
  w/o causing an exception

• Landing pads
  \[
  \text{While } p \text{ do } s \rightarrow \text{ if } p \{ \\
  \text{preheader} \\
  \text{repeat } s \\
  \text{until not } p; \\
  \}
  \]

  Ensures preheader executes only
  if enter loop
LICM Summary

• Precise definition and algorithm for loop invariant computation

• Precise algorithm for code motion

• Use of reaching definitions and dominators in optimizations
Induction Variables and Strength Reduction

I. Overview of optimization

II. Algorithm to find induction variables
Example

FOR i = 0 to 100
    A[i] = 0;
    i = 0
L2: IF i >= 100 GOTO L1
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i + 1
    GOTO L2
L1:
Definitions

• A **basic induction variable** is
  – a variable X whose only definitions within the loop are assignments of the form:
    
    \[ X = X + c \text{ or } X = X - c, \]

    where \( c \) is either a **constant** or a **loop-invariant variable**.

• An **induction variable** is
  • a **basic induction variable**, or
  • a variable defined once within the loop, whose value is a **linear function of some basic induction variable** at the time of the definition:
    
    \[ A = c_1 \times B + c_2 \]

• The **FAMILY of a basic induction variable** \( B \) is
  • the set of induction variables \( A \) such that each time \( A \) is assigned in the loop, the value of \( A \) is a linear function of \( B \).
Optimizations

1. Strength reduction:

- A is an induction variable in family of basic induction variable B \((A = c_1 * B + c_2)\)

  - Create new variable: \(A'\)
  - Initialization in preheader: \(A' = c_1 * B + c_2;\)
  - Track value of B: \(A' = A' + x * c_1;\)
  - Replace assignment to A: \(A = A';\)
Optimizations (continued)

2. Optimizing non-basic induction variables
   - copy propagation
   - dead code elimination

3. Optimizing basic induction variables
   - Eliminate basic induction variables used only for
     • calculating other induction variables and loop tests
   - **Algorithm**:
     • Select an induction variable \( A \) in the family of \( B \), preferably with simple constants \( A = c_1 \times B + c_2 \).
     • Replace a comparison such as
       \[
       \text{if } B > X \text{ goto L1}
       \]
       with
       \[
       \text{if } (A' > c_1 \times X + c_2) \text{ goto L1} \quad \text{(assuming } c_1 \text{ is positive)}
       \]
     • if \( B \) is live at any exit from the loop, **recompute it from** \( A' \)
       - After the exit, \( B = (A' - c_2) / c_1 \)
II. Basic Induction Variables

• A BASIC induction variable in a loop $L$
  – a variable $X$ whose only definitions within $L$ are assignments of the form:
    $X = X+c$ or $X = X-c$, where $c$ is either a constant or a loop-invariant variable.

• **Algorithm**: can be detected by scanning $L$

• **Example**:

```c
k = 0;
for (i = 0; i < n; i++) {
    k = k + 3;
    ... = m;
    if (x < y)
        k = k + 4;
    if (a < b)
        m = 2 * k;
    k = k - 2;
    ... = m;
}
```

*Each iteration may execute a different number of increments/decrements!!*
Strength Reduction Algorithm

- **Key idea:**
  - For each induction variable $A$, ($A = c_1 \times B + c_2$ at time of definition)
    - variable $A'$ holds expression $c_1 \times B + c_2$ at all times
    - replace definition of $A$ with $A = A'$ only when executed

- **Result:**
  - Program is correct
  - Definition of $A$ does not need to refer to $B$
Finding Induction Variable Families

• Let B be a basic induction variable
  – Find all induction variables A in family of B:
    • $A = c_1 \times B + c_2$
      (where B refers to the value of B at time of definition)

• Conditions:
  – If A has a single assignment in the loop L, and assignment is one of:

    $A = B \times c$
    $A = c \times B$
    $A = B / c$  (assuming A is real)
    $A = B + c$
    $A = c + B$
    $A = B - c$
    $A = c - B$

  – OR, ... (next page)
Finding Induction Variable Families
(continued)

Let D be an induction variable in the family of B \( (D = c_1 \cdot B + c_2) \)

- If A has a single assignment in the loop L, and assignment is one of:
  
  \[
  \begin{align*}
  A &= D \cdot c \\
  A &= c \cdot D \\
  A &= D / c \quad \text{(assuming } A \text{ is real)} \\
  A &= D + c \\
  A &= c + D \\
  A &= D - c \\
  A &= c - D
  \end{align*}
  \]

- No definition of D outside L reaches the assignment to A

- Between the lone point of assignment to D in L and the assignment to A, there are no definitions of B
Summary

• Precise definitions of induction variables
• Systematic identification of induction variables
• Strength reduction

• Clean up:
  – eliminating basic induction variables
    • used in other induction variable calculations
    • replacement of loop tests
  – eliminating other induction variables
    • standard optimizations
Partial Redundancy Elimination

