COS320: Compiling Techniques

Zak Kincaid

April 21, 2020

Logistics

- Reminder: HW4 is due **today**
  - Typo in `hw4programs/field_overlap.oat`:
    `int program (int argc, string[] argv)`
- HW5 released today. 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
    - Loop unrolling: avoid branching by excecuting several iterations of a loop
    - Strength reduction: replace a costly operation inside a loop with a cheaper one
    - 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`, ...)

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

What is a loop?

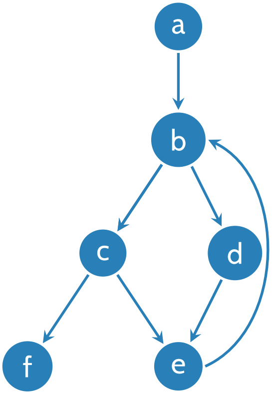
- We're after a *graph-theoretic* definition of a loop
  - Not sensitive to syntax of source language (loops can be created with `while`, `for`, `goto`, ...)
- First attempt: SCCs
  - Not fine enough – nested loops have only one SCC, but we want to transform them separately
  - Too general – makes it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

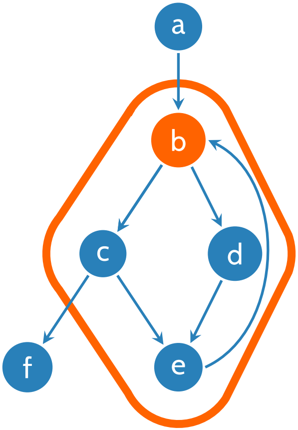
What is a loop?

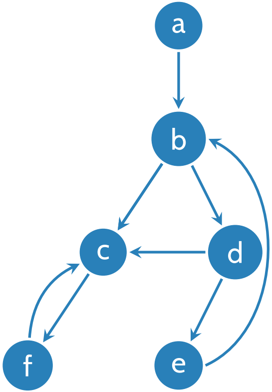
- A loop of a control flow graph is a set of nodes S such that
  1. S is strongly connected
  2. There is a *header* node h that dominates all nodes in S
  3. There is no edge from any node *outside* of S to any node *inside* of S, except for h

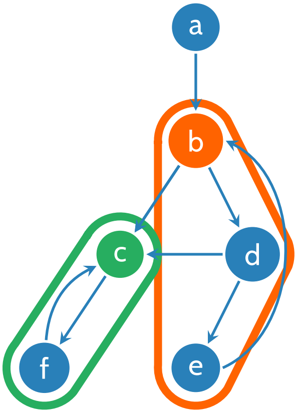
What is a loop?

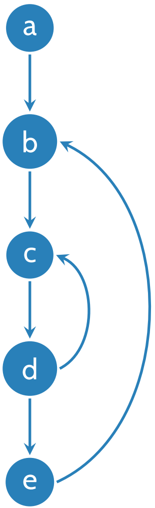
- A loop of a control flow graph is a set of nodes S such that
  1. S is strongly connected
  2. There is a *header* node h that dominates all nodes in S
  3. There is no edge from any node *outside* of S to any node *inside* of S, except for h
- Observe: a loop has one entry, but may have multiple exits (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

