Optimizing Compilers

Effective optimizing compilers need to gather information about the structure and the flow of control through programs.

- Which instructions are always executed before a given instruction
- Which instructions are always executed after a given instruction
- Where the loops in a program are
  - 90% of any computation is normally spent in 10% of the code: the inner loops
- We’ve already seen how construction of a control-flow graph can help give us some of this information
- In this lecture, we’ll show how to analyze the control-flow graph to detect more refined control-flow information.
Basic Blocks

- *Basic Block* - run of code with single entry and exit.
- Control flow graph of basic blocks more convenient.
- Determine by the following:
  1. Find *leaders*:
     (a) First statement
     (b) Targets of conditional and unconditional branches
     (c) Instructions that follow branches
  2. Basic blocks are leader up to, but not including next leader.
Basic Block Example

r1 = 0

LOOP:
r1 = r1 + 1
r2 = r1 & 1
BRANCH r2 == 0, ODD
r3 = r3 + 1
JUMP NEXT

ODD:
r4 = r4 + 1

NEXT:
BRANCH r1 <= 10, LOOP
Domination Motivation

Constant Propagation:

\[ r_1 = 4 \]
\[ r_2 = r_1 + 5 \]
\[ r_2 = 9 \]

\[ r_1 = 4 \]
\[ r_2 = r_1 + 5 \]
\[ r_2 = 9 \]
Domination

- Assume every Control Flow Graph (CFG) has start node $s_0$ with no predecessors.
- Node $d$ dominates node $n$ if every path of directed edges from $s_0$ to $n$ must go through $d$.
- Every node dominates itself.
- Consider:

  ![Diagram]

  - If $d$ dominates each of the $p_i$, then $d$ dominates $n$.
  - If $d$ dominates $n$, then $d$ dominates each of the $p_i$. 
Dominator Analysis

- If $d$ dominates each of the $p_i$, then $d$ dominates $n$.
- If $d$ dominates $n$, then $d$ dominates each of the $p_i$.
- $Dom[n] = \text{set of nodes that dominate node } n$.
- $N = \text{set of all nodes}$.

- Computation:
  1. $Dom[s_0] = \{s_0\}$.
  2. for $n \in N - \{s_0\}$ do $Dom[n] = N$
  3. while (changes to any $Dom[n]$ occur) do
  4. for $n \in N - \{s_0\}$ do
  5. $Dom[n] = \{n\} \cup (\cap_{p \in pred[n]} Dom[p])$. 
## Dominator Analysis Example

<table>
<thead>
<tr>
<th>Node</th>
<th>$Dom[n]$</th>
<th>$Dom[n]$</th>
<th>$IDom[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing the flow of control through nodes 1 to 12 with connections indicated by lines.
Immediate Dominator

- Immediate dominator used in constructing *dominator tree*.
- Dominator Tree:
  - efficient representation of dominator information
  - used for other types of analysis (e.g. control dependence)
- $s_0$ is root of dominator tree.
- Each node $d$ dominates only its descendants in tree.
- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
  - $IDom[n] \neq n$
  - $IDom[n]$ dominates $n$
  - $IDom[n]$ does not dominate any other dominator of $n$.
- Last dominator of $n$ on any path from $s_0$ to $n$ is $IDom[n]$. 
Immediate Dominator Example

<table>
<thead>
<tr>
<th>Node</th>
<th>$Dom[n]$</th>
<th>$IDom[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,2,3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1,2,4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1,2,5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1,2,4,6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1,2,7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1,2,5,8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1,2,5,8,9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1,2,5,8,9,10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1,2,7,11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1,2,12</td>
<td></td>
</tr>
</tbody>
</table>
Post-Domination

- Assume every Control Flow Graph (CFG) has exit node $x$ with no successors.
- Node $p$ post-dominates node $n$ if every path of directed edges from $n$ to $x$ must go through $p$.
- Every node post-dominates itself.
- Derivation of post-dominator and immediate post-dominator analysis analogous to dominator and immediate dominator analysis.
- Post-dominators will be useful in computing control dependence.
- Control dependence will be useful in many future optimizations.
Loop Optimizations

- First step in loop optimization → find the loops.

- A loop is a set of CFG nodes $S$ such that:
  1. there exists a header node $h$ in $S$ that dominates all nodes in $S$.
     - there exists a path of directed edges from $h$ to any node in $S$.
     - $h$ is the only node in $S$ with predecessors not in $S$.
  2. from any node in $S$, there exists a path of directed edges to $h$.

