COS320: Compiling Techniques

Zak Kincaid

April 26, 2019

Loop transformations

Loops

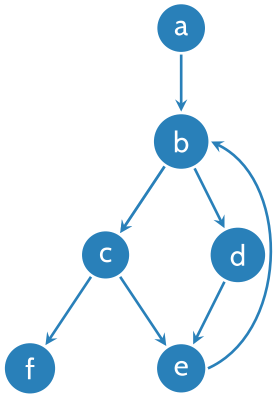
- Almost all execution time is inside loops
- Many optimizations are centered around transforming loops
    - Loop invariant code motion: avoid re-computing expressions by hoisting them out of the loop
    - Loop unrolling: avoid branching by excecuting several iterations of a loop at a time
    - Strength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

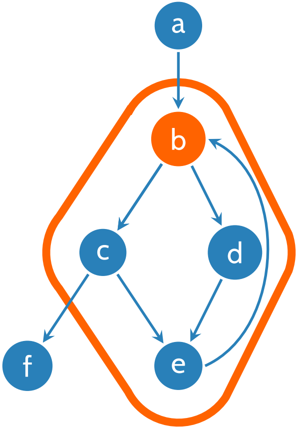
What is a loop?

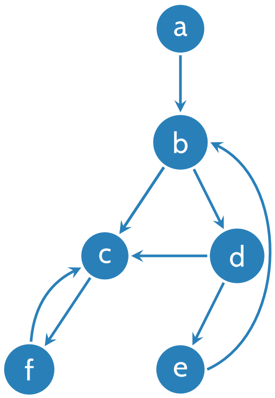
- We're after a *graph-theoretic* definition of a loop
    - Not sensitive to syntax of source language (loops can be created with `while`, `for`, `goto`, …)
- First attempt: SCCs
    - Not fine enough – nested loops have only one SCC, but we want to transform them separately
    - Too general – makes 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`)

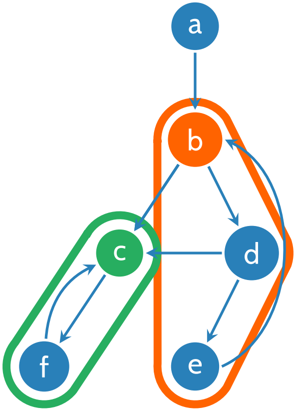
What is a loop?

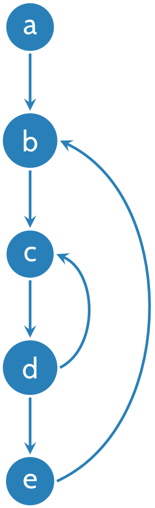
- A loop of a control flow graph is a set of nodes S such that
  1. S is strongly connected
  2. There is a *header* node h that dominates all nodes in S
  3. There is no edge from any node *outside* of S to any node *inside* of S, except for h
- A *loop entry* is a node with some predecessor outside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

