15-745 Lecture 5

Control flow analysis
Natural loops
Classical Loop Optimizations
 Dependencies

Loops are Key

• Loops are extremely important
  - the “90-10” rule
• Loop optimization involves
  - understanding control-flow structure
  - Understanding data-dependence information
  - sensitivity to side-effecting operations
  - extra care in some transformations such as register spilling

Common loop optimizations

• Hoisting of loop-invariant computations
  - pre-compute before entering the loop
• Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
• Loop unrolling
  - to improve scheduling of the loop body
• Software pipelining
  - To improve scheduling of the loop body
• Loop permutation
  - to improve cache memory performance

Finding Loops

• To optimize loops, we need to find them!
• Could use source language loop information in the abstract syntax tree...
• BUT:
  - There are multiple source loop constructs: for, while, do-while, even goto in C
  - Want IR to support different languages
  - Ideally, we want a single concept of a loop so all have same analysis, same optimizations
  - Solution: dismantle source-level constructs, then re-find loops from fundamentals

Copyright © Seth Goldstein, 2008

Based on slides from Lee & Callahan
Finding Loops

- To optimize loops, we need to find them!
- Specifically:
  - loop-header node(s)
    - nodes in a loop that have immediate predecessors not in the loop
  - back edge(s)
    - control-flow edges to previously executed nodes
  - all nodes in the loop body

Control-flow analysis

- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called control-flow analysis
- Based on the notion of dominators

Dominators

- $a \text{ dom } b$
  - node $a$ dominates $b$ if every possible execution path from entry to $b$ includes $a$
- $a \text{ sdom } b$
  - $a$ strictly dominates $b$ if $a \text{ dom } b$ and $a \neq b$
- $a \text{ idom } b$
  - $a$ immediately dominates $b$ if $a \text{ sdom } b$, AND there is no $c$ such that $a \text{ sdom } c$ and $c \text{ sdom } b$

Back edges and loop headers

- A control-flow edge from node $B_3$ to $B_2$ is a back edge if $B_2 \text{ dom } B_3$
- Furthermore, in that case node $B_2$ is a loop header
Natural loop

- Consider a back edge from node n to node h
- The natural loop of $n \rightarrow h$ is the set of nodes $L$ such that for all $x \in L$:
  - $h \in dom x$ and
  - there is a path from $x$ to $n$ not containing $h$

Examples

Simple example:

```
(a, b, c)
```

(often it's more complicated, since a FOR loop found in the source code might need an if/then guard)

Examples

Try this:

```
for (...) {
  if {
    ...
  } else {
    ...
    if (x) {
      e;
      break;
    }
  }
}
```
**Examples**

for (..) {  
    if {  
    } else {  
    }  
}  

if (x) {  
    e;  
    break;  
}

Examples

- Yes, it can happen in C

Natural Loops

What are the natural loops?

One loop per header..
Nested Loops

- Unless two natural loops have the same header, they are either disjoint or nested within each other.
- If A and B are loops (sets of blocks) with headers a and b such that \( a \neq b \) and \( b \in A \):
  - B is in A
  - Loop B is nested within A
- B is the inner loop
- Can compute the loop-nest tree

General Loops

- A more general looping structure is a strongly connected component of the control flow graph.
  - Subgraph \( <N_{scc}, E_{scc}> \) such that:
    - Every block in \( N_{scc} \) is reachable from every other node using only edges in \( E_{scc} \).

Reducible Flow Graphs

There is a special class of flow graphs, called reducible flow graphs, for which several code-optimizations are especially easy to perform.

In reducible flow graphs, loops are unambiguously defined and dominators can be efficiently computed.

Reducible flow graphs

**Definition:** A flow graph \( G \) is reducible iff we can partition the edges into two disjoint groups, forward edges and back edges, with the following two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of \( G \).
2. The back edges consist only of edges whose heads dominate their tails.

Why isn't this reducible?

This flow graph has no back edges. Thus, it would be reducible if the entire graph were acyclic, which is not the case.
Alternative definition

- **Definition:** A flow graph $G$ is reducible if we can repeatedly collapse (reduce) together blocks $(x,y)$ where $x$ is the only predecessor of $y$ (ignoring self-loops) until we are left with a single node.