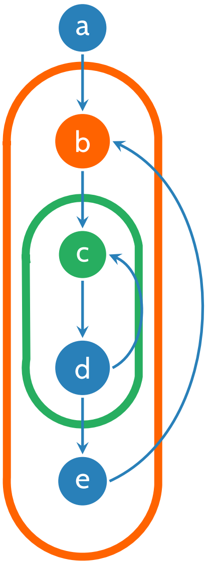

a
b
c
d
f
e


a
b
c
d
e
f

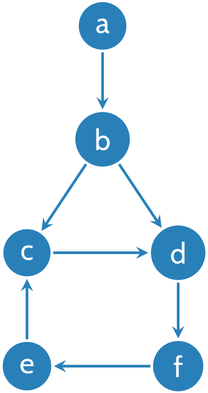

a
b
c
d
e
f

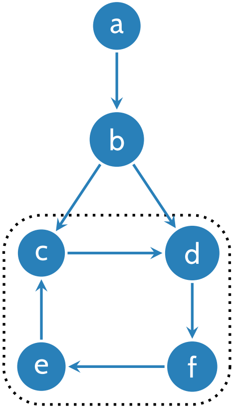

a
b
c
d
e
f


a
b
c
d
e



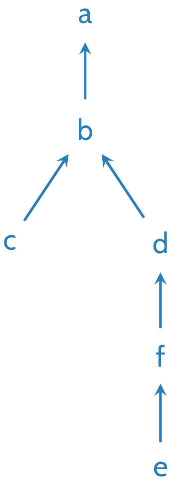
a
b
c
d
e


a
b
c
d
e
f



Strongly connected subgraph

Dominator tree



Identifying loops

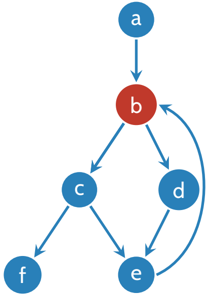
- A **back edge** is an edge $u \rightarrow v$ such that v dominates u

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.

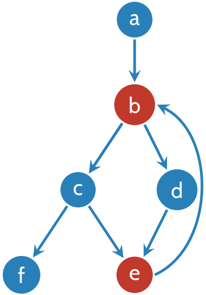
Identifying loops

- 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



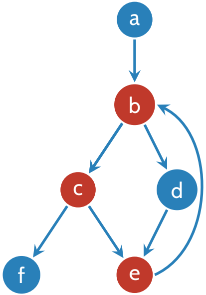
Identifying loops

- 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



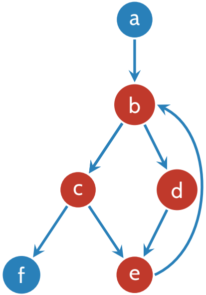
Identifying loops

- 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



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:

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. Header (v) dominates 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. Header (v) dominates the loop
3. Single entry

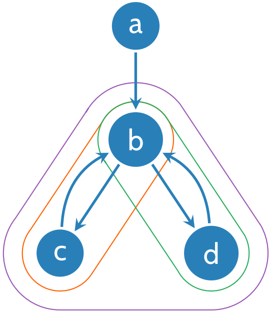
Every natural loop is a loop:

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. Header (v) dominates the loop
3. Single entry
   - By DFS construction, all predecessors of any node except v belongs 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. Header (v) dominates the loop
3. Single entry
   - By DFS construction, all predecessors of any node except v belongs 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
  - Commonly, we merge natural loops with the same header

Nested loops

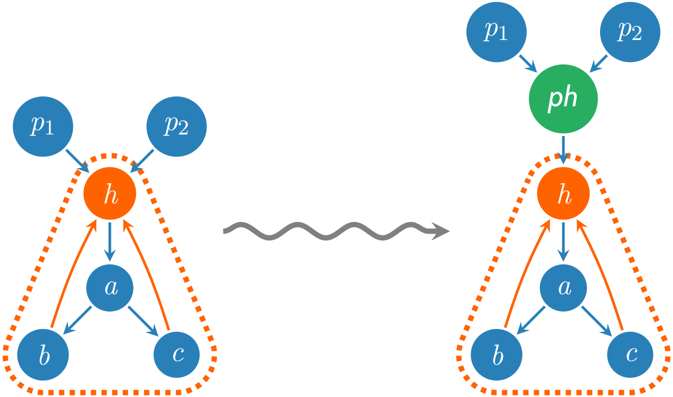
- Say that a loop B is *nested* within A if $B \subseteq A$
- A node can be the header of more than one natural loop.
  - Neither is nested inside the other
  - Commonly, we merge natural loops with the same header
- Loops obtained by merging natural loops with the same header are either disjoint or nested
  - 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 merge 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 the loop's preheader, as long as they are not side-effecting