- A loop is a single entry, multiple exit region.
Examples of Loops
Back Edges

- **Back-edge** - flow graph edge from node $n$ to node $h$ such that $h$ dominates $n$
- Each back-edge has a corresponding *natural loop*.
Natural Loops

- Natural loop of back-edge \( \langle n, h \rangle \):
  - has a loop header \( h \).
  - set of nodes \( X \) such that \( h \) dominates \( x \in X \) and there is a path from \( x \) to \( n \) not containing \( h \).
- A node \( h \) may be header of more than one natural loop.
- Natural loops may be nested.
Loop Optimization

- Compiler should optimize inner loops first.
  - Programs *typically* spend most time in inner loops.
  - Optimizations may be more effective $\rightarrow$ loop invariant code removal.
- Convenient to merge natural loops with same header.
- These merged loops are not natural loops.
- Not all cycles in CFG are loops of any kind (more later).
Loop Optimization

Loop invariant code motion

- An instruction is loop invariant if it computes the same value in each iteration.
- Invariant code may be hoisted outside the loop.

```
ADDI   r1 = r0 + 0
LOAD   r2 = M[FP + a]
ADDI   r3 = r0 + 4
LOAD   r6 = M[FP + x]

LOOP:
  MUL   r4 = r3 * r1
  ADD   r5 = r2 + r4
  STORE M[r5] = r6

ADDI   r1 = r1 + 1
BRANCH r1 <= 10, LOOP
```
Loop Optimization

- **Induction variable analysis and elimination** - $i$ is an induction variable if only definitions of $i$ within loop increment/decrement $i$ by loop-invariant value.

- **Strength reduction** - replace expensive instructions (like multiply) with cheaper ones (like add).

```
ADDI   r1 = r0 + 0
LOAD   r2 = M[FP + a]
ADDI   r3 = r0 + 4
LOAD   r6 = M[FP + x]

LOOP:
MUL    r4 = r3 * r1
ADD    r5 = r2 + r4
STORE  M[r5] = r6

ADDI   r1 = r1 + 1
BRANCH r1 <= 10, LOOP
```
Non-Loop Cycles

Examples:
Non-Loop Cycles

• Loops are instances of *reducible* flow graphs.
  – Each cycle of nodes has a unique header.
  – During reduction, entire loop becomes a single node.

• Non-Loops are instances of *irreducible* flow graphs.
  – Analysis and optimization is more efficient on reducible flow graphs.
  – Irreducible flow graphs occur rarely in practice.
    * Use of structured constructs (e.g. if-then, if-then-else, while, repeat, for) leads to reducible flow graphs.
    * Use of goto’s *may* lead to irreducible flow graphs.
  – Fortunately, Tiger and ML don’t have gotos.
Loop Preheaders

Recall:

- A loop is a set of CFG nodes $S$ such that:
  1. there exists a header node $h$ in $S$ that dominates all nodes in $S$.
     - there exists a path of directed edges from $h$ to any node in $S$.
     - $h$ is the only node in $S$ with predecessors not in $S$.
  2. from any node in $S$, there exists a path of directed edges to $h$.

- A loop is a single entry, multiple exit region.

Loop Preheaders:

- Some loop optimizations (loop invariant code removal) need to insert statements immediately before loop header.

- Create a loop preheader - a basic block before the loop header block.
Loop Preheader Example
Loop Invariant Computations

- Given statements in loop $s$: $\tau = a_1 \circ \mathcal{P} a_2$:
  - $s$ is loop-invariant if $a_1, a_2$ have same value each loop iteration.
  - may sometimes be possible to hoist $s$ outside loop.

- Cannot always tell whether $a$ will have same value each iteration $\rightarrow$ conservative approximation.

- $d$: $\tau = a_1 \circ \mathcal{P} a_2$ is loop-invariant within loop $L$ if for each $a_i$:
  1. $a_i$ is constant, or
  2. all definitions of $a_i$ that reach $d$ are outside $L$, or
  3. only one definition of $a_i$ reaches $d$, and is loop-invariant.
Loop Invariant Computation: Algorithm

Iterative algorithm for determining loop-invariant computations:

mark "invariant" all definitions whose operands
- are constant, or
- whose reaching definitions are outside loop.

