HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 23:
Loop Optimization

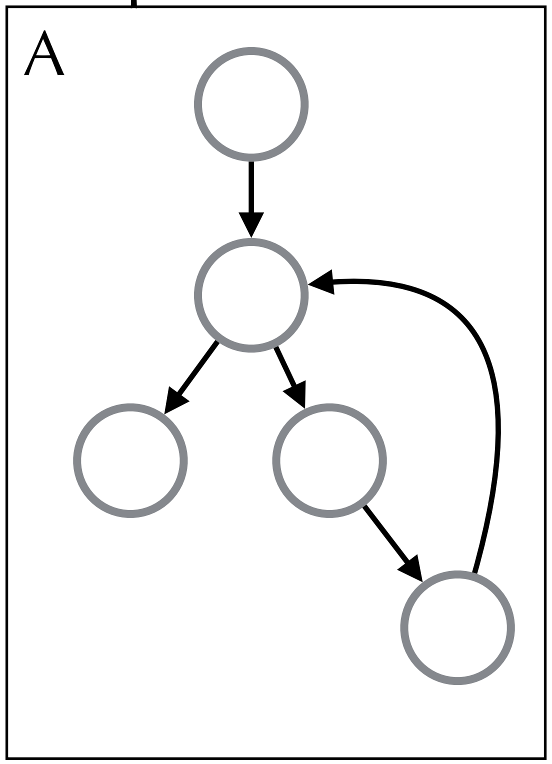
Stephen Chong

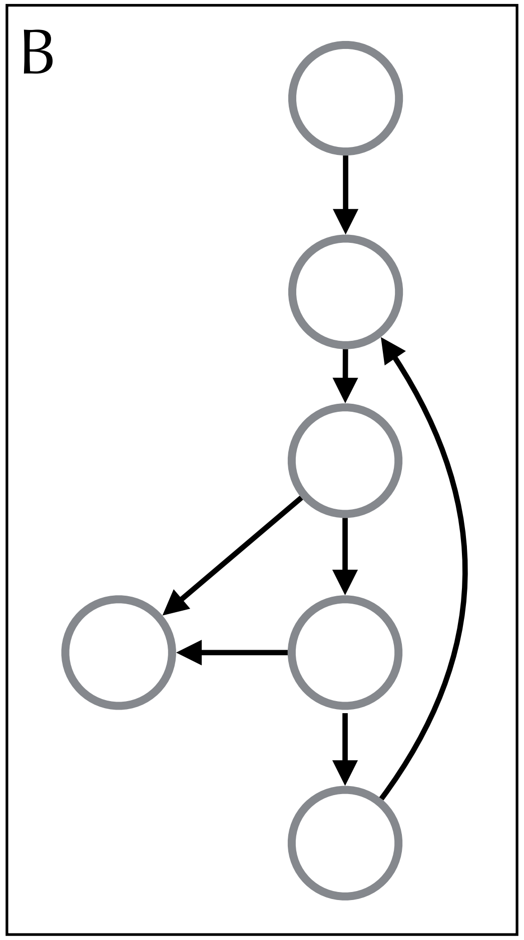
https://www.seas.harvard.edu/courses/cs153

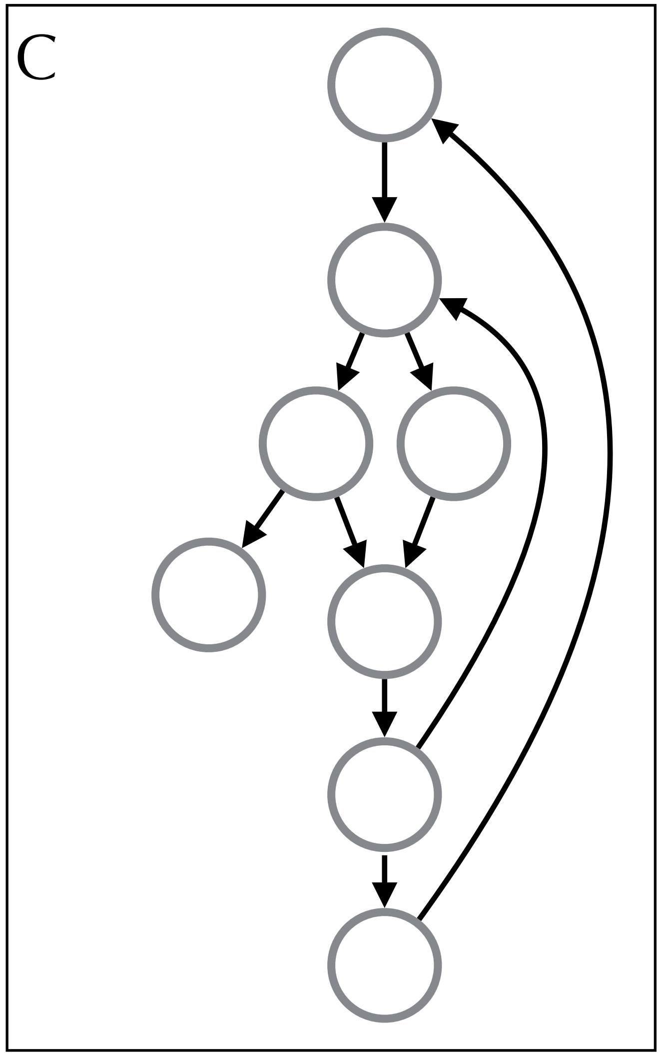
Contains content from lecture notes by Greg Morrisett

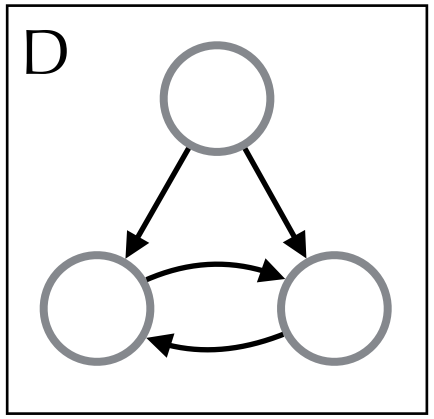
Pre-class Puzzle

- For each of these Control Flow Graphs (CFGs), what is a C program that corresponds to it?