Loop invariant code motion

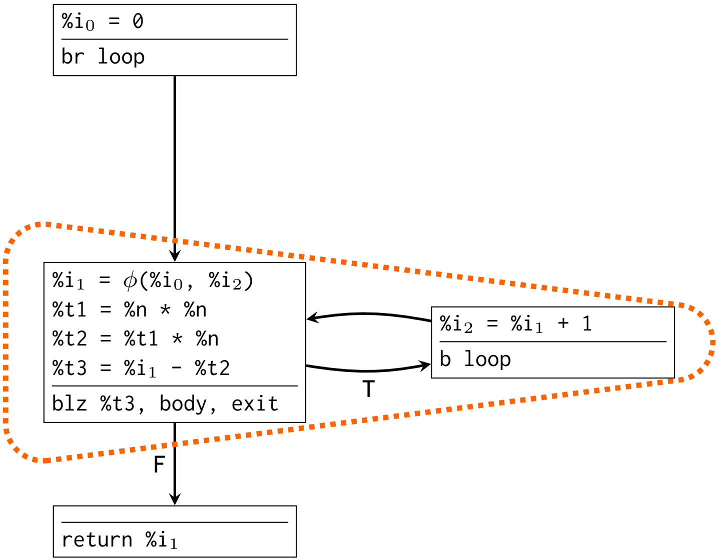
- Loop invariant code motion saves the cost of re-computing expressions that are left invariant (i.e., do not change) in the loop.
  - Such computations can be moved the loop's preheader, as long as they are not side-effecting
- SSA based LICM:
  - An operand is *invariant* in a loop L if
    1. It is a constant, or
    2. It is a gid, or
    3. It is a uid, whose definition does not belong to L

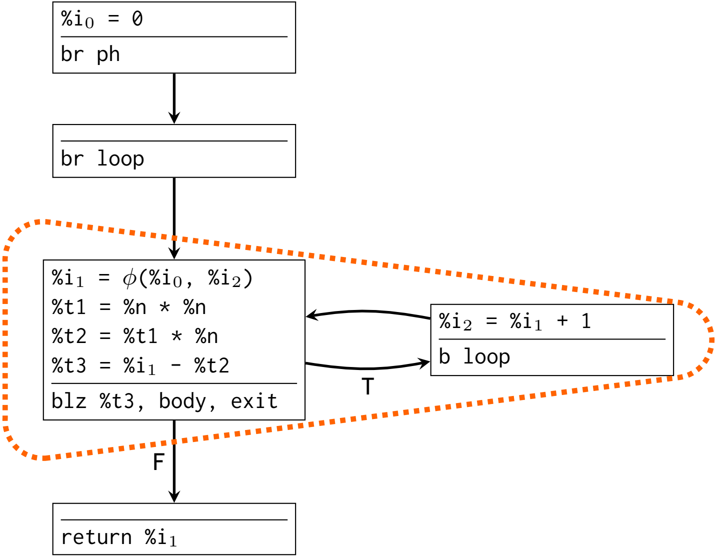
Loop invariant code motion

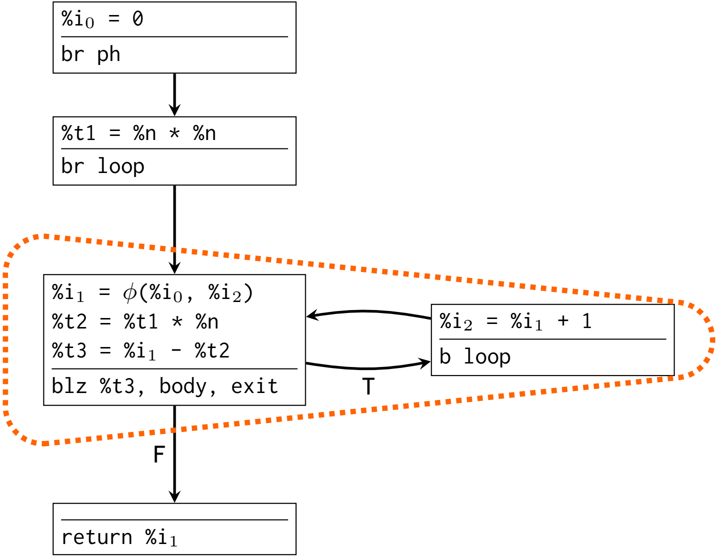
- Loop invariant code motion saves the cost of re-computing expressions that are left invariant (i.e., do not change) in the loop.
  - Such computations can be moved the loop's preheader, as long as they are not side-effecting
- SSA based LICM:
  - An operand is *invariant* in a loop L if
    1. It is 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

