Lecture 29: Control Flow Analysis and Loop Optimization
4 Apr 08
Agenda

• Discovering loops in control-flow graphs
  – Dominators
    • Compute dominators by data-flow analysis

• Loop invariant code motion
  – Discovering loop-invariant definitions
    • Application of reaching definitions
  – Validating movement of loop-invariant definition
    • Application of live variable analysis
    • Application of reaching definitions
Program Loops

• **Loop** = a computation repeatedly executed until a terminating condition is reached.

• High-level loop constructs:
  - While loop: \( \text{while}(E) \ S \)
  - Do-while loop: \( \text{do} \ S \text{ while}(E) \)
  - For loop: \( \text{for}(i=1; \ i<=u; \ i+=c) \ S \)

• **Why are loops important:**
  - Most of the execution time is spent in loops
  - Typically: 90/10 rule, 10% code is a loop

• Therefore, loops are important targets of optimizations
Detecting Loops

• Need to identify loops in the program
  - Easy to detect loops in high-level constructs
  - Harder to detect loops in low-level code or in general control-flow graphs

• Examples where loop detection is difficult:
  - Languages with unstructured “goto” constructs: structure of high-level loop constructs may be destroyed
  - Optimizing Java bytecodes (without high-level source program): only low-level code is available
Control-Flow Analysis

• **Goal:** identify loops in the control flow graph

• A loop in the CFG:
  - Is a *set of CFG nodes* (basic blocks)
  - Has a *loop header* such that control to all nodes in the loop always goes through the header
  - Has a *back edge* from one of its nodes to the header
Control-Flow Analysis

- **Goal:** identify loops in the control flow graph

- **A loop in the CFG:**
  - Is a set of CFG nodes (basic blocks)
  - Has a loop header such that control to all nodes in the loop always goes through the header
  - Has a back edge from one of its nodes to the header
Dominators

- Use concept of dominators in CFG to identify loops
- Node d dominates node n if all paths from the entry node to n go through d

Every node dominates itself
1 dominates 1, 2, 3, 4
2 doesn’t dominate 4
3 doesn’t dominate 4

- Intuition:
  - Header of a loop dominates all nodes in loop body
  - Back edges = edges whose heads dominate their tails
  - Loop identification = back edge identification
Immediate Dominators

• Properties:
  1. CFG entry node $n_0$ dominates all CFG nodes
  2. If $d_1$ and $d_2$ dominate $n$, then either
    - $d_1$ dominates $d_2$, or
    - $d_2$ dominates $d_1$

• $d$ strictly dominates $n$ if $d$ dominates $n$ and $d \neq n$

• The immediate dominator $\text{idom}(n)$ of a node $n$ is the unique last strict dominator on any path from $n_0$ to $n$
Dominator Tree

• Build a dominant tree as follows:
  – Root is CFG entry node $n_0$
  – $m$ is child of node $n$ iff $n = idom(m)$

• Example:
Computing Dominators

• Formulate problem as a system of constraints:
  - Define $\text{dom}(n) =$ set of nodes that dominate $n$
  - $\text{dom}(n_0) = \{n_0\}$
  - $\text{dom}(n) = \cap \{ \text{dom}(m) \mid m \in \text{pred}(n) \} \cup \{n\}$
    
    i.e, the dominators of $n$ are the dominators of all of $n$’s predecessors and $n$ itself
Dominators as a Dataflow Problem

- Let $N = \text{set of all basic blocks}$
- Lattice: $(2^N, \subseteq)$; has finite height
- Meet is set intersection, top element is $N$
- Is a forward dataflow analysis
- Dataflow equations:
  - $\text{out}[B] = F_B(\text{in}[B])$, for all $B$
  - $\text{in}[B] = \cap \{\text{out}[B'] | B' \in \text{pred}(B)\}$, for all $B$
  - $\text{in}[B_s] = \emptyset$
- Transfer functions: $F_B(X) = X \cup \{B\}$
  - are monotonic and distributive
- Iterative solving of dataflow equation:
  - terminates
  - computes MOP solution
Natural Loops

- **Back edge**: edge $n \rightarrow h$ such that $h$ dominates $n$
- **Natural loop** of a back edge $n \rightarrow h$:
  - $h$ is loop header
  - Set of loop nodes is set of all nodes that can reach $n$ without going through $h$
- **Algorithm to identify natural loops in CFG**:
  - Compute dominator relation
  - Identify back edges
  - Compute the loop for each back edge

for each node $h$ in dominator tree
  for each node $n$ for which there exists a back edge $n \rightarrow h$
    define the loop with
    header $h$
    back edge $n \rightarrow h$
    body consisting of all nodes reachable from $n$ by a depth first search backwards from $n$ that stops at $h$
Disjoint and Nested Loops