Global code motion optimization
1. Remove partially redundant expressions
2. Loop invariant code motion
3. Can be extended to do Strength Reduction
   • No loop analysis needed
   • Bidirectional flow problem
Redundancy

• A **Common Subexpression** is a **Redundant Computation**

\[
\begin{align*}
t_1 &= a + b \\
t_2 &= a + b \\
t_3 &= a + b
\end{align*}
\]

• Occurrence of expression E at P is **redundant** if E is **available** there:
  – E is evaluated along every path to P, with no operands redefined since.
• Redundant expression can be eliminated
Partial Redundancy

• Partially Redundant Computation

\[ t_1 = a + b \]

\[ t_3 = a + b \]

• Occurrence of expression E at P is **partially redundant** if E is **partially available** there:
  - E is evaluated along **at least one path** to P, with no operands redefined since.

• Partially redundant expression **can be eliminated** if we can **insert computations** to make it **fully redundant**.
Loop Invariants are Partial Redundancies

- Loop invariant expression is partially redundant

\[
t_1 = a + b
\]

As before, partially redundant computation can be eliminated if we insert computations to make it fully redundant.

- Remaining copies can be eliminated through copy propagation or more complex analysis of partially redundant assignments.

\[
t_1 = a + b
\]

\[
t_2 = a + b
\]
Partial Redundancy Elimination (PRE)

• The Method:
  1. Insert Computations to make partially redundant expression(s) fully redundant.
  2. Eliminate redundant expression(s).

• Issues [Outline of Lecture]:
  1. What expression occurrences are candidates for elimination?
  2. Where can we safely insert computations?
  3. Where do we want to insert them?

• For this lecture, we assume one expression of interest, a+b.
  – In practice, with some restrictions, can do many expressions in parallel.
Which Occurrences Might Be Eliminated?

• In CSE,
  – E is **available** at P if it is previously evaluated along **every** path to P, with no subsequent redefinitions of operands.
  – If so, we can eliminate computation at P.

• In PRE,
  – E is **partially available** at P if it is previously evaluated along **at least one** path to P, with no subsequent redefinitions of operands.
  – If so, we might be able to eliminate computation at P, if we can insert computations to make it fully redundant.

• Occurrences of E where E is **partially available** are candidates for elimination.
Finding Partially Available Expressions

- **Forward flow problem**
  - Lattice = \{ 0, 1 \}, meet is union (\( \cup \)), Top = 0 (not PAVAIL), entry = 0

- \( \text{PAVOUT}[i] = (\text{PAVIN}[i] - \text{KILL}[i]) \cup \text{AVLOC}[i] \)

- \( \text{PAVIN}[i] = \begin{cases} 
0 & i = \text{entry} \\
\bigcup_{p \in \text{preds}(i)} \text{PAVOUT}[p] & \text{otherwise} 
\end{cases} \)

- **For a block:** Expression is **locally available** (AVLOC) downwards exposed; Expression is killed (KILL) if any assignments to operands.

\[
\begin{align*}
a & = \ldots \\
\ldots & = a + b
\end{align*}
\]
Partial Availability Example

• For expression a+b.

```
    a = ...
```

KILL = 1    PAVIN =
AVLOC = 0    PAVOUT =

```
    t1 = a + b
```

KILL = 0    PAVIN =
AVLOC = 1    PAVOUT =

```
    a = ...
```

```
    t2 = a + b
```

KILL = 1    PAVIN =
AVLOC = 1    PAVOUT =

• Occurrence in loop is partially redundant.
Where Can We Insert Computations?

- **Safety**: never introduce a new expression along any path.
  - Insertion could introduce exception, change program behavior.
  - If we can add a new basic block, can insert safely in most cases.
  - Solution: insert expression only where it is **anticipated**.

- **Performance**: never increase the # of computations on any path.
  - Under simple model, guarantees program won’t get worse.
  - Reality: might increase register lifetimes, add copies, lose.
Finding Anticipated Expressions

• **Backward flow problem**
  – Lattice = \{ 0, 1 \}, meet is intersection (\cap), top = 1 (ANT), exit = 0

  • \text{ANTIN}[i] = \text{ANTLOC}[i] \cup (\text{ANTOUT}[i] - \text{KILL}[i])

  • \text{ANTOUT}[i] = \begin{cases} 
  0 & i = \text{exit} \\
  \cap \text{ANTIN}[s] & \text{otherwise} 
  \end{cases}

  \text{\textbf{For a block:} Expression locally anticipated (ANTLOC) if upwards exposed.}

\[
\begin{align*}
\text{a} &= \ldots \\
\ldots &= \text{a} + \text{b} \\
\text{a} &= \ldots
\end{align*}
\]
Anticipation Example

- For expression $a+b$.

- Expression is anticipated at end of first block.
- Computation may be safely inserted there.
Where Do We Want to Insert Computations?