Announcements

- HW5: Oat v.2 due today (Tue Nov 19)
- HW6: Optimization and Data Analysis
  - Due: Tue Dec 3 (in 2 weeks)
- Final exam
  - 9am-12pm Thursday December 19
  - Extension school: online exam, 24 hour window
  - Open book, open note, open laptop
    - No communication, no searching for answers on internet
  - ~30 multiple choice or short answer questions
  - Comprehensive exam (i.e., all material covered in course)
    - Won't need to program, won't depend on
  - We will release some study material in a few weeks

Announcements: Upcoming Lectures

- Thursday Nov 21: Embedded EthiCS module
  - Ethics of Open Source
  - Guest lecturer Meica Magnani
  - Pre-lecture viewing/thinking posted on Piazza
  - Will be a brief assignment posted on Piazza after lecture
- Tuesday Dec 3: The Economics of Programming Languages
  - Evan Czaplicki '12, creator of the Elm programming language
    - https://elm-lang.org/

Today

- Loop optimization
  - Examples
  - Identifying loops
    - Dominators
  - Loop-invariant removal
  - Induction variable reduction
  - Loop fusion
  - Loop fission
  - Loop unrolling
  - Loop interchange
  - Loop peeling
  - Loop tiling
  - Loop parallelization

Loop Optimizations

- Vast majority of time spent in loops
- So we want techniques to improve loops!
  - Loop invariant removal
  - Induction variable elimination
  - Loop unrolling
  - Loop fusion
  - Loop fission
  - Loop peeling
  - Loop interchange
  - Loop tiling
  - Loop parallelization
  - Software pipelining

Example 1: Invariant Removal

```
L0:  t := 0

L1:  i := i + 1
     t := a + b
     *i := t
     if i<N goto L1 else L2

L2:  x := t
```

Example 1: Invariant Removal

```
L0:  t := 0
     t := a + b

L1:  i := i + 1

     *i := t
     if i<N goto L1 else L2

L2:  x := t
```

Example 2: Induction Variable

```
s=0;
for (i=0; i < 100; i++)
   s += a[i];
```

```
L0:   i  := 0
      s  := 0
      jump L2
L1:   t1  := i*4
      t2  := a+t1
      t3  := *t2
      s   := s + t3
      i   := i+1
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      jump L2
L1:   t1 := i*4
      t2 := a+t1
      t3 := *t2
      s  := s + t3
      i  := i+1
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

t1 is always equal to i*4 !

Example 2: Induction Variable

```
L0:   i  := 0
      s  := 0
      t1  := 0
      jump L2
L1:   t2  := a+t1
      t3  := *t2
      s   := s + t3
      i   := i+1
      t1  := t1+4
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

`t1` is always equal to `i*4` !

Example 2: Induction Variable

```
L0:  i  := 0
     s  := 0
     t1 := 0
     jump L2
L1:  t2 := a+t1
     t3 := *t2
     s  := s + t3
     i  := i+1
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

Example 2: Induction Variable

```
L0:  i  := 0
     s  := 0
     t1 := 0
     jump L2
L1:  t2 := a+t1
     t3 := *t2
     s  := s + t3
     i  := i+1
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

`t2` is always equal to `a+t1` == `a+i*4` !

Example 2: Induction Variable

```
L0:   i  := 0
      s  := 0
      t1  := 0
      t2  := a
      jump L2
L1:   t3  := *t2
      s   := s + t3
      i   := i+1
      t2  := t2+4
      t1  := t1+4
L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

t2 is always equal to a+t1 == a+i*4 !

Example 2: Induction Variable

```
L0:  i  := 0
     s  := 0
     t1 := 0
     t2 := a
     jump L2
L1:  t3 := *t2
     s  := s + t3
     i  := i+1
     t2 := t2+4
     t1 := t1+4
L2:  if i < 100 goto L1 else goto L3
L3:  ...
```

`t1` is no longer used!

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      jump L2

L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      jump L2

L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if i < 100 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations.
But `t2` = `4*i + a`
so `i < 100`
when
`t2 < a+400`

Example 2: Induction Variable

```
L0:   i := 0
      s := 0
      t2 := a
      t5 := t2 + 400
      jump L2
L1:   t3 := *t2
      s  := s + t3
      i  := i+1
      t2 := t2+4

L2:   if t2 < t5 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations. But `t2` = `4*i + a` so `i < 100` when `t2 < a+400`