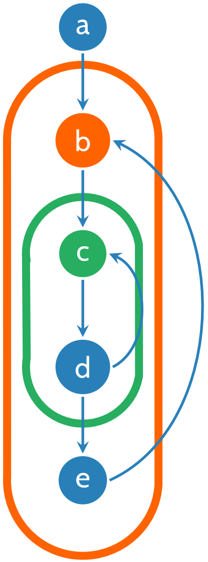

a
b
c
d
f
e

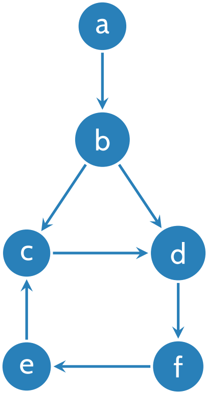

a
b
c
d
e
f

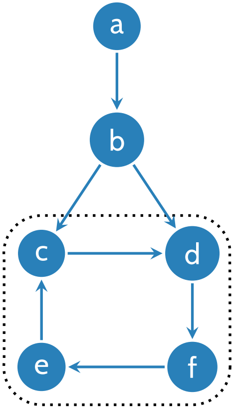

a
b
c
d
f
e


a
b
c
d
f
e


a
b
c
d
e

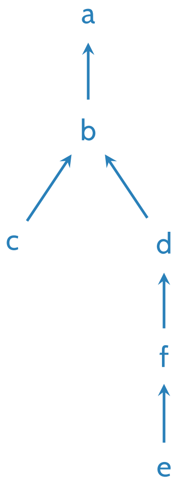

a
b
c
d
e


a
b
c
d
e
f



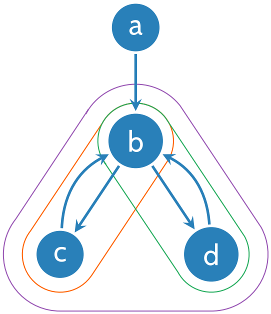
Strongly connected subgraph

Dominator tree



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 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.