Properties of Reducible Flow Graphs

- In a reducible flow graph, all loops are natural loops.
- Can use DFS to find loops.
- Many analyses are more efficient
  - polynomial versus exponential

Good News

- Most flow graphs are reducible.
- Languages prohibit irreducibility
  - goto free C
  - Java
- Programmers usually don't use such constructs even if they're available
  - $>$90% of old Fortran code reducible

Dealing with Irreducibility

- Don't
- Can split nodes and duplicate code to get reducible graph
  - possible exponential blowup
- Other techniques...
Loop optimizations:
Hoisting of loop-invariant computations

Loop-invariant computations

- A definition
  \[ t = x \text{ op } y \]
in a loop is (conservatively) loop-invariant if
  - \( x \) and \( y \) are constants, or
  - all reaching definitions of \( x \) and \( y \) are
    outside the loop, or
  - only one definition reaches \( x \) (or \( y \)), and
    that definition is loop-invariant
  - so keep marking iteratively

Loop-invariant computations

- Be careful:
  Of course, not an issue in SSA

\[
\begin{align*}
  t &= \text{expr}; \\
  \text{for} () \{ \\
    s &= t * 2; \\
    t &= \text{loop_inv} \\
    x &= t + 2; \\
    \ldots \\
  \}
\]

- Even though \( t \)'s two\ L2:
  each invariant, \( s \) is not invariant...

Hoisting

- In order to "hoist" a loop-invariant computation out of a loop, we need a place to put it
  - We could copy it to all immediate predecessors
    (except along the back-edge) of the loop header...
  - ...But we can avoid code duplication by inserting
    a new block, called the pre-header
Hoisting conditions

- For a loop-invariant definition
  \( d: t = x \text{ op } y \)
- we can hoist \( d \) into the loop's pre-header only if
  1. \( d \)'s block dominates all loop exits at which \( t \) is live-out, and
  2. \( d \) is only the only definition of \( t \) in the loop, and
  3. \( t \) is not live-out of the pre-header

We need to be careful...

- All hoisting conditions must be satisfied!

- \( \begin{align*}
  \text{L0:} & \quad t = 0 \\
  \text{L1:} & \quad i = i + 1 \\
  & \quad t = a * b \\
  & \quad M[i] = t \\
  & \quad \text{if } i < N \text{ goto L1} \\
  \text{L2:} & \quad x = t
\end{align*} \)

- \( \begin{align*}
  \text{L0:} & \quad t = 0 \\
  \text{L1:} & \quad i = i + 1 \\
  & \quad t = a * b \\
  & \quad M[i] = t \\
  & \quad \text{if } i < N \text{ goto L1} \\
  \text{L2:} & \quad x = t
\end{align*} \)

- \( \begin{align*}
  \text{L0:} & \quad t = 0 \\
  \text{L1:} & \quad i = i + 1 \\
  & \quad t = a * b \\
  & \quad M[i] = t \\
  & \quad t = 0 \\
  & \quad M[j] = t \\
  & \quad \text{if } i < N \text{ goto L1} \\
  \text{L2:} & \quad x = t
\end{align*} \)

OK violates 1, 3 violates 2
Loop optimizations:
Induction-variable Strength reduction

The basic idea of IVE

- Suppose we have a loop variable
  - $i$ initially 0; each iteration $i = i + 1$

- and a variable that linearly depends on it:
  - $x = i \times c1 + c2$

- In such cases, we can try to
  - initialize $x = i_0 \times c1 + c2$ (execute once)
  - increment $x$ by $c1$ each iteration

Simple Example of IVE

```plaintext
for i = 0 to n
  a[i] = 0
  H:
    i <- 0
    if i >= n goto exit
    j <- i \times 4
    k <- j + a
    M[k] <- 0
    i <- i + 1
    goto H
```

Clearly, $j$ & $k$ do not need to be computed anew each time since they are related to $i$ and $i$ changes linearly.

But, then we don't even need $j$ (or $j'$)
Simple Example of IVE

```cpp
i <- 0
j' <- 0
k' <- a

if i >= n goto exit
j <- j'
k <- k'
M[k] <= 0
i <- i + 1
j' <- j' + 4
k' <- k' + 4
goto H
```

Do we need i?

But, a+(n*4) is loop invariant

Invariant code motion on a+(n*4)

```cpp
i <- 0
k' <- a

if k' >= a+(n*4) goto exit
k <- k'
M[k] <= 0
k' <- k' + 4
goto H
```

now, we do copy propagation and eliminate k

Copy propagation

```cpp
k' <- a
n' <- a + (n * 4)

if k' >= n' goto exit
k <- k'
M[k] <= 0
k' <- k' + 4
goto H
```

Voila!

```cpp
k' <- a
n' <- a + (n * 4)

if k' >= n' goto exit
k <- k'
M[k] <= 0
k' <- k' + 4
goto H
```
Simple Example of IVE

Compare original and result of IVE

\[
\begin{align*}
i &\leftarrow 0 \\
H: & \quad \text{if } i \geq n \text{ goto exit} \\
j &\leftarrow i \times 4 \\
k &\leftarrow j + a \\
M[k] &\leftarrow 0 \\
i &\leftarrow i + 1 \\
goto H
\end{align*}
\]

\[
\begin{align*}
k' &\leftarrow a \\
n' &\leftarrow a + (n \times 4) \\
H: & \quad \text{if } k' \geq n' \text{ goto exit} \\
M[k'] &\leftarrow 0 \\
k' &\leftarrow k' + 4 \\
goto H
\end{align*}
\]

Voila!