Example 2: Induction Variable

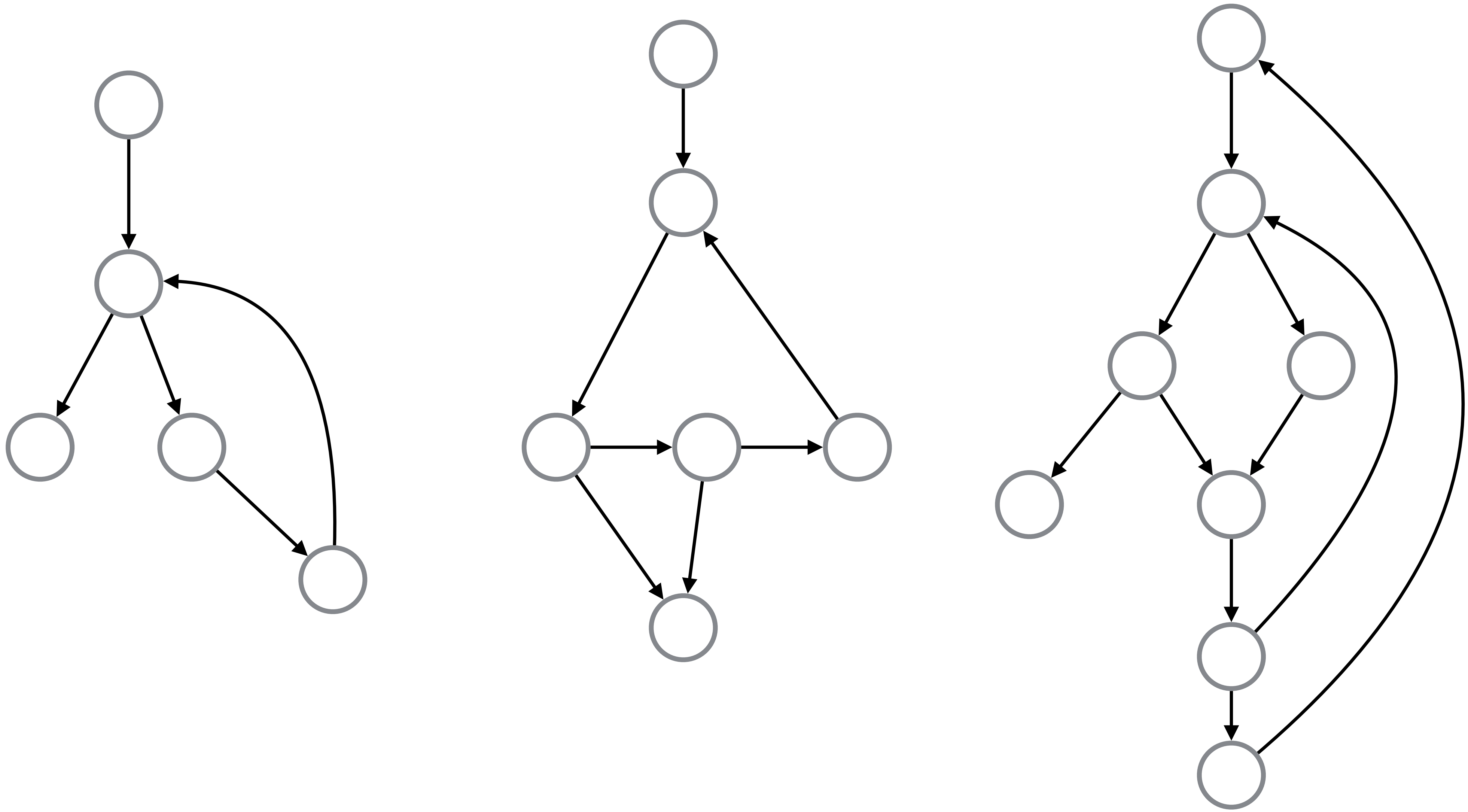
```
L0:   s := 0
      t2 := a
      t5 := t2 + 400
      jump L2

L1:   t3 := *t2
      s  := s + t3
      t2 := t2+4

L2:   if t2 < t5 goto L1 else goto L3
L3:   ...
```

`i` is now used just to count 100 iterations. But `t2` = `4*i + a` so `i < 100` when `t2 < a+400`