WHILE (changes have occurred)
  mark "invariant" all definitions whose operands
  - are constant,
  - whose reaching definitions are outside loop, or
  - which have a single reaching definition in loop
    marked invariant.
Loop Invariant Code Motion

After detecting loop-invariant computations, perform code motion.

1: \( r1 = 0 \)

2: \( r2 = 5 \)

Preheader:

3: \( r3 = r3 + 1 \)

4: \( r1 = r2 + 10 \)

5: \( M[r3] = r1 \)

6: \( \text{branch } r3 < N \)

7: \( r4 = r1 \)

Subject to some constraints.
Loop Invariant Code Motion: Constraint 1

\[ d : t = a \, \text{op} \, b \]

\( d \) must dominate all loop exit nodes where \( t \) is live out.

```
1: r1 = 0
2: r2 = 5

Preheader:

3: branch r3 < N

8: r4 = r1
4: r3 = r3 + 1
5: r1 = r2 + 10
6: M[r3] = r1
7: jump
```
Loop Invariant Code Motion: Constraint 2

\[ d: \ t = a \ \text{op} \ b \]

there must be only one definition of \( t \) inside loop.

1: \( r1 = 0 \)

2: \( r2 = 5 \)

Preheader:

3: \( r3 = r3 + 1 \)

4: \( r1 = r2 + 10 \)

5: \( M[r3] = r1 \)

6: \( r1 = 0 \)

7: \( M[r3] = r1 \)

8: branch \( r3 < N \)

9: [empty block]
Loop Invariant Code Motion: Constraint 3

d: \( t = a \ op \ b \)

\( t \) must not be live-out of loop preheader node (live-in to loop)

1: \( r1 = 0 \)
2: \( r2 = 5 \)

Preheader:

3: \( M[r3] = r1 \)
4: \( r3 = r3 + 1 \)
5: \( r1 = r2 + 10 \)
6: \( M[r3] = r1 \)
7: \( \text{branch } r3 < N \)
8: \( r4 = r1 \)
Loop Invariant Code Motion

Algorithm for code motion:

- Examine invariant statements of $L$ in same order in which they were marked.
- If invariant statement $s$ satisfies three criteria for code motion, remove $s$ from $L$, and insert into preheader node of $L$. 
Induction Variables

Variable $i$ in loop $L$ is called induction variable of $L$ if each time $i$ changes value in $L$, it is incremented/decremented by loop-invariant value.

Assume $a, c$ loop-invariant.

- $i$ is an induction variable
- $j$ is an induction variable

- $j = i \times c$ is equivalent to $j = j + a \times c$

- compute $e = a \times c$ outside loop:
  $j = j + e \Rightarrow$ strength reduction

- may not need to use $i$ in loop $\Rightarrow$ induction variable elimination
Induction Variable Detection

Scan loop $L$ for two classes of induction variables:

- **basic** induction variables - variables ($i$) whose only definitions within $L$ are of the form $i = i + c$ or $i = i - c$, $c$ is loop invariant.

- **derived** induction variables - variables ($j$) defined only once within $L$, whose value is linear function of some basic induction variable $L$.

Associate triple $(i, a, b)$ with each induction variable $j$

- $i$ is basic induction variable; $a$ and $b$ are loop invariant.

- value of $j$ at point of definition is $a + b \times i$

- $j$ belongs to the family of $i$
Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- Scan statements of $L$ for basic induction variables $i$
  - for each $i$, associate triple $(i, 0, 1)$
  - $i$ belongs to its own family.

- Scan statements of $L$ for derived induction variables $k$:
  1. there must be single assignment to $k$ within $L$ of the form $k = j \times c$ or $k = j + d$, $j$ is an induction variable; $c, d$ loop-invariant, and
  2. if $j$ is a derived induction variable belonging to the family of $i$, then:
     - the only definition of $j$ that reaches $k$ must be one in $L$, and
     - no definition of $i$ must occur on any path between definition of $j$ and definition of $k$

- Assume $j$ associated with triple $(i, a, b): j = a + b \times i$ at point of definition.