What we did

- identified induction variables \((i,j,k)\)
- strength reduction (changed \(*\) into \(+\))
- dead-code elimination \((j \leftarrow j')\)
- useless-variable elimination \((j' \leftarrow j' + 4)\)
  (This can also be done with ADCE)
- loop invariant identification & code-motion
- almost useless-variable elimination \((i)\)
- copy propagation

Is it faster?

- On some hardware, adds are much faster than multiplies
- Furthermore, one fewer value is computed,
  - thus potentially saving a register
  - and decreasing the possibility of spilling

Loop preparation

- Before attempting IVE, it is best to first perform:
  - constant propagation & constant folding
  - copy propagation
  - loop-invariant hoisting
How to do it, step 1

- First, find the basic IVs
  - scan loop body for defs of the form
    \( x = x + c \) or \( x = x - c \)
    - where \( c \) is loop-invariant
  - record these basic IVs as
    \( x = (x, 1, c) \)
  - this represents the IV: \( x = x \cdot 1 + c \)

Representing IVs

- Characterize all induction variables by:
  (base-variable, offset, multiple)
  - where the offset and multiple are loop-invariant
- IOW, after an induction variable is defined it equals:
  \[ \text{offset} + \text{multiple} \cdot \text{base-variable} \]

How to do it, step 2

- Scan for derived IVs of the form
  \( k = i \cdot c_1 + c_2 \)
  - where \( i \) is a basic IV,
    - this is the only def of \( k \) in the loop, and
    \( c_1 \) and \( c_2 \) are loop invariant
- We say \( k \) is in the family of \( i \)
- Record as \( k = (i, c_1, c_2) \)

How to do it, step 3

- Iterate, looking for derived IVs of the form
  \( k = j \cdot c_1 + c_2 \)
  - where IV \( j = (i, a, b) \), and
    \( c_1 \) and \( c_2 \) are loop invariant
- We say \( k \) is in the family of \( j \)
- Record as \( k = (i, ac_1, bc_1 + c_2) \)
Simple Example of IVE

```
i <- 0
H:
if i >= n goto exit
j <- i * 4
k <- j + a
M[k] <- 0
i <- i + 1
goto H
```

i: (i, 1, 1) i.e., i = 1 + 1 * i
j: (i, 0, 4) i.e., j = 0 + 4 * i
k: (i, a, 4) i.e., k = a + 4 * i

So, j & k are in family of i

Finding the IVs

• Maintain three tables: basic & maybe & other
• Find basic IVs:
  Scan stmts. If var ∉ maybe, and of proper form, put into basic. Otherwise, put var in other and remove from maybe.
• Find compound IVs:
  - If var defined more than once, put into other
  - For all stmts of proper form that use a basic IV

How to do it, step 4

• This is the strength reduction step
• For an induction variable k = (i, c1, c2)
  - initialize k = i * c1 + c2 in the preheader
  - replace k's def in the loop by
    k = k + c1
  - make sure to do this after i's def

How to do it, step 5

• This is the comparison rewriting step
• For an induction variable k = (i, ak, bk)
  - If k used only in definition and comparison
  - There exists another variable, j, in the same class and is not "useless" and j=(i, aj, bj)
  - Rewrite k < n as
    j < (bj/bk)(n-ak)+aj
• Note: since they are in same class:
  (j-aj)/bj = (k-ak)/bk
Notes

• Are the c1, c2 constant, or just invariant?
  - if constant, then you can keep folding them: they're always a constant even for derived IVs
  - otherwise, they can be expressions of loop-invariant variables

• But if constant, can find IVs of the type
  \[ x = \frac{i}{b} \]
  and know that it’s legal, if b evenly divides the stride...

Is it faster? (2)

• On some hardware, adds are much faster than multiplies
  • But...not always a win!
    - Constant multiplies might otherwise be reduced to shifts/adds that result in even better code than IVE
    - Scaling of addresses \((i*4)\) might come for free on your processor’s address modes
  • So maybe: only convert \(i*c1+c2\) when \(c1\) is loop invariant but not a constant

Common loop optimizations

• Hoisting of loop-invariant computations
  - pre-compute before entering the loop

• Elimination of induction variables
  - change \(p=i*w+b\) to \(p=b, p+=w\), when \(w, b\) invariant

• Loop unrolling
  - to improve scheduling of the loop body

<table>
<thead>
<tr>
<th>Software pipelining</th>
<th>Requires understanding data dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>To improve scheduling of the loop body</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>to improve cache memory performance</td>
</tr>
</tbody>
</table>