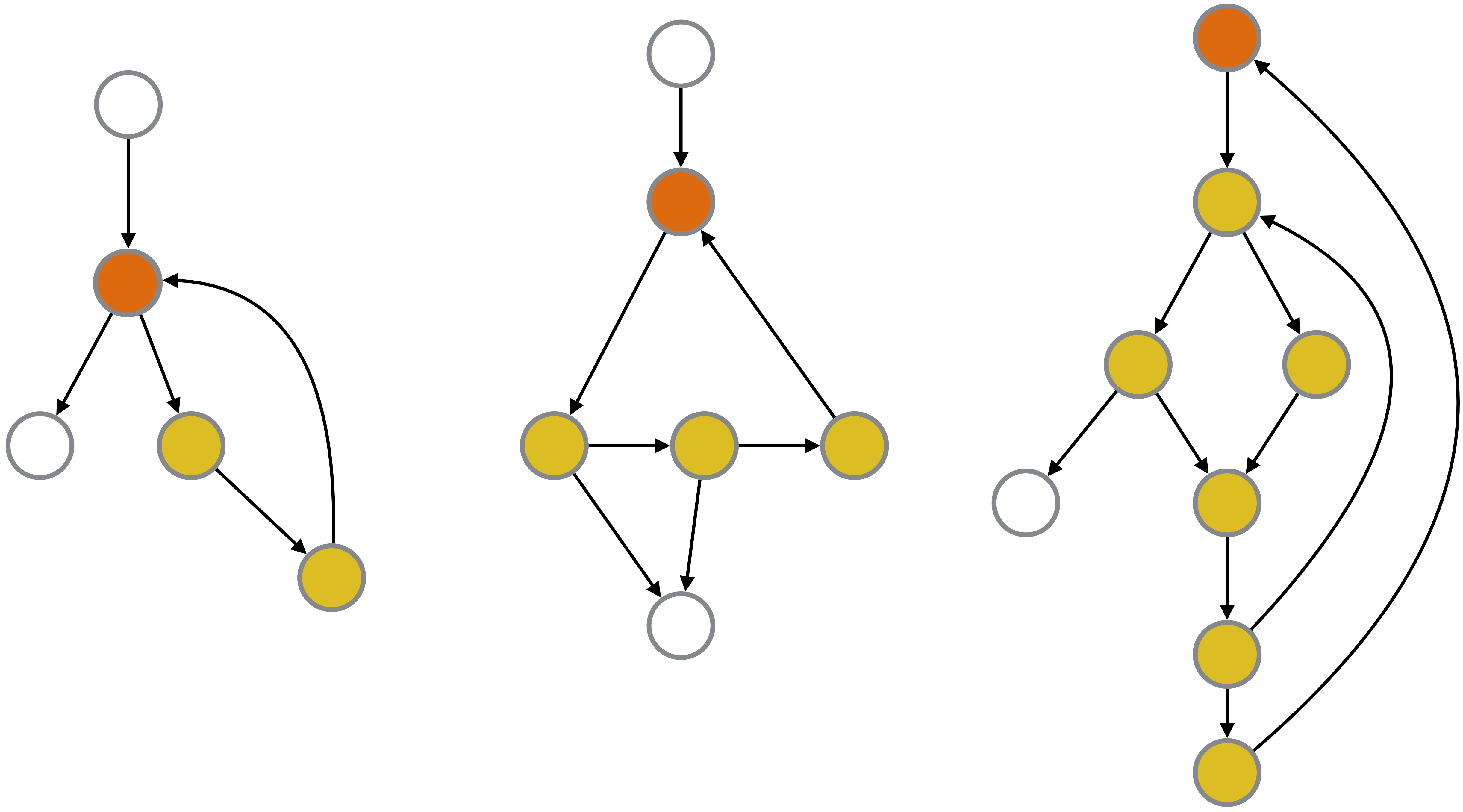
Loop Analysis

- How do we identify loops?
- What is a loop?
  - Can't just "look" at graphs
  - We're going to assume some additional structure
- **Definition:** a **loop** is a subset S of nodes where:
  - S is strongly connected:
    - For any two nodes in S, there is a path from one to the other using only nodes in S
  - There is a distinguished header node $h \in S$ such that there is no edge from a node outside S to $S \setminus \{h\}$

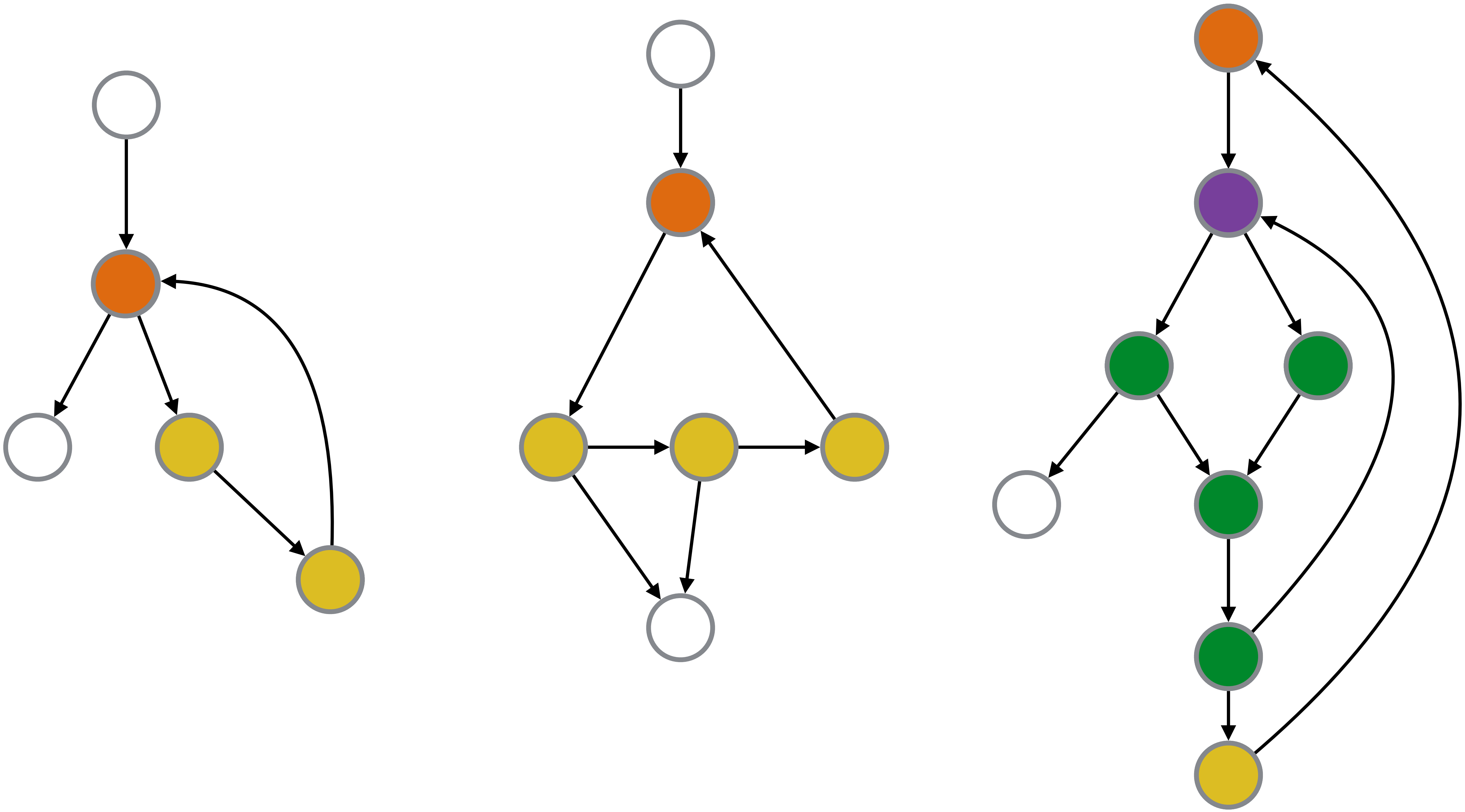
Examples



Examples

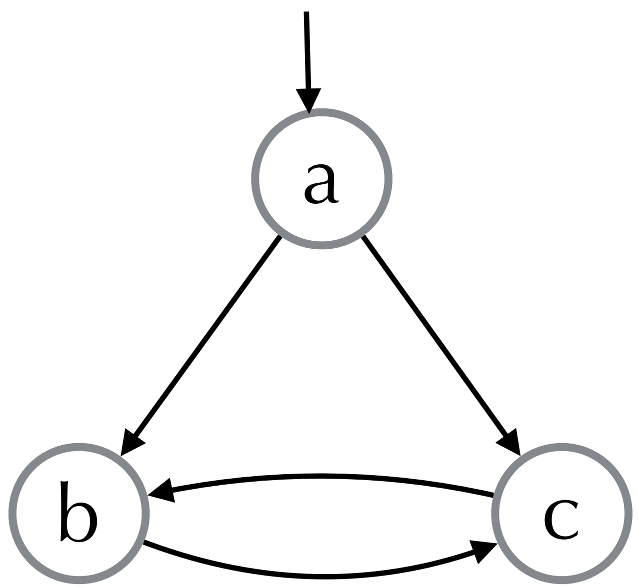


Examples



Non-example

- Consider the following:



- a can't be header
  - No path from b to a or c to a
- b can't be header
  - Has outside edge from a
- c can't be header
  - Has outside edge from a
- So no loop...
- But clearly a cycle!

