Loop transformations
Loops

- Almost all execution time is inside loops
- Many optimizations are centered around transforming loops
  - Loop invariant code motion: avoid re-computing expressions by hoisting them out of the loop
  - Loop unrolling: avoid branching by executing several iterations of a loop at a time
  - Strength reduction: replace a costly operation (e.g., multiplication) inside a loop with a cheaper one (e.g., addition)
  - Lots more: parallelization, tiling, vectorization, ...
What is a loop?

• We're after a graph-theoretic definition of a loop
  • Not sensitive to syntax of source language (loops can be created with while, for, goto, ...)

• First attempt: SCCs
  • Not fine enough – nested loops have only one SCC, but we want to transform them separately
  • Too general – makes it difficult to apply transformations

• Desiderata:
  • Many loop optimizations require inserting code immediately before the loop enters, so loop definition should make that easy
  • Want to at least capture loops that would result from structured programming (programs built with while, if, and sequencing, no goto)
What is a loop?

• A **loop** of a control flow graph is a set of nodes $S$ such that
  1. $S$ is strongly connected
  2. There is a *header* node $h$ that dominates all nodes in $S$
  3. There is no edge from any node *outside* of $S$ to any node *inside* of $S$, except for $h$

• A *loop entry* is a node with some predecessor outside the loop
• A *loop exit* is a node with some successor outside the loop
• A loop has one entry, but may have multiple exits (or none)
Strongly connected subgraph

Dominator tree
Identifying loops

• A **back edge** is an edge $u \rightarrow v$ such that $v$ dominates $u$.

• The **natural loop** of a back edge $u \rightarrow v$ is the set of nodes $n$ such that $v$ dominates $n$ and there is a path from $n$ to $u$ not containing $v$.

  • The natural loop of a back edge can be computed with a DFS on the *reversal* of the CFG, starting from $v$.

• Every natural loop is a loop, but not every loop is natural.

  • Every node that reaches $u$ without going through $v$ is dominated by $v$ (otherwise, $v$ does not dominate $u$ – contradiction).

  • Suppose that a node $n$ in the natural loop has a predecessor outside of the natural loop.

    • That predecessor has a path to $u$ that doesn’t go through $v$, so it belongs to the loop by definition. Again, contradiction.
Nested loops

• Say that a loop $B$ is *nested* within $A$ if $B \subseteq A$
• A node can be the header of more than one natural loop.
  • Neither is nested inside the other
  • Commonly, we merge natural loops with the same header
• Loops obtained by merging natural loops with the same header are either disjoint or nested
• We typically apply loop transformations “bottom-up”, starting with innermost loops
Loop preheaders

- Some optimizations (e.g., loop-invariant code motion) require inserting statements immediately before a loop executes
- A loop preheader is a basic block that is inserted immediately before the loop header, to serve as a place to store these statements
Loop invariant code motion

- Loop invariant code motion saves the cost of re-computing expressions that are left invariant (i.e., do not change) in the loop.
  - Such computations can be moved the loop's preheader, as long as they are not side-effecting
- SSA based LICM:
  - An operand is *invariant* in a loop $L$ if
    1. It is a constant, or
    2. It is a gid, or
    3. It is a uid, whose definition does not belong to $L$
  - For each computation $%x = opn_1 \ op \ opn_2$, if $opn_1$ and $opn_2$ are both invariant, move $%x = opn_1 \ op \ opn_2$ to pre-header
  - This moves definition of $%x$ outside of the loop, so $%x$ is now invariant
%i_0 = 0
br loop

%i_1 = \phi(%i_0, %i_2)
%t_1 = %n * %n
%t_2 = %t_1 * %n
%t_3 = %i_1 - %t_2
blz %t_3, body, exit

%t_2 = %i_1 + 1
b loop

return %i_1
\%i_0 = 0 \\
br ph \\

br loop \\

\%i_1 = \phi(\%i_0, \%i_2) \\
\%t1 = \%n \times \%n \\
\%t2 = \%t1 \times \%n \\
\%t3 = \%i_1 - \%t2 \\
blz \%t3, body, exit \\