- Can determine triple for $k$ based on triple for $j$ and instruction defining $k$:
\(-k = j \cdot c \rightarrow (i, a \cdot c, b \cdot c)\)
\(-k = j + d \rightarrow (i, a + d, b)\)
Induction Variable Detection: Example

```
s = 0;
for(i = 0; i < N; i++)
    s += a[i];
```
1: r1 = 0
2: r2 = 0

Preheader:

3: branch r2 >= N
4: r3 = r2 * 4
5: r4 = r3 + a
6: r5 = M[r4]
7: r1 = r1 + r5
8: r2 = r2 + 1
9: jump
10: 

Computer Science 320
Prof. David Walker
Strength Reduction

1. For each derived induction variable $j$ with triple $(i, a, b)$, create new $j'$.
   
   • all derived induction variables with same triple $(i, a, b)$ may share $j'$

2. After each definition of $i$ in $L, i = i + c$, insert statement:
   
   $j' = j' + b * c$

   • $b * c$ is loop-invariant and may be computed in preheader or during compile time.

3. Replace unique assignment to $j$ with $j = j'$.

4. Initialize $j'$ at end of preheader node:

   $j' = b * i$
   
   $j' = j' + a$

   • Strength reduction still requires multiplication, but multiplication now performed outside loop.

   • $j'$ also has triple $(i, a, b)$
Strength Reduction: Example

1: \( r_1 = 0 \)

2: \( r_2 = 0 \)

Preheader:

3: branch \( r_2 \geq N \)

10: \( r_1 = r_1 + r_5 \)

4: \( r_3 = r_2 \times 4 \)

5: \( r_4 = r_3 + a \)

6: \( r_5 = M[r_4] \)

7: \( r_1 = r_1 + r_5 \)

8: \( r_2 = r_2 + 1 \)

9: jump
**Strength Reduction: Example**

1: \( r_1 = 0 \)

2: \( r_2 = 0 \)

Preheader:
- \( r_{33} = r_2 \times 4 \)
- \( r_{33} = r_{33} + 0 \)
- \( r_{44} = r_2 \times 4 \)
- \( r_{44} = r_{44} + a \)

3: branch \( r_2 \geq N \)

4: \( r_3 = r_{33} \)

5: \( r_4 = r_{44} \)

6: \( r_5 = M[r_4] \)

7: \( r_1 = r_1 + r_5 \)

8: \( r_2 = r_2 + 1 \)

8': \( r_{33} = r_{33} + 4 \)

8'': \( r_{44} = r_{44} + 4 \)

9: jump
Induction Variable Elimination

After strength reduction has been performed:

- some induction variables are only used in comparisons with loop-invariant values.
- some induction variables are *useless*
  - dead on all loop exits, used only in definition of itself.
  - dead code elimination will not remove useless induction variables.
Induction Variable Elimination: Example

1: \( r_1 = 0 \)
2: \( r_2 = 0 \)

Preheader:

3: \( r_{33} = 0 \)
   \( r_44 = a \)
4: \( \text{branch } r_2 \geq N \)

5: \( r_4 = r_44 \)
6: \( r_5 = M[r_4] \)
7: \( r_1 = r_1 + r_5 \)
8: \( r_2 = r_2 + 1 \)
8': \( r_{33} = r_{33} + 4 \)
8'': \( r_44 = r_44 + 4 \)
9: \( \text{jump} \)

Computer Science 320
Prof. David Walker
Induction Variable Elimination

- Variable \( k \) is *almost useless* if it is only used in comparisons with loop-invariant values, and there exists another induction variable \( t \) in the same family as \( k \) that is not useless.

- Replace \( k \) in comparison with \( t \)
  \[ \rightarrow k \text{ is useless} \]
Induction Variable Elimination: Example

1: \( r1 = 0 \)
2: \( r2 = 0 \)

Preheader:

\( r44 = a \)

3: branch \( r2 \geq N \)

5: \( r4 = r44 \)

6: \( r5 = M[r4] \)

7: \( r1 = r1 + r5 \)

8: \( r2 = r2 + 1 \)

8': \( r44 = r44 + 4 \)

9: jump
Induction Variable Elimination: Example

1: \( r1 = 0 \)

2: \( r2 = 0 \)

Preheader:

- \( r44 = a \)
- \( r100 = 4 \times N \)
- \( r101 = r100 + a \)

3: \( \text{branch } r44 \geq r101 \)

10: 

5: \( r4 = r44 \)

6: \( r5 = M[r4] \)

7: \( r1 = r1 + r5 \)

8: \( r2 = r2 + 1 \)

8\': \( r44 = r44 + 4 \)

9: \( \text{jump} \)