Reducible Flow Graphs

- So why did we define loops this way?
- Loop header gives us a "handle" for the loop
  - e.g., a good spot for hoisting invariant statements
- Structured control-flow only produces **reducible graphs**
  - a graph where all cycles are loops according to our definition.
  - Java: only reducible graphs
  - C/C++: goto can produce irreducible graph
- Many analyses & loop optimizations depend upon having reducible graphs

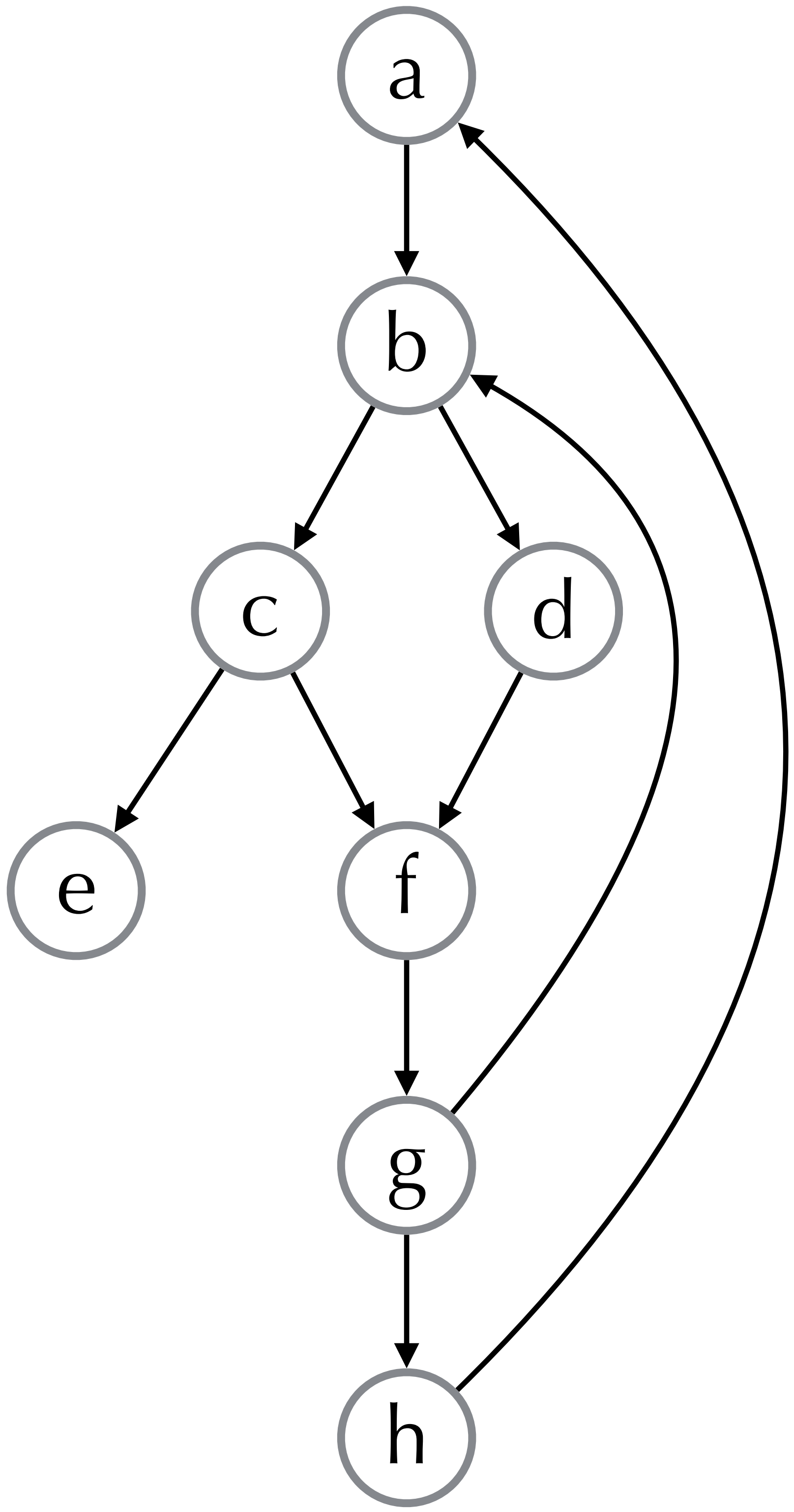
Finding Loops

- **Definition:** node *d* **dominates** node *n* if every path from the start node to *n* must go through *d*
- **Definition:** an edge from *n* to a dominator *d* is called a **back-edge**
- **Definition:** a **loop** of a back edge $n \rightarrow d$ is the set of nodes *x* such that *d* dominates *x* and there is a path from *x* to *n* not including *d*

- So to find loops, we figure out dominators, and identify back edges

Example

- a dominates a,b,c,d,e,f,g,h
- b dominates b,c,d,e,f,g,h
- c dominates c,e
- d dominates d
- e dominates e
- f dominates f,g,h
- g dominates g,h
- h dominates h
- back-edges?
  - g→b
  - h→a
- loops?