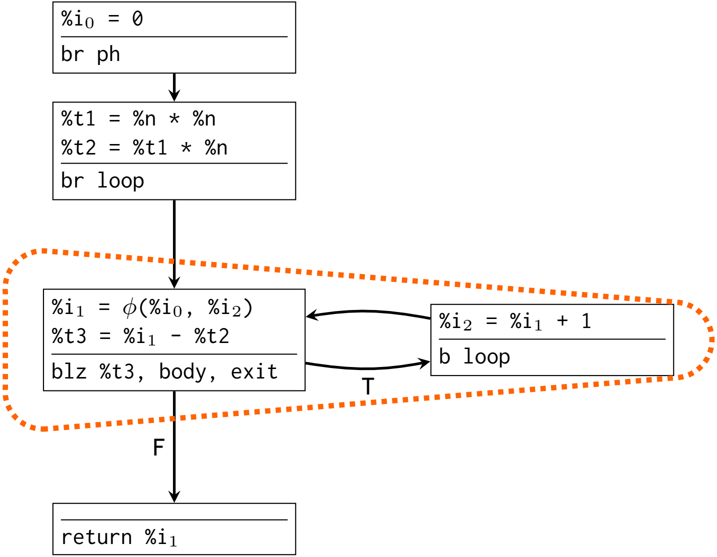
- Loop invariant code motion saves the cost of re-computing expressions that are left invariant (i.e., do not change) in the loop.
  - Such computations can be moved the loop's preheader, as long as they are not side-effecting
- SSA based LICM:
  - An operand is *invariant* in a loop L if
    1. It is 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


%i0 = 0
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
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

- 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
    ```
    for (int i = 0; i < n; i++)
    ```
  - Using $\%x(k)$ to denote the value of $\%x$ in the kth iteration of a loop, there is some constant $\Delta(\%x)$ such that

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

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
    ```
    for (int i = 0; i < n; i++)
    ```
  - Using $\%x(k)$ to denote the value of $\%x$ in the kth iteration of a loop, there is some constant $\Delta(\%x)$ such that
    $$\%x(k+1) = \%x(k) + \Delta(\%x)$$
- Useful for several optimizations
  - Strength reduction, loop unrolling, induction variable elimination, parallelization, array bound-check elision

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
    `for (int i = 0; i < n; i++)`
  - Using $\%x(k)$ to denote the value of $\%x$ in the kth iteration of a loop, there is some constant $\Delta(\%x)$ such that
$$\%x(k+1) = \%x(k) + \Delta(\%x)$$
- Useful for several optimizations
  - Strength reduction, loop unrolling, induction variable elimination, parallelization, array bound-check elision
- 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

- 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
    `for (int i = 0; i < n; i++)`
  - Using $\%x(k)$ to denote the value of $\%x$ in the kth iteration of a loop, there is some constant $\Delta(\%x)$ such that
    $$\%x(k+1) = \%x(k) + \Delta(\%x)$$
- Useful for several optimizations
  - Strength reduction, loop unrolling, induction variable elimination, parallelization, array bound-check elision
- 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 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.

Finding induction variables

- Basic induction variable detection:
  - 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 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;
}
```

```
%i₁ = φ(%i₀, %i₂)
```

$\%\mathtt{i}_1 = \phi(\%\mathtt{i}_0,\ \%\mathtt{i}_2)$

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

%t1 = $\%\mathtt{i}_1$ - %n

blz %t1, body, exit

%t2 = $\%\mathtt{i}_1$ * %n

%t3 = %m + %t2

%t4 = %t3 + $\%\mathtt{i}_1$

%t5 = load %t4

$\%\mathtt{result}_2$ = $\%\mathtt{result}_1$ + %t5

$\%\mathtt{i}_2$ = $\%\mathtt{i}_1$ + 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
```

i := i + 1

```
%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
```

i := i + 1

```
%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
```

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$_1$ = ϕ(%i$_0$, %i$_2$)	i := i + 1
%result$_1$ = ϕ(%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$_1$	
%t5 = load %t4	
%result$_2$ = %result$_1$ + %t5	
%i$_2$ = %i$_1$ + 1	
b loop	

```
%i_1 = φ(%i_0, %i_2)                              i := i + 1
%result_1 = φ(%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
```