- **Property:** for any two natural loops in the flow graph, one of the following is true:
  1. They are disjoint
  2. They are nested
  3. They have the same header

- **Eliminate alternative 3:** if two loops have the same header and none is nested in the other, combine all nodes into a single loop

Two loops: \{1,2\} and \{1,3\}
Combine into one loop: \{1,2,3\}
Loop Preheader

• Several optimizations add code before header
• Insert a new basic block (called preheader) in the CFG to hold this code
Loop optimizations

• Now we know the loops

• Next: optimize these loops
  – Loop invariant code motion
  – Strength reduction of induction variables
  – Induction variable elimination
Loop Invariant Code Motion

• **Idea:** if a computation produces same result in all loop iterations, move it out of the loop

• **Example:**
  
  ```c
  for (i=0; i<10; i++)
      buf[i] = 10*i + x*x;
  ```

• **Expression** $x^2$ produces the same result in each iteration; move it out of the loop:

  ```c
  t = x*x;
  for (i=0; i<10; i++)
      buf[i] = 10*i + t;
  ```
Loop Invariant Computation

• An instruction $a = b \text{ OP } c$ is loop-invariant if each operand is:
  - Constant, or
  - Has all definitions outside the loop, or
  - Has exactly one definition, and that is a loop-invariant computation

• Reaching definitions analysis computes all the definitions of $x$ and $y$ that may reach $t = x \text{ OP } y$
Algorithm

\[ INV = \emptyset \]

repeat

for each instruction I in loop such that I \( \notin \) INV
  
  if operands are constants, or operands have definitions outside the loop, or operands have exactly one definition d \( \in \) INV

  then \( INV = INV \cup \{I\} \)

until no changes in INV
Code Motion

• Next: move loop-invariant code out of the loop
• Suppose $a = b \text{ OP } c$ is loop-invariant
• We want to hoist it out of the loop
Valid Code Motion

- Code motion of a definition \( d: a = b \text{ OP } c \) to pre-header is valid if:
  1. Definition \( d \) dominates all loop exits where \( a \) is live
     - Use dominator tree to check whether each loop exit is dominated by \( d \)
  2. There is no other definition of \( a \) in loop
     - Scan all body for any other definitions of \( a \)
  3. All uses of \( a \) in loop can only be reached from definition \( d \)
     - Consult reaching definitions at each use of \( a \) for any definitions of \( a \) other than \( d \)
Valid Code Motion

• Invalid example 1: \( a = x^2; \) does not dominate break to use of \( a\)
  
  \[
  a = 0;
  \]
  
  \[
  \text{for } (i=0; i<10; i++)
  \]
  
  \[
  \text{if ( f(i ) a = x^2; break;}
  \]
  
  \[
  b = a;
  \]

• Invalid example 2: there is another definition of \( a \) in loop

  \[
  \text{for } (i=0; i<10; i++)
  \]
  
  \[
  \text{if ( f(i ) a = x^2; else a = 0;}
  \]

• Invalid example 3: use of \( a \) in loop can be reached from \( a=0; \)

  \[
  a = 0;
  \]
  
  \[
  \text{for } (i=0; i<10; i++)
  \]
  
  \[
  \text{if ( f(i ) a = x^2; else buf[i] = a;}
  \]
Other Issues

- **Preserve dependencies** between loop-invariant instructions when hoisting code out of the loop

  ```
  for (i=0; i<N; i++) {
    x = y+z;
    x = y+z;
    a[i] = 10*i + x*x;
  }
  ```

  ```
  for (i=0; i<N; i++) {
    t = x*x;
    a[i] = 10*i + x*x; 
  }
  ```

- **Nested loops**: apply loop-invariant code motion algorithm multiple times

  ```
  for (i=0; i<N; i++) {
    for (j=0; j<M; j++)
      a[i][j] = x*x + 10*i + 100*j;
  }
  ```

  ```
  t1 = x*x;
  for (i=0; i<N; i++) {
    t2 = t1 + 10*i;
    for (j=0; j<M; j++)
      a[i][j] = t2 + 100*j; 
  }
  ```