\%i_2 = \%i_1 + 1 \\
b loop \\

return \%i_1
\%i_0 = 0
br ph

\%t1 = \%n \times \%n
br loop

\%i_1 = \phi(\%i_0, \%i_2)
\%t2 = \%t1 \times \%n
\%t3 = \%i_1 - \%t2
blz \%t3, body, exit

\%i_2 = \%i_1 + 1
b loop

return \%i_1
\[ %i_0 = 0 \]
\[ \text{br ph} \]
\[ %t_1 = %n \times %n \]
\[ %t_2 = %t_1 \times %n \]
\[ \text{br loop} \]

\[ %i_1 = \phi(%i_0, %i_2) \]
\[ %t_3 = %i_1 - %t_2 \]
\[ \text{blz} \ %t_3, \ \text{body, exit} \]

\[ %i_2 = %i_1 + 1 \]
\[ \text{b loop} \]

\[ \text{return} \ %i_1 \]
Induction variables

- A variable $\%x$ is an **basic induction variable** for a loop $L$ if it is increased / decreased by a fixed amount in any iteration of the loop.
- A variable $\%y$ is an **derived induction variable** for a loop $L$ if it is an affine function of a basic induction variable ($\%y = a \cdot \%x + b$ for some loop invariant quantities $a$ and $b$).
Induction variables

- A variable $\% x$ is an **basic induction variable** for a loop $L$ if it is increased / decreased by a fixed amount in any iteration of the loop.
- A variable $\% y$ is an **derived induction variable** for a loop $L$ if it is an affine function of a basic induction variable ($\% y = a \cdot \% x + b$ for some loop invariant quantities $a$ and $b$).
- To detect basic induction variables:
  - Look for $\phi$ statements $\% x = \phi(\% x_1, \ldots, \% x_n)$ in header
  - Each position $\% x_i$ corresponding to a back edge of the loop must be the same uid, say $\% x_k$
  - Find chain of assignments for $\% x_k$ leading back to $\% x$, such that each either adds or subtracts an invariant quantity. Success $\Rightarrow \% x$ is an basic induction var.
Induction variables

- A variable \( x \) is an **basic induction variable** for a loop \( L \) if it is increased / decreased by a fixed amount in any iteration of the loop.
- A variable \( y \) is an **derived induction variable** for a loop \( L \) if it is an affine function of a basic induction variable (\( y = a \cdot x + b \) for some loop invariant quantities \( a \) and \( b \)).
- To detect basic induction variables:
  - Look for \( \phi \) statements \( x = \phi(x_1, \ldots, x_n) \) in header
  - Each position \( x_i \) corresponding to a back edge of the loop must be the same uid, say \( x_k \)
  - Find chain of assignments for \( x_k \) leading back to \( x \), such that each either adds or subtracts an invariant quantity. Success \( \Rightarrow x \) is a basic induction var.
- To detect derived induction variables:
  - Choose a basic induction variable \( x \)
  - Find assignments of the form \( y = opn_1 \ op \ opn_2 \) where
    - \( op \) is + or − and \( opn_1 \) and \( opn_2 \) are either \( x \), derived induction variables of \( x \), or loop invariant quantities
    - \( op \) is \( \ast \) and \( opn_1 \) and \( opn_2 \) are as above, and at least one is a loop invariant quantity
Strength reduction

Idea: replace expensive operation (e.g., multiplication) w/ cheaper one (e.g., addition).

```c
long trace (long *m, long n) {  
    long i;
    long result = 0;
    for (i = 0; i < n; i++) {  
        result += *(m + i*n + i);
    }
    return result;
}
```

```c
long trace (long *m, long n) {  
    long i;
    long result = 0;
    long *next = m;
    for (i = 0; i < n; i++) {  
        result += *next;
        next += i + 1;
    }
    return result;
}
```
%i_1 = \phi(%i_0, %i_2)
%result_1 = \phi(%result_0, %result_2)
%t1 = %i_1 - %n
blz %t1, body, exit