• Morel-Renvoise and variants: “Placement Possible”
  – Dataflow analysis shows where to insert:
    • PPIN = “Placement possible at entry of block or before.”
    • PPOUT = “Placement possible at exit of block or before.”
  – Insert at earliest place where PP = 1.
  – Only place at end of blocks,
    • PPIN really means “Placement possible or not necessary in each predecessor block.”
  – Don’t need to insert where expression is already available.
    • $\text{INSERT}[i] = \text{PPOUT}[i] \cap (\neg \text{PPIN}[i] \cup \text{KILL}[i]) \cap \neg \text{AVOUT}[i]$

– Remove (upwards-exposed) computations where PPIN=1.
  • $\text{DELETE}[i] = \text{PPIN}[i] \cap \text{ANTLOC}[i]$
Where Do We Want to Insert?

\[ a = \ldots \]
\[ t_1 = a + b \]
\[ a = \ldots \]
\[ t_2 = a + b \]

PPIN =
PPOUT =

PPIN =
PPOUT =

PPIN =
PPOUT =

PPIN =
PPOUT =
Formulating the Problem

- **PPOUT**: we want to place at output of this block only if
  - we want to place at entry of all successors

- **PPIN**: we want to place at input of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don’t insert at input)
  - we can gain something by placing it here (PAVIN)

- **Forward or Backward?**
  - BOTH!

- **Problem is bidirectional**, but lattice \{0, 1\} is finite, so
  - as long as transfer functions are monotone, it converges.
Computing “Placement Possible”

- **PPOUT**: we want to place at output of this block only if
  - we want to place at entry of all successors

\[
\text{PPOUT}[i] = \begin{cases} 
0 & i = \text{entry} \\
\bigcap_{s \in \text{succ}(i)} \text{PPIN}[s] & \text{otherwise}
\end{cases}
\]

- **PPIN**: we want to place at start of this block only if (all of):
  - we have a local computation to place, or a placement at the end of this block which we can move up
  - we want to move computation to output of all predecessors where expression is not already available (don’t insert at input)
  - we gain something by moving it up (PAVIN heuristic)

\[
\text{PPIN}[i] = \begin{cases} 
0 & i = \text{exit} \\
([\text{ANTLOC}[i] \cup (\text{PPOUT}[i] - \text{KILL}[i]]) \\
\bigcap_{p \in \text{preds}(i)} (\text{PPOUT}[p] \cup \text{AVOUT}[p]) \bigcap \text{PAVIN}[i]) & \text{otherwise}
\end{cases}
\]
“Placement Possible” Example 1

\[
t_1 = a + b
\]
\[
a = \ldots
\]
\[
t_2 = a + b
\]

- KILL = 1
- PAVIN = 0
- PPIN =
- AVLOC = 0
- PAVOUT = 0
- PPOUT =
- ANTLOC = 0
- AVOUT = 0
- PPOUT =

- KILL = 0
- PAVIN = 1
- PPIN =
- AVLOC = 1
- PAVOUT = 1
- PPOUT =
- ANTLOC = 1
- AVOUT = 1
- PPOUT =

- KILL = 1
- PAVIN = 1
- PPIN =
- AVLOC = 1
- PAVOUT = 1
- PPOUT =
- ANTLOC = 0
- AVOUT = 1
- PPOUT =
“Placement Possible” Example 2

\[ a = \ldots \]
\[ t_1 = a + b \]
\[ a = \ldots \]
\[ t_2 = a + b \]

- \( KILL = 1 \)
- \( AVLOC = 1 \)
- \( ANTLOC = 0 \)
- \( PAVIN = 0 \)
- \( PAVOUT = 1 \)
- \( PPOUT = \) 

- \( KILL = 1 \)
- \( AVLOC = 0 \)
- \( ANTLOC = 0 \)
- \( PAVIN = 0 \)
- \( PAVOUT = 0 \)
- \( PPOUT = \) 

- \( KILL = 0 \)
- \( AVLOC = 1 \)
- \( ANTLOC = 1 \)
- \( PAVIN = 1 \)
- \( PAVOUT = 1 \)
- \( PPOUT = \)
“Placement Possible” Correctness

- **Convergence** of analysis: transfer functions are monotone.
- **Safety**: Insert only if anticipated.

\[
PPIN[i] \subseteq (PPOUT[i] - KILL[i]) \cup ANTLOC[i]
\]

\[
PPOUT[i] = \begin{cases} 
0 & i = \text{exit} \\
\bigcap_{s \in \text{succ}(i)} PPIN[s] & \text{otherwise}
\end{cases}
\]

- **INSERT** $\subseteq PPOUT \subseteq ANOUT$, so insertion is safe.
- **Performance**: never increase the # of computations on any path
  - **DELETE** = $PPIN \cap ANLOC$
  - On every path from an INSERT, there is a DELETE.
  - The number of computations on a path does not increase.
CSC D70: Compiler Optimization
LICM: Loop Invariant Code Motion

Prof. Gennady Pekhimenko
University of Toronto
Winter 2018

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons
Backup Slides