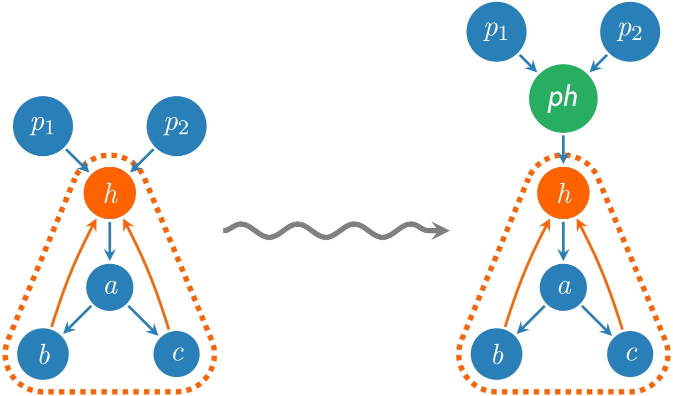

a
b
c
d

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

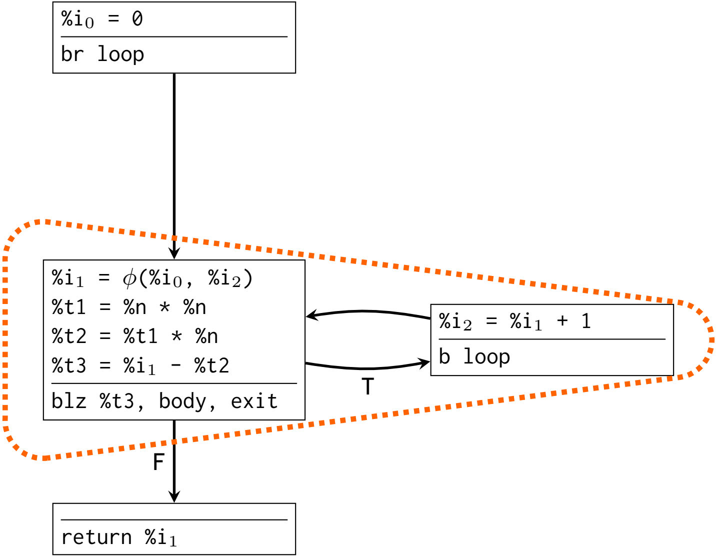
Loop preheaders

- 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

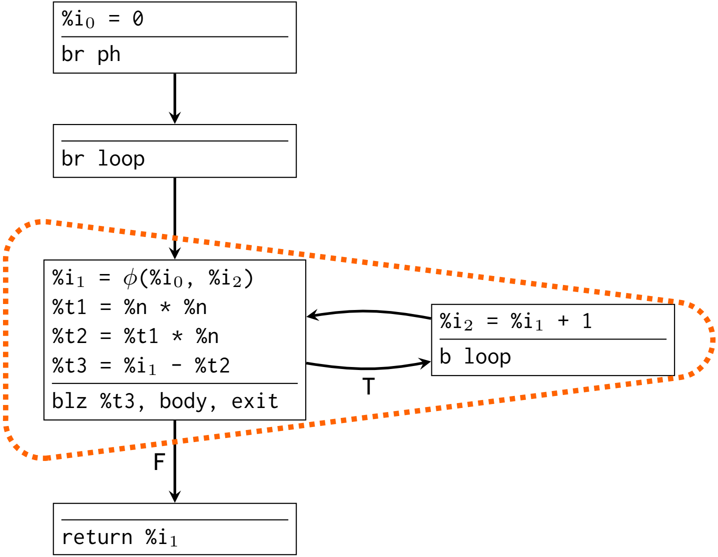


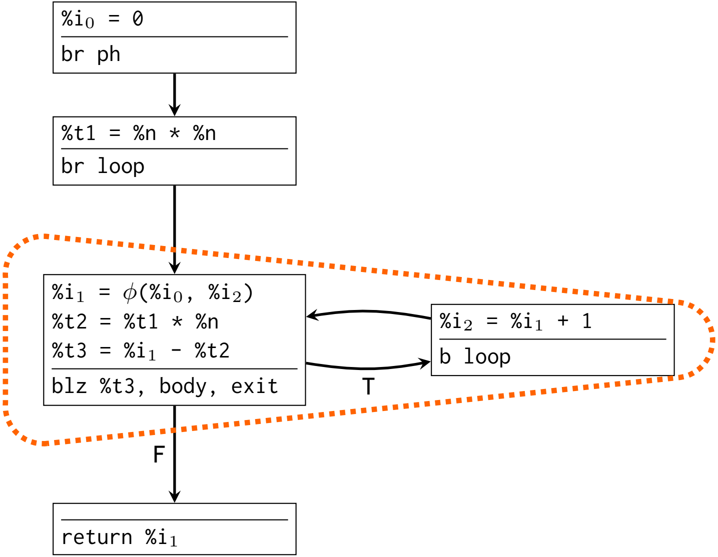
Loop invariant code motion

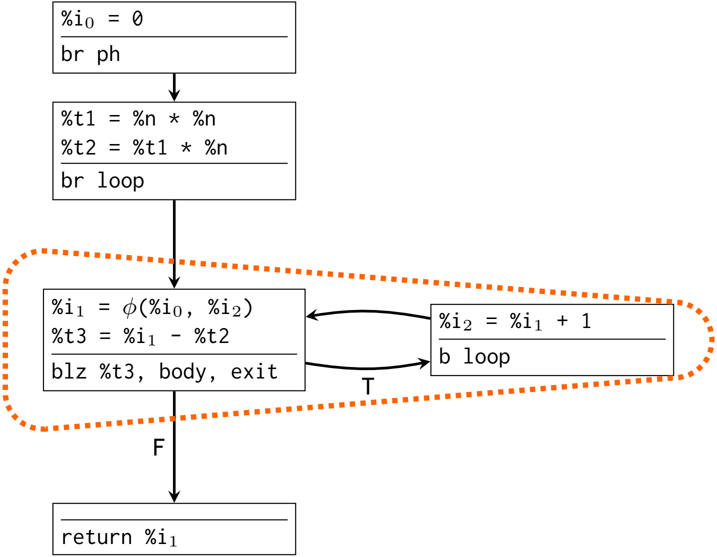
- Loop invariant code motion saves the cost of re-computing expressions that are left invariant (i.e., do not change) in the loop.
  - Such computations can be moved the loop's preheader, as long as they are not side-effecting
- SSA based LICM:
  - An operand is *invariant* in a loop L if
    1. It is 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$_1$ = ϕ(%i$_0$, %i$_2$)
%t1 = %n * %n
%t2 = %t1 * %n
%t3 = %i$_1$ - %t2
blz %t3, body, exit
%i$_2$ = %i$_1$ + 1
b loop
T
F
return %i$_1$


%i0 = 0
br ph
br loop
%i1 = φ(%i0, %i2)
%t1 = %n * %n
%t2 = %t1 * %n
%t3 = %i1 - %t2
blz %t3, body, exit
%i2 = %i1 + 1
b loop
T
F
return %i1


%i0 = 0
br ph
%t1 = %n * %n
br loop
%i1 = φ(%i0, %i2)
%t2 = %t1 * %n
%t3 = %i1 - %t2
blz %t3, body, exit
%i2 = %i1 + 1
b loop
T
F
return %i1


%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
T
F
return %i1

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 ϕ statements $\%x = \phi(\%x_1, ..., \%x_n)$ 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 ϕ statements $\%x = \phi(\%x_1, ..., \%x_n)$ 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.
- 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 $*$ 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).