Calculating Dominators

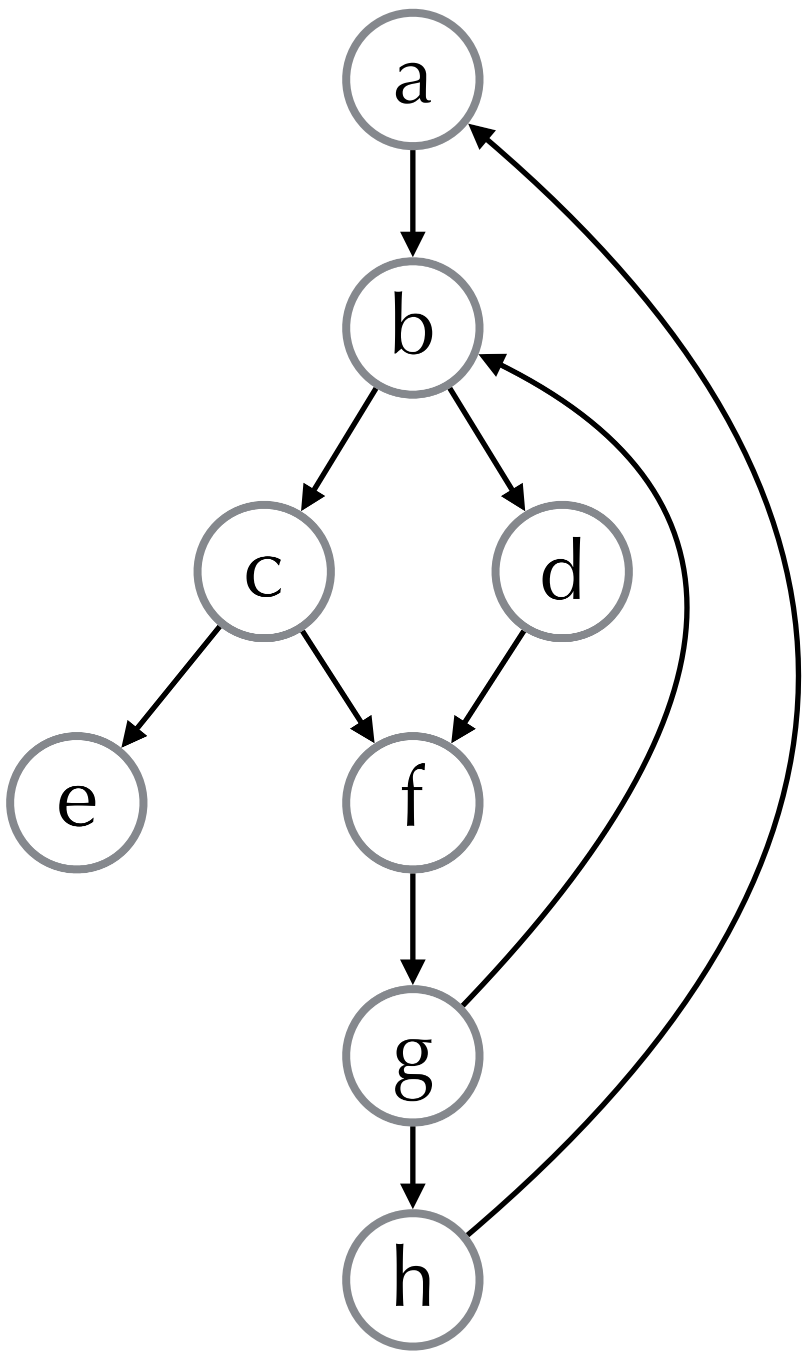
- $D[n]$: the set of nodes that dominate n
- $D[n] = \{n\} \cup (D[p_1] \cap D[p_2] \cap \ldots \cap D[p_m])$
  where $pred[n] = \{p_1, p_2, \ldots, p_m\}$
- It's pretty easy to solve this equation:
  - start off assuming $D[n]$ is all nodes.
    - except for the start node (which is dominated only by itself)
  - iteratively update $D[n]$ based on predecessors until you reach a fixed point

Representing Dominators

- Don't actually need to keep set of all dominators for each node
- Instead, construct a **dominator tree**
  - Insight: if both *d* and *e* dominate *n*, then either *d* dominates *e* or vice versa
  - So that means that node *n* has a "closest" or **immediate dominator**

