COS320: Compiling Techniques

Zak Kincaid

April 16, 2024
Logistics

- Last HW is due on Dean’s date. You will implement:
  - The worklist algorithm for dataflow analysis
  - Constant propagation
  - Alias analysis & dead code elimination
  - Register allocation
Loop transformations
Loops

- Almost all execution time is inside loops
- Many optimizations are centered around transforming loops
  - Loop invariant code motion: hoist expressions out of loops to avoid re-computation
  - Strength reduction: replace a costly operation inside a loop with a cheaper one
  - Loop unrolling: avoid branching by executing several iterations of a loop
  - Lots more: parallelization, tiling, vectorization, ...
What is a loop?

- We're after a *graph-theoretic* definition of a loop
  - Typically no explicit loop syntax at the IR level
  - Not sensitive to syntax of source language (loops can be created with *while*, *for*, *goto*, ...)

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

- We're after a *graph-theoretic* definition of a loop
  - Typically no explicit loop syntax at the IR level
  - Not sensitive to syntax of source language (loops can be created with *while*, *for*, *goto*, ...)
- First attempt: strongly connected components (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
What is a loop?

- We're after a *graph-theoretic* definition of a loop
  - Typically no explicit loop syntax at the IR level
  - Not sensitive to syntax of source language (loops can be created with `while`, `for`, `goto`, ...)
- First attempt: strongly connected components (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:
  - Want to *at least* capture loops that would result from structured programming (programs built with `while`, `if`, and sequencing (no `goto`!))
  - Many loop optimizations require inserting code *immediately before* the loop enters, so loop definition should make that easy
What is a loop?

• A loop of a control flow graph is a set of nodes $S$ such that with a distinguished header node $h$ such that

1. $S$ is strongly connected
   • There is a directed path from $h$ to every node in $S$
   • There is a directed path from any to in $S$ to $h$

2. There is no edge from any node outside of $S$ to any node inside of $S$, except for $h$
   • Implies $h$ dominates all nodes in $S$: every path from entry to a node in $S$ must go through $h$
What is a loop?

• A **loop** of a control flow graph is a set of nodes $S$ such that with a distinguished **header** node $h$ such that
  1. $S$ is strongly connected
     • There is a directed path from $h$ to every node in $S$
     • There is a directed path from any to in $S$ to $h$
  2. There is no edge from any node outside of $S$ to any node inside of $S$, except for $h$
     • Implies $h$ dominates all nodes in $S$: every path from entry to a node in $S$ must go through $h$

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

Dominator tree
Identifying loops

- A back edge is an edge $u \rightarrow v$ such that $v$ dominates $u$
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 $v$. 
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 $u$
Identifying loops

- A **back edge** is an edge \( u \to v \) such that \( v \) dominates \( u \).
- The **natural loop** of a back edge \( u \to v \) 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 \( u \).
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 $u$
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 $u$.
Every natural loop is a loop:

- By DFS construction, every node has a path to \( u \) (that doesn't pass through \( v \)).
- Every node has a path from \( v \) (path from entry to node to \( u \) must include \( v \)).

But not every loop is natural:
Every natural loop is a loop:

1. **Strongly connected**
   - By DFS construction every node has a path to \( u \) (that doesn’t pass through \( v \))
   - Every node has a path from \( v \) (path from entry to node to \( u \) must include \( v \))
Every natural loop is a loop:

1 Strongly connected
   - By DFS construction every node has a path to $u$ (that doesn’t pass through $v$)
   - Every node has a path from $v$ (path from entry to node to $u$ must include $v$)

2 Single entry $v$
Every natural loop is a loop:

1. Strongly connected
   - By DFS construction every node has a path to $u$ (that doesn’t pass through $v$)
   - Every node has a path from $v$ (path from entry to node to $u$ must include $v$)

2. Single entry $v$
   - By DFS construction, all predecessors of any node except $v$ belong to the loop
Every natural loop is a loop:

1. Strongly connected
   - By DFS construction every node has a path to $u$ (that doesn’t pass through $v$)
   - Every node has a path from $v$ (path from entry to node to $u$ must include $v$)