%t2 = %i_1 * %n
%t3 = %m + %t2
%t4 = %t3 + %i_1
%t5 = load %t4
%result_2 = %result_1 + %t5
%i_2 = %i_1 + 1
b loop
%i_1 = \phi(%i_0, %i_2)
%result_1 = \phi(%result_0, %result_2)
%t1 = %i_1 - %n
blz %t1, body, exit

%t2 = %i_1 * %n
%t3 = %m + %t2
%t4 = %t3 + %i_1
%t5 = load %t4
%result_2 = %result_1 + %t5
%i_2 = %i_1 + 1
b loop
\[ i_1 = \phi(i_0, i_2) \]

\[ \text{result}_1 = \phi(\text{result}_0, \text{result}_2) \]

\[ t_1 = i_1 - n \]

`blz t_1, body, exit`

\[ t_2 = i_1 \times n \]

\[ t_3 = m + t_2 \]

\[ t_4 = t_3 + i_1 \]

\[ t_5 = \text{load } t_4 \]

\[ \text{result}_2 = \text{result}_1 + t_5 \]

\[ i_2 = i_1 + 1 \]

`b loop`
$i_1 = \phi(i_0, i_2)$ 

$i := i + 1$

$\textit{result}_1 = \phi(\textit{result}_0, \textit{result}_2)$

$t_1 = i_1 - n$

$\text{blz } t_1, \text{ body, exit}$

$t_2 = i_1 \times n$

t_3 = m + t_2$

t_4 = t_3 + i_1$

t_5 = \text{load } t_4$

$\textit{result}_2 = \textit{result}_1 + t_5$

$i_2 = i_1 + 1$

b loop
%i_1 = \phi(\%i_0, \%i_2) 
\%result_1 = \phi(\%result_0, \%result_2) 
\%t_1 = \%i_1 - \%n 
\text{t1} := i + n 
\text{blz} \; \%t_1, \text{body, exit} 

\%t_2 = \%i_1 * \%n 
\text{t2} := n \times i 
\%t_3 = \%m + \%t_2 
\%t_4 = \%t_3 + \%i_1 
\%t_5 = \text{load} \; \%t_4 
\%result_2 = \%result_1 + \%t_5 
\%i_2 = \%i_1 + 1 
\text{b loop}
\[ i_1 = \phi(i_0, i_2) \]
\[ \text{result}_1 = \phi(\text{result}_0, \text{result}_2) \]
\[ t_1 = i_1 - n \]
\[ \text{t1 := i + n} \]
\[ \text{blz t1, body, exit} \]

\[ t_2 = i_1 \times n \]
\[ t_3 = m + t_2 \]
\[ t_4 = t_3 + i_1 \]
\[ t_5 = \text{load t4} \]
\[ \text{result}_2 = \text{result}_1 + t_5 \]
\[ i_2 = i_1 + 1 \]
\[ \text{b loop} \]
\[ i_1 = \phi(i_0, i_2) \quad i := i + 1 \]
\[ \text{result}_1 = \phi(\text{result}_0, \text{result}_2) \]
\[ t_1 = i_1 - n \quad t1 := i + n \]
blz \( t_1 \), body, exit

\[ t_2 = i_1 \times n \quad t2 := n \times i \]
\[ t_3 = m + t_2 \quad t3 := n \times i + m \]
\[ t_4 = t_3 + i_1 \quad t4 := (n+1) \times i + m \]
\[ t_5 = \text{load } t_4 \]
\[ \text{result}_2 = \text{result}_1 + t_5 \]
\[ i_2 = i_1 + 1 \]
bl loop
\[\begin{align*}
&t_2^0 = 0 \\
&t_3^0 = m \\
&t_4^0 = m \\