Example

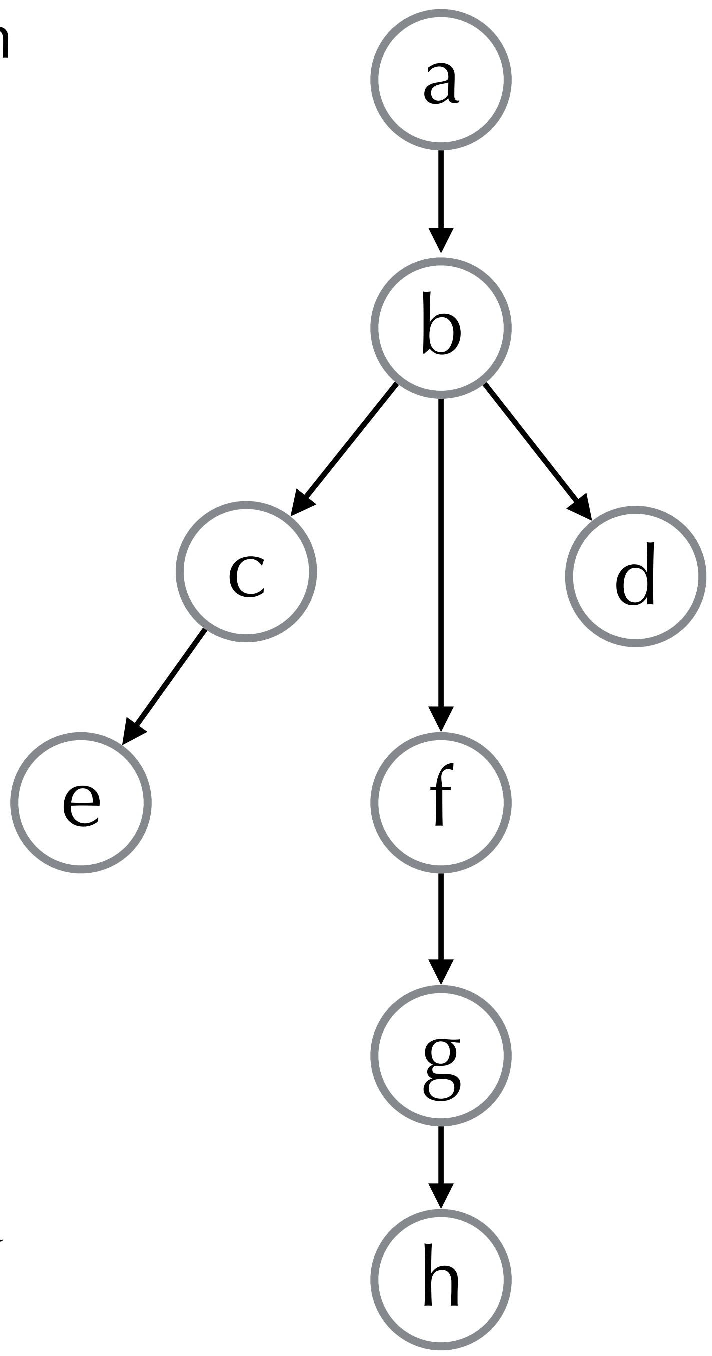
CFG



a dominates a,b,c,d,e,f,g,h
b dominates b,c,d,e,f,g,h
c dominates c,e
d dominates d
e dominates e
f dominates f,g,h
g dominates g,h
h dominates h

a dominated by a
b dominated by b,a
c dominated by c,b,a
d dominated by d,b,a
e dominated by e,c,b,a
f dominated by f,b,a
g dominated by g,f,b,a
h dominated by h,g,f,b,a

Immediate Dominator Tree



Nested Loops

- If loops A and B have distinct headers and all nodes in B are in A (i.e., B⊆A), then we say B is **nested** within A
- An **inner loop** is a nested loop that doesn't contain any other loops
- We usually concentrate our attention on nested loops first (since we spend most time in them)

Loop-Invariant Removal

Loop Invariants

- An assignment `x := v1 op v2` (v_1, v_2) is **invariant** for a loop if for each operand v_1 and v_2 either
  - the operand is constant, or
  - all of the definitions that reach the assignment are outside the loop, or
  - only one definition reaches the assignment and it is a loop invariant

Example

```
L0:  t := 0
     a := x
L1:  i := i + 1
     b := 7
     t := a + b
     *i := t
     if i<N goto L1 else L2

L2:  x := t
```

Hoisting

- We would like to **hoist** invariant computations out of the loop
- But this is trickier than it sounds:
  - We need to potentially introduce an extra node in the CFG, right before the header to place the hoisted statements (the **pre-header**)
  - Even then, we can run into trouble…

Valid Hoisting Example

```
L0:  t := 0

L1:  i := i + 1
     t := a + b
     *i := t
     if i<N goto L1 else L2

L2:  x := t
```

Valid Hoisting Example

```
L0:   t := 0
      t := a + b

L1:   i := i + 1

      *i := t
      if i<N goto L1 else L2

L2:   x := t
```

Invalid Hoisting Example

```
L0:  t := 0


L1:  i := i + 1
     *i := t
     t := a + b
     if i<N goto L1 else L2


L2:  x := t
```

Although `t`'s definition is loop invariant, hoisting conflicts with this use of `t`

Conditions for Safe Hoisting

- An invariant assignment *d*: `x:= v₁ op v₂` is safe to hoist if:
  - *d* dominates all loop exits at which **x** is live and
  - there is only one definition of **x** in the loop, and
  - **x** is not live at the entry point for the loop (the pre-header)

Induction Variable Reduction

Induction Variables

```
      s := 0
      i := 0
L1:   if i >= n goto L2
      j := i*4
      k := j+a
      x := *k
      s := s+x
      i := i+1
L2:   ...
```

- Can express `j` and `k` as linear functions of `i` where the coefficients are either constants or loop-invariant
  - `j` = `4*i + 0`
  - `k` = `4*i + a`

Induction Variables

```
        s := 0
        i := 0
L1:     if i >= n goto L2
        j := i*4
        k := j+a
        x := *k
        s := s+x
        i := i+1
L2:     ...
```

- Note that `i` only changes by the same amount each iteration of the loop
- We say that `i` is a **linear induction variable**
- It's easy to express the change in `j` and `k`
  - Since `j` = `4*i + 0` and `k` = `4*i + a`, if `i` changes by *c*, `j` and `k` change by `4*`*c*

Detecting Induction Variables

- **Definition:** `i` is a **basic induction variable** in a loop *L* if the only definitions of `i` within *L* are of the form `i:=i+`*`c`* or `i:=i-`*`c`* where *`c`* is loop invariant
- **Definition:** `k` is a **derived induction variable** in loop *L* if:
  - 1.There is only one definition of `k` within *L* of the form `k:=j*c` or `k:=j+`*`c`* where `j` is an induction variable and *`c`* is loop invariant; and
  - 2.If `j` is an induction variable in the family of `i` (i.e., linear in `i`) then:
    - the only definition of `j` that reaches `k` is the one in the loop; and
    - there is no definition of `i` on any path between the definition of `j` and the definition of `k`
- If `k` is a derived induction variable in the family of `j` and `j` = *`a`*`*i+`*`b`* and, say, `k:=j*`*`c`*, then *`k`* = *`a`*`*`*`c`*`*i+`*`b`*`*`*`c`*

Strength Reduction

- For each derived induction variable `j` where `j` = e_1`*i + `e_0 make a fresh temp `j'`
- At the loop pre-header, initialize `j'` to e_0
- After each `i:=i+`*`c`*, define `j':=j'+(`e_1`*`*`c`*`)`
  - note that e_1`*`*`c`* can be computed in the loop header (i.e., it's loop invariant)
- Replace the unique assignment of j in the loop with `j := j'`

Example

```
       s  := 0
       i  := 0
L1:    if i >= n goto L2
       j  := i*4
       k  := j+a
       x  := *k
       s  := s+x
       i  := i+1
L2:    ...
```

- `i` is basic induction variable
- `j` is derived induction variable in family of `i`
  - `j` = `4*i + 0`
- `k` is derived induction variable in family of `j`
  - `k` = `4*i + a`