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

→

```
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;
}
```

```
%i₁ = φ(%i₀, %i₂)
%result₁ = φ(%result₀, %result₂)
%t1 = %i₁ - %n
blz %t1, body, exit
```

```
%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

```
%i₁ = φ(%i₀, %i₂)
%result₁ = φ(%result₀, %result₂)
%t1 = %i₁ - %n
blz %t1, body, exit
```

```
%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

```
%i₁ = φ(%i₀, %i₂)
%result₁ = φ(%result₀, %result₂)
%t1 = %i₁ - %n
blz %t1, body, exit


%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

i := i + 1

```
%i₁ = φ(%i₀, %i₂)
%result₁ = φ(%result₀, %result₂)
 %t1 = %i₁ - %n
blz %t1, body, exit
```

```
%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

i := i + 1

```
%i₁ = φ(%i₀, %i₂)
%result₁ = φ(%result₀, %result₂)
%t1 = %i₁ - %n
blz %t1, body, exit
```

i := i + 1

t1 := i + n

```
%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

t2 := n*i

$\%i_1 = \phi(\%i_0, \%i_2)$ i := i + 1

$\%result_1 = \phi(\%result_0, \%result_2)$

```
%t1 = %i₁ - %n
blz %t1, body, exit
```

t1 := i + n

```
%t2 = %i₁ * %n
%t3 = %m + %t2
%t4 = %t3 + %i₁
%t5 = load %t4
%result₂ = %result₁ + %t5
%i₂ = %i₁ + 1
b loop
```

t2 := n*i

t3 := n*i + m

$\%i_1 = \phi(\%i_0, \%i_2)$ i := i + 1

$\%result_1 = \phi(\%result_0, \%result_2)$

$\%t1 = \%i_1 - \%n$ t1 := i + n

blz %t1, body, exit

$\%t2 = \%i_1 * \%n$ t2 := n*i

%t3 = %m + %t2 t3 := n*i + m

$\%t4 = \%t3 + \%i_1$ t4 := (n+1)*i + m

%t5 = load %t4

$\%result_2 = \%result_1 + \%t5$

$\%i_2 = \%i_1 + 1$

b loop

```
%t2_0 = 0
%t3_0 = %m
%t4_0 = %m


%i_1 = φ(%i_0, %i_2)                                  i := i + 1
%t2_1 = φ(%t2_0, %t2_2)
%t3_1 = φ(%t3_0, %t3_2)
%t4_1 = φ(%t4_0, %t4_2)
%result_1 = φ(%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_2 + %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

- 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 `blz t, body, exit`, where `t` is a derived induction variable `t = a·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·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 `blz t, body, exit`, where `t` is a derived induction variable `t = a·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 `blz t, body, exit` w/ `blz (t + a · c), body, exit`
- In h', replace `blz t, body, exit` w/ `b body`

- Given a loop L with header h *Suppose* loop exit is `blz t, body, exit`, where `t` is a derived induction variable `t = a·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 `blz t, body, exit` w/ `blz (t + a · c), body, exit`
- In h', replace `blz t, body, exit` w/ `b body`
- Add loop epilogue to execute last iteration, if needed

```
%t2_0 = 0
%t3_0 = %m
%t4_0 = %m
```

```
%i_1 = φ(%i_0, %i_2')
%t2_1 = φ(%t2_0, %t2_2')
%t3_1 = φ(%t3_0, %t3_2')
%t4_1 = φ(%t4_0, %t4_2')
%result_1 = φ(%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_1 + 1
b loop'
```

```
b body'
```

```
%t2_2' = %t2_2 + %n
%t3_2' = %t3_2 + %n
%t6' = %t4_2 + %n
%t4_2' = %t6' + 1
%t5' = load %t4_2'
%result_2' = %result_2 + %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!