2. Single entry $v$
   - By DFS construction, all predecessors of any node except $v$ belong to the loop

But not every loop is natural:
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
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 resolve this issue by merging natural loops with the same header
  • Loops obtained by merging natural loops with the same header are either disjoint or nested
  • Loops can be organized into a forest
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 resolve this issue by merging natural loops with the same header
  • Loops obtained by merging natural loops with the same header are either disjoint or nested
  • Loops can be organized into a forest
• 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 to the loop’s preheader, as long as they are not side-effecting
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 \textit{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 $L$
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
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

%i_2 = %i_1 + 1
b loop

return %i_1
%i0 = 0
br ph

br loop

%i1 = \phi(%i0, %i2)
%t1 = %n * %n
%t2 = %t1 * %n
%t3 = %i1 - %t2
blz %t3, body, exit

%i2 = %i1 + 1
b loop

return %i1
\( \%i_0 = 0 \)
\( \text{br ph} \)

\( \%t_1 = \%n \times \%n \)
\( \text{br loop} \)

\( \%i_1 = \phi(\%i_0, \%i_2) \)
\( \%t_2 = \%t_1 \times \%n \)
\( \%t_3 = \%i_1 - \%t_2 \)
\( \text{blz} \%t_3, \text{body}, \text{exit} \)

\( \%i_2 = %i_1 + 1 \)
\( \text{b loop} \)

\( \text{return} \%i_1 \)
%i₀ = 0
br ph

%t₁ = %n * %n
%t₂ = %t₁ * %n
br loop

%ᵢ₁ = ϕ(%i₀, %i₂)
%t₃ = %ᵢ₁ - %t₂
blz %t₃, body, exit

%ᵢ₂ = %ᵢ₁ + 1
b loop

return %ᵢ₁
• An *induction variable* is a variable \( x \) such that the difference between successive values of \( x \) in a loop is constant.
  • Common example: the loop counter in a for loop
    ```java
    for (int i = 0; i < n; i++)
    ```
Induction variables

- An *induction variable* is a variable $x$ such that the difference between successive values of $x$ in a loop is constant.
  - Common example: the loop counter in a `for` loop
    ```java
    for (int i = 0; i < n; i++)
    ```
- Useful for several optimizations
  - Strength reduction, loop unrolling, induction variable elimination, parallelization, array bound-check elision
Induction variables, formally

- Use $\%x(k)$ to denote the value of $\%x$ in the $k$th iteration of a loop. $\%x$ is an induction variable if there is some constant (loop-invariant) $\Delta(\%x)$ such that

$$\%x(k + 1) = \%x(k) + \Delta(\%x)$$

for all $k$
Induction variables, formally

- Use $%x(k)$ to denote the value of $%x$ in the $k$th iteration of a loop. $%x$ is an induction variable if there is some constant (loop-invariant) $\Delta(%x)$ such that

$$%x(k+1) = %x(k) + \Delta(%x)$$

for all $k$

- A variable $%x$ is an *basic induction variable* for a loop $L$ if it is increased / decreased by a fixed loop-invariant quantity in any iteration of the loop.
  - $%x(i + 1) = %x(i) + c \Rightarrow \Delta(%x) = c$
Induction variables, formally

- Use $\%x(k)$ to denote the value of $\%x$ in the $k$th iteration of a loop. $\%x$ is an induction variable if there is some constant (loop-invariant) $\Delta(\%x)$ such that

$$\%x(k + 1) = \%x(k) + \Delta(\%x)$$

for all $k$

- A variable $\%x$ is an **basic induction variable** for a loop $L$ if it is increased / decreased by a fixed loop-invariant quantity in any iteration of the loop.
  - $\%x(i + 1) = \%x(i) + c \Rightarrow \Delta(\%x) = c$

- A variable $\%y$ is an **derived induction variable** for a loop $L$ if it is an affine function of a basic induction variable
  - $\%y(i) = a \cdot \%x(i) + b \Rightarrow \Delta(\%y) = a \cdot c$
Finding induction variables

- Basic induction variable detection:
  - 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.
Finding induction variables

- Basic induction variable detection:
  - 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 with cheaper one (e.g., replace multiplication w/ 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)
%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
%result_2 = %result_1 + t_5
%i_2 = %i_1 + 1
b \text{ loop}
%i_1 = ϕ(%i_0, %i_2)
%result_1 = ϕ(%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
\( i_1 = \phi(i_0, i_2) \)
\( \text{result}_1 = \phi(\text{result}_0, \text{result}_2) \)
\( t_1 = i_1 - n \)
\( \text{blz } t_1, \text{body, exit} \)