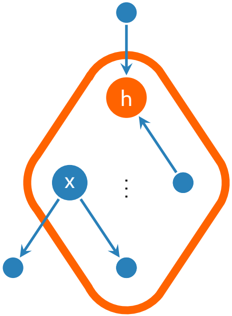
%t2$_0$ = 0
%t3$_0$ = %m
%t4$_0$ = %m

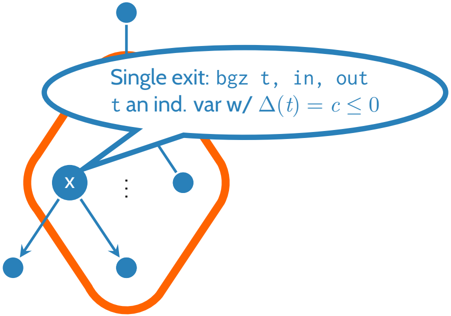
%i$_1$ = ϕ(%i$_0$, %i$_2$) i := i + 1
%t2$_1$ = ϕ(%t2$_0$, %t2$_2$)
%t3$_1$ = ϕ(%t3$_0$, %t3$_2$)
%t4$_1$ = ϕ(%t4$_0$, %t4$_2$)
%result$_1$ = ϕ(%result$_0$, %result$_2$)
%t1 = %i$_1$ - %n t1 := i + n
blz %t1, body, exit

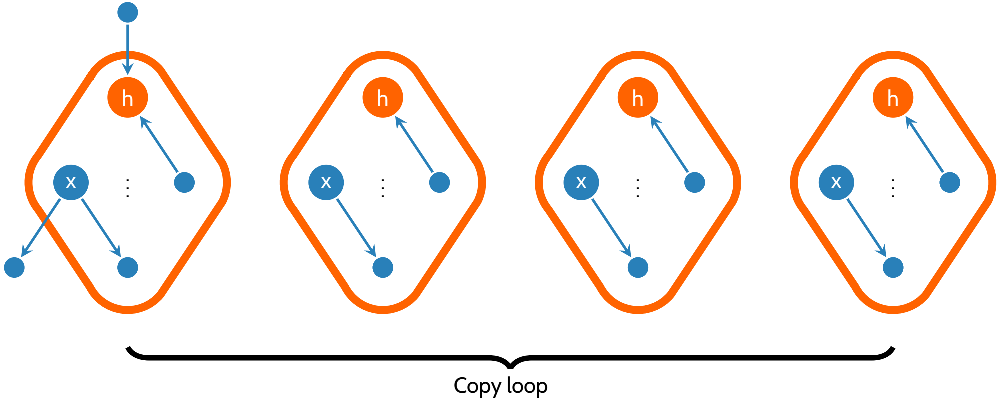
%t2$_2$ = %t2$_1$ + %n t2 := n*i
%t3$_2$ = %t3$_1$ + %n t3 := n*i + m
%t6 = %t4$_2$ + %n
%t4$_2$ = %t6 + 1 t4 := (n+1)*i + m
%t5 = load %t4$_2$
%result$_2$ = %result$_1$ + %t5
%i$_2$ = %i$_1$ + 1
b loop

