Control Flow Analysis and Loop Optimization

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 $d_1$
  3. $d$ 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 dominator tree as follows:
  - Root is CFG entry node $n_0$
  - $m$ is child of node $n$ iff $n = \text{idom}(m)$

- Example:
Computing Dominators

- Formulate problem as a system of constraints:
  - Define \( \text{dom}(n) = \text{set of nodes that dominate } n \)
  - \( \text{dom}(n) = \{ n \} \)
  - \( \text{dom}(n) = n \{ \text{dom}(m) \mid m \in \text{pred}(n) \} \cup \{ n \} \)
    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:
  \[
  \text{out}[B] = \text{FB}(\text{in}[B]), \quad \forall B
  \]
  \[
  \text{in}[B] = \bigcap \{ \text{out}[B'] \mid B' \in \text{pred}(B) \}, \quad \forall B
  \]
- Transfer functions: \( \text{FB}(X) = X \cup \{ B \} \)
  - 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

![CFG Diagram](image)

### 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 the same result in all loop iterations, move it out of the loop

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

Expression `x*x` produces the same result in each iteration; move it out of the loop:
```
t = x*x;
for (i=0; i<10; i++)
    buf[i] = 10*i + t;
```

### Loop Invariant Computation

- An instruction `a = b 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 OP y`
Algorithm

INV = ∅
repeat
   for each instruction I in loop such that I ∉ INV
      if operands are constants, or operands have definitions outside the loop, or operands have exactly one definition d ∈ INV then INV = INV U {I}
until no changes in INV

Code Motion

• Next: move loop-invariant code out of the loop
• Suppose a = b 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 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*x; does not dominate break to use of a

```plaintext
a = 0;
for (i=0; i<10; i++)
   if ( f(i) ) a = x*x; else break;
   b = a;
```

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

```plaintext
f(i)
for (i=0; i<10; i++)
   if ( f(i) ) a = x*x;
   else a = 0;
```

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

```plaintext
a = 0;
for (i=0; i<10; i++)
   if ( f(i) ) a = x*x;
   else buf[i] = a;
```
Other Issues

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

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

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

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

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

Induction Variables

- An induction variable is a variable in a loop, whose value is a function of the loop iteration number \( v = f(i) \)

- In compilers, this a linear function:

\[ f(i) = ci + d \]

- Observation: linear combinations of linear functions are linear functions

\( \text{Consequence: linear combinations of induction variables are induction variables} \)

Families of Induction Variables

- Basic induction variable: a variable whose only definition in the loop body is of the form

\( i = i + c \)

where \( c \) is a loop-invariant value

- Derived induction variables: Each basic induction variable \( i \) defines a family of induction variables \( \text{Family}(i) \)

\( i \in \text{Family}(i) \)

- \( k \in \text{Family}(i) \) if there is only one definition of \( k \) in the loop body, and it has the form \( k = ci \) or \( k = sj + c \), where

\( j \in \text{Family}(i) \)

\( c \) is loop invariant

\( (a) \) The only definition of \( j \) that reaches the definition of \( k \) is in the loop

\( (b) \) There is no definition of \( i \) between the definitions of \( j \) and \( k \)
Representation

- **Representation of induction variables in family i by triples:**
  - Denote basic induction variable i by \(<i, 1, 0>\)
  - Denote induction variable \(k = ia + b\) by triple \(<i, a, b>\)

Finding Induction Variables

Scan loop body to find all basic induction variables

\[
\text{do} \quad \text{Scan loop to find all variables } \quad \text{k with one assignment of form } \quad k = j*b, \text{ where } j \text{ is an induction variable } \quad \text{<i,c,d>}, \text{ and make } k \text{ an} \quad \text{induction variable with triple } \quad \text{<i,c*b,d>}
\]

\[
\text{Scan loop to find all variables } \quad \text{k with one assignment of form } \quad k = j*b, \text{ where } j \text{ is an induction variable with triple } \quad \text{<i,c,d>}, \text{ and} \quad \text{make } k \text{ an induction variable with triple } \quad \text{<i,c*b,d>}
\]

until no more induction variables found

Strength Reduction

- **Basic idea:** replace expensive operations (multiplications) with cheaper ones (additions) in definitions of induction variables

\[
\text{while } (i<10) \{ \\
\quad j = \ldots; \quad \text{// <i,3,1>} \\
\quad a[j] = a[j] - 2; \\
\quad i = i+2; \\
\}
\]

\[
\text{ benefited: cheaper to compute } \quad s = s+6 \text{ than } \quad j = 3*i \\
\text{ - } \quad s = s+6 \text{ requires an addition} \\
\text{ - } \quad j = 3*i \text{ requires a multiplication}
\]

General Algorithm

- **Algorithm:**

For each induction variable \(j\) with triple \(<i,a,b>\) whose definition involves multiplication:

1. create a new variable \(s\)
2. replace definition of \(j\) with \(j=s\)
3. immediately after \(i=i+c\), insert \(s = s+a*c\) (here \(a*c\) is constant) 
4. insert \(s = a*i+b\) into preheader

- **Correctness:** transformation maintains invariant \(s = a*i+b\)
Strength Reduction

• Gives opportunities for copy propagation, dead code elimination

\[
\begin{align*}
s &= 3^i + 1; \\
\text{while}(i<10) \{ \\
    &j = s; \\
    &a[j] = a[j] - 2; \\
    &i = i + 2; \\
    &s = s + 6; \\
\} 
\end{align*}
\]

Induction Variable Elimination

• Idea: eliminate each basic induction variable whose only uses are in loop test conditions and in their own definitions \( i = i + c \)
  - rewrite loop test to eliminate induction variable

\[
\begin{align*}
s &= 3^i + 1; \\
\text{while}(i<10) \{ \\
    &a[s] = a[s] - 2; \\
    &i = i + 2; \\
    &s = s + 6; \\
\} 
\end{align*}
\]

• When are induction variables used only in loop tests?
  - Usually, after strength reduction
  - Use algorithm from strength reduction even if definitions of induction variables don’t involve multiplications

Induction Variable Elimination

• Rewrite test condition using derived induction variables
• Remove definition of basic induction variables (if not used after the loop)

\[
\begin{align*}
s &= 3^i + 1; \\
\text{while}(i<10) \{ \\
    &a[s] = a[s] - 2; \\
    &i = i + 2; \\
    &s = s + 6; \\
\} 
\end{align*}
\]

Induction Variable Elimination

For each basic induction variable \( i \) whose only uses are
  - The test condition \( i < u \)
  - The definition of \( i: i = i + c \)

• Take a derived induction variable \( k \) in family \( i \), with triple \( <i,c,d> \)
• Replace test condition \( i < u \) with \( k < c*u + d \)
• Remove definition \( i = i + c \) if \( i \) is not live on loop exit