&i_1 = \phi(i_0, i_2) \quad i := i + 1 \\
&t_2^1 = \phi(t_2^0, t_2^2) \\
&t_3^1 = \phi(t_3^0, t_3^2) \\
&t_4^1 = \phi(t_4^0, t_4^2) \\
\text{result}_1 = \phi(\text{result}_0, \text{result}_2) \\
&t_1 = i_1 - n \quad t_1 := i + n \\
\text{blz } t_1, \text{ body, exit} \\

&t_{22} = t_{21} + n \quad t_2 := n \cdot i \\
&t_{32} = t_{31} + n \quad t_3 := n \cdot i + m \\
&t_{6} = t_{42} + n \quad t_4 := (n+1) \cdot i + m \\
&t_{42} = t_{6} + 1 \\
&t_5 = \text{load } t_{42} \\
\text{result}_2 = \text{result}_1 + t_5 \\
&i_2 = i_1 + 1 \\
\text{b loop}
\end{align*}\]
Loop unrolling

- Some loops are so small that a significant portion of the running time is due to testing the loop exit condition
- We can avoid branching by executing several iterations of the loop at once
- This optimization trades (potential) run-time performance with code size.
• Given a loop $L$ with header $h$ Suppose loop exit is \texttt{blz t, body, exit}, where $t$ is a derived induction variable $t = a \cdot i + b$ with $i$ a basic induction variable $i := i + c$
• Given a loop $L$ with header $h$ Suppose loop exit is $blz\ t,\ body,\ exit$, where $t$ is a derived induction variable $t = a \cdot i + b$ with $i$ a basic induction variable $i := i + c$

• Copy all nodes in $L$ to make a loop $L'$ with header $h'$

• Redirect back edges in $L$ to $h'$

• Redirect back edges in $L'$ to $h$
Given a loop $L$ with header $h$ Suppose loop exit is $\text{blz } t, \text{body, exit}$, where $t$ is a derived induction variable $t = a \cdot i + b$ with $i$ a basic induction variable $i := i + c$

- Copy all nodes in $L$ to make a loop $L'$ with header $h'$
- Redirect back edges in $L$ to $h'$
- Redirect back edges in $L'$ to $h$
- In $h$, replace $\text{blz } t, \text{body, exit}$ with $\text{blz } (t + a \cdot c), \text{body, exit}$
- In $h'$, replace $\text{blz } t, \text{body, exit}$ with $b \text{ body}$
Given a loop $L$ with header $h$ Suppose loop exit is $\text{blz } t, \text{body}, \text{exit}$, where $t$ is a derived induction variable $t = a \cdot i + b$ with $i$ a basic induction variable $i := i + c$

- Copy all nodes in $L$ to make a loop $L'$ with header $h'$
- Redirect back edges in $L$ to $h'$
- Redirect back edges in $L'$ to $h$
- In $h$, replace $\text{blz } t, \text{body}, \text{exit}$ w/ $\text{blz } (t + a \cdot c), \text{body}, \text{exit}$
- In $h'$, replace $\text{blz } t, \text{body}, \text{exit}$ w/ $b \text{ body}$
- Add loop epilogue to execute last iteration, if needed
%t2_0 = 0
%t3_0 = %m
%t4_0 = %m

%i_1 = \phi(%i_0, %i_2')
%t2_1 = \phi(%t2_0, %t2_2')
%t3_1 = \phi(%t3_0, %t3_2')
%t4_1 = \phi(%t4_0, %t4_2')
%result_1 = \phi(%result_0, %result_2')
%t1 = %i_1 - %n
%t12 = %t1 + 1
blz %t12, body, epilogue

%t2_2 = %t2_1 + %n
%t3_2 = %t3_1 + %n
%t6 = %t4_2 + %n
%t4_2 = %t6 + 1
%t5 = load %t4_2
%result_2 = %result_1 + %t5
%i_2' = %i_2 + 1
b loop'}
Optimization wrap-up

- Optimizer operates as a series of IR-to-IR transformations
- Transformations are typically supported by some analysis that proves the transformation is save
- Each transformation is simple
- Transformations are mutually beneficial
  - Series of transformations can make drastic changes!