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

\[%t_2 = \%i_1 * %n\]
\[\%t_3 = %m + %t_2\]
\[\%t_4 = %t_3 + %i_1\]
\[\%t_5 = \text{load} \ %t_4\]
\[%\text{result}_2 = %\text{result}_1 + %t_5\]
\[%i_2 = %i_1 + 1\]
\text{b} \text{ loop}
\[ i_1 = \phi(i_0, i_2) \]  \[ i := i + 1 \]

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

\[ t_1 = i_1 - n \]  \[ t_1 := i + n \]

```
blz %t1, body, exit
```

\[ t_2 = i_1 \times n \]  \[ t_2 := n \times i \]

\[ t_3 = m + t_2 \]
\[ t_4 = t_3 + i_1 \]
\[ t_5 = \text{load} \ t_4 \]
\[ \text{result}_2 = \text{result}_1 + t_5 \]
\[ i_2 = i_1 + 1 \]

b loop
\( \%i_1 = \phi(\%i_0, \%i_2) \)

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

\( \%t1 = \%i_1 - \%n \)

\( \text{blz } \%t1, \text{ body, exit} \)

\( \%t2 = \%i_1 \times \%n \)

\( \%t3 = \%m + \%t2 \)

\( \%t4 = \%t3 + \%i_1 \)

\( \%t5 = \text{load } \%t4 \)

\( \% \text{result}_2 = \% \text{result}_1 + \%t5 \)

\( \%i_2 = \%i_1 + 1 \)

\( b \text{ loop} \)
\[ %i_1 = \phi(\%i_0, \%i_2) \]
\[ \text{result}_1 = \phi(\text{result}_0, \text{result}_2) \]
\[ %t_1 = %i_1 - \%n \]
\[ \text{t1} := i + n \]

\[ \text{blz} \ %t_1, \ \text{body, exit} \]

\[ %t_2 = %i_1 \times \%n \]
\[ %t_3 = %m + %t_2 \]
\[ %t_4 = %t_3 + %i_1 \]
\[ %t_5 = \text{load} \ %t_4 \]
\[ \text{result}_2 = \text{result}_1 + %t_5 \]
\[ %i_2 = %i_1 + 1 \]
\[ \text{b loop} \]
%t2_0 = 0
%t3_0 = %m
%t4_0 = %m

%i_1 = \phi(%i_0, %i_2) i := i + 1
%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 t1 := i + n
blz %t1, body, exit

%t2_2 = %t2_1 + %n t2 := n*i
%t3_2 = %t3_1 + %n t3 := n*i + m
%t6 = %t4_1 + %n
%t4_2 = %t6 + 1 t4 := (n+1)*i + m
%t5 = load %t4_2
%result_2 = %result_1 + %t5
%i_2 = %i_1 + 1
b loop
Loop unrolling

• Can expose opportunities for using Single Instruction Multiple Data (SIMD) instructions
• 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
• Loop unrolling trades (potential) run-time performance with code size.
bgz t + 3
\[ \Delta(t), \text{in, out} \]

Conditional branch \( \Rightarrow \) unconditional branch

Redirect back-edges to next loop copy

Insert epilogue, in case # iterations is not divisible by 4

Copy loop

Single exit:

\[ \text{bgz t, in, out} \]

t an ind. var w/ \( \Delta(t) = c \leq 0 \)
Single exit: `bgz t, in, out`

t an ind. var w/ $\Delta(t) = c \leq 0$
Conditional branch \( \Rightarrow \) unconditional branch

Redirect back-edges to next loop copy

Insert epilogue, in case # iterations is not divisible by 4

Copy loop

\[ h \]

\[ t \]

\[ \Delta(t) \]

\[ \leq \]
Conditional branch $\leadsto$ unconditional branch

bgz $t + 3\Delta(t)$, in, out
Redirect back-edges to next loop copy
Insert epilogue, in case # iterations is not divisible by 4
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 safe
- Each transformation is simple
- Transformations are mutually beneficial
  - Series of transformations can make drastic changes!