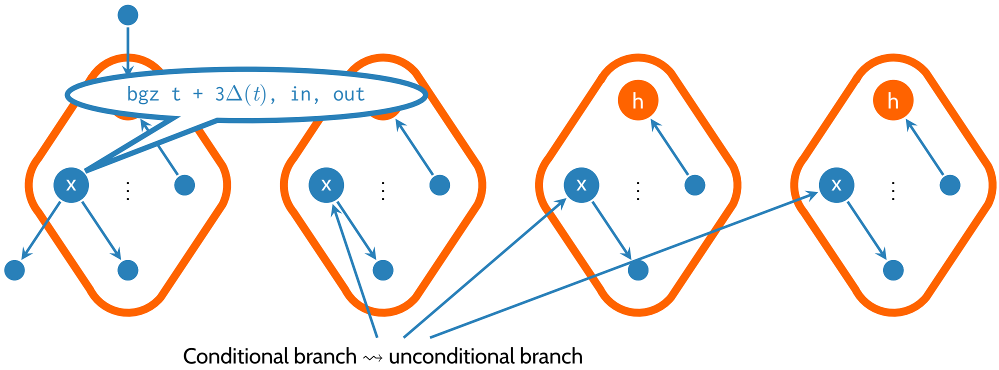
Loop unrolling

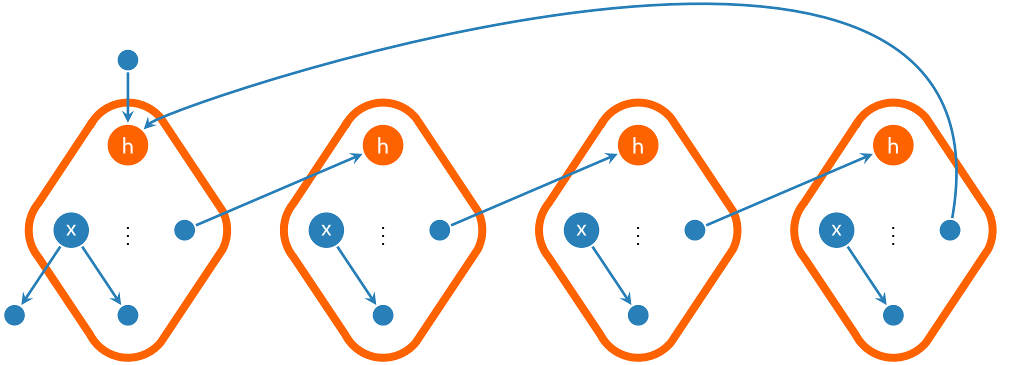
- Some loops are so small that a significant portion of the running time is due to testing the loop exit condition
- We can avoid branching by executing several iterations of the loop at once
- This optimization trades (potential) run-time performance with code size.


h
x
⋮

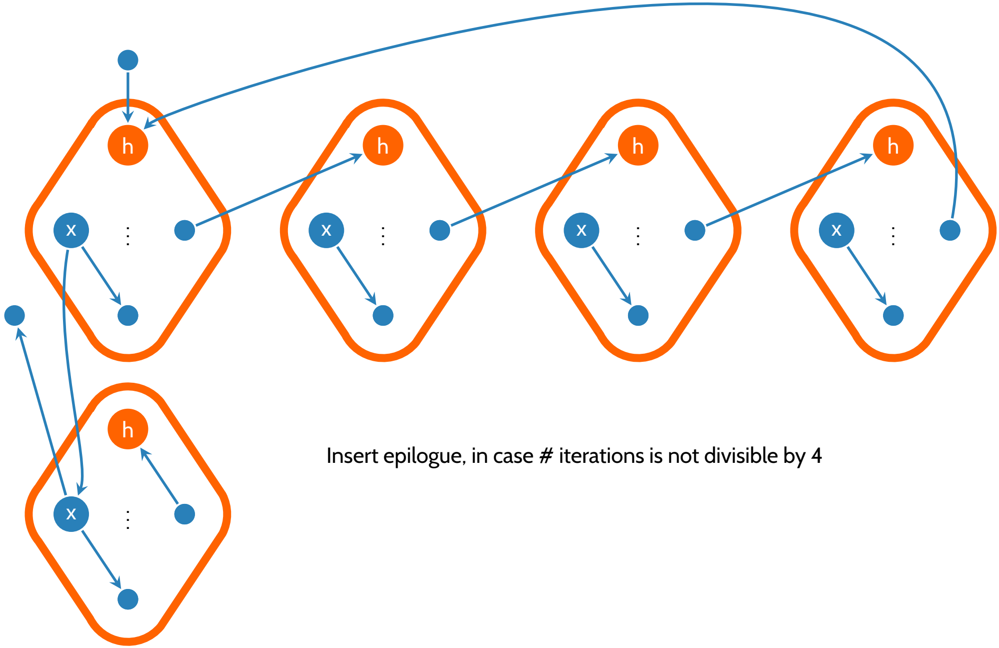

Single exit: bgz t, in, out
t an ind. var w/ $\Delta(t) = c \leq 0$
x
⋮


h
x
h
x
h
x
h
x
Copy loop


bgz t + 3Δ(t), in, out
x
h
Conditional branch ⇝ unconditional branch



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!