Example

```
        s  := 0
        i  := 0
        j' := 0
        k' := a
L1:     if i >= n goto L2
        j  := i*4
        k  := j+a
        x  := *k
        s  := s+x
        i  := i+1
L2:     ...
```

- `i` is basic induction variable
- `j` is derived induction variable in family of `i`
  - `j` = `4*i + 0`
- `k` is derived induction variable in family of `j`
  - `k` = `4*i + a`

Example

```
      s  := 0
      i  := 0
      j' := 0
      k' := a

L1:   if i >= n goto L2
      j  := i*4
      k  := j+a
      x  := *k
      s  := s+x
      i  := i+1
      j' := j'+4
      k' := k'+4

L2:   ...
```

- `i` is basic induction variable
- `j` is derived induction variable in family of `i`
  - `j` = `4*i + 0`
- `k` is derived induction variable in family of `j`
  - `k` = `4*i + a`

Example

```
      s  := 0
      i  := 0
      j' := 0
      k' := a

L1:   if i >= n goto L2
      j  := j'
      k  := k'
      x  := *k
      s  := s+x
      i  := i+1
      j' := j'+4
      k' := k'+4

L2:   ...
```

- `i` is basic induction variable
- `j` is derived induction variable in family of `i`
  - `j` = `4*i + 0`
- `k` is derived induction variable in family of `j`
  - `k` = `4*i + a`

Example

```
        s  := 0
        i  := 0
        j’:= 0
        k’:= a

L1:     if i >= n goto L2
        x  := *k’
        s  := s+x
        i  := i+1
        j’:= j’+4
        k’:= k’+4

L2:     ...
```

- `i` is basic induction variable
- `j` is derived induction variable in family of `i`
  - `j` = `4*i + 0`
- `k` is derived induction variable in family of `j`
  - `k` = `4*i + a`

Useless Variables

```
       s  := 0
       i  := 0
       j’:= 0
       k’:= a

L1:    if i >= n goto L2
       x  := *k’
       s  := s+x
       i  := i+1
       j’:= j’+4
       k’:= k’+4

L2:    ...
```

- A variable is **useless** for *L* if it is dead at all exits from L and its only use is in a definition of itself
  - E.g., `j’` is useless
- Can delete useless variables

Useless Variables

```
       s  := 0
       i  := 0
       j’ := 0
       k’ := a
L1:    if i >= n goto L2
       x  := *k’
       s  := s+x
       i  := i+1
       k’ := k’+4

L2:    ...
```

- A variable is **useless** for *L* if it is dead at all exits from L and its only use is in a definition of itself
  - E.g., `j’` is useless
- Can delete useless variables

Useless Variables

```
        s  := 0
        i  := 0
        k’:= a

L1:     if i >= n goto L2
        x  := *k’
        s  := s+x
        i  := i+1
        k’:= k’+4

L2:     ...
```

- A variable is **useless** for *L* if it is dead at all exits from L and its only use is in a definition of itself
  - E.g., `j’` is useless
- Can delete useless variables

Almost Useless Variables

```
      s  := 0
      i  := 0
      k’:= a
L1:   if i >= n goto L2
      x  := *k’
      s  := s+x
      i  := i+1
      k’:= k’+4

L2:   ...
```

- A variable is **almost useless** for *L* if it is used only in comparison against loop invariant values and in definitions of itself, and there is some other non-useless induction variable in same family
    - E.g., `i` is almost useless
- An almost-useless variable may be made useless by modifying comparison
    - See Appel for details

Loop Fusion and Loop Fission

- Fusion: combine two loops into one
- Fission: split one loop into two

Loop Fusion

- Before

```
int acc = 0;
for (int i = 0; i < n; ++i) {
  acc += a[i];
  a[i] = acc;
}
for (int i = 0; i < n; ++i) {
  b[i] += a[i];
}
```

- After

```
int acc = 0;
for (int i = 0; i < n; ++i) {
  acc += a[i];
  a[i] = acc;
  b[i] += acc;
}
```

- What are the potential benefits? Costs?
- Locality of reference

Loop Fission

- Before

```
for (int i = 0; i < n; ++i) {
  a[i] = e1;
  b[i] = e2;   // e1 and e2 independent
}
```

- After

```
for (int i = 0; i < n; ++i) {
  a[i] = e1;
}
for (int i = 0; i < n; ++i) {
  b[i] = e2;
}
```

- What are the potential benefits? Costs?
- Locality of reference

Loop Unrolling

- Make copies of loop body
- Say, each iteration of rewritten loop performs 3 iterations of old loop

Loop Unrolling

- Before

```
for (int i = 0; i < n; ++i) {
  a[i] = b[i] * 7 + c[i] / 13;
}
```

- After

```
for (int i = 0; i < n % 3; ++i) {
   a[i] = b[i] * 7 + c[i] / 13;
}
for (; i < n; i += 3) {
  a[i] = b[i] * 7 + c[i] / 13;
  a[i + 1] = b[i + 1] * 7 + c[i + 1] / 13;
  a[i + 2] = b[i + 2] * 7 + c[i + 2] / 13;
}
```

- What are the potential benefits? Costs?
- Reduce branching penalty, end-of-loop-test costs
- Size of program increased

Loop Unrolling

- If fixed number of iterations, maybe turn loop into sequence of statements!
- Before

```
for (int i = 0; i < 6; ++i) {
  if (i % 2 == 0) foo(i); else bar(i);
}
```

- After

```
foo(0);
bar(1);
foo(2);
bar(3);
foo(4);
bar(5);
```

Loop Interchange

- Change order of loop iteration variables

Loop Interchange

- Before

```
for (int j = 0; j < n; ++j) {
  for (int i = 0; i < n; ++i) {
    a[i][j] += 1;
  }
}
```

- After

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    a[i][j] += 1;
  }
}
```

- What are the potential benefits? Costs?
  - Locality of reference

Loop Peeling

- Split first (or last) few iterations from loop and perform them separately

Loop Peeling

- Before

```
for (int i = 0; i < n; ++i) {
  b[i] = (i == 0) ? a[i] : a[i] + b[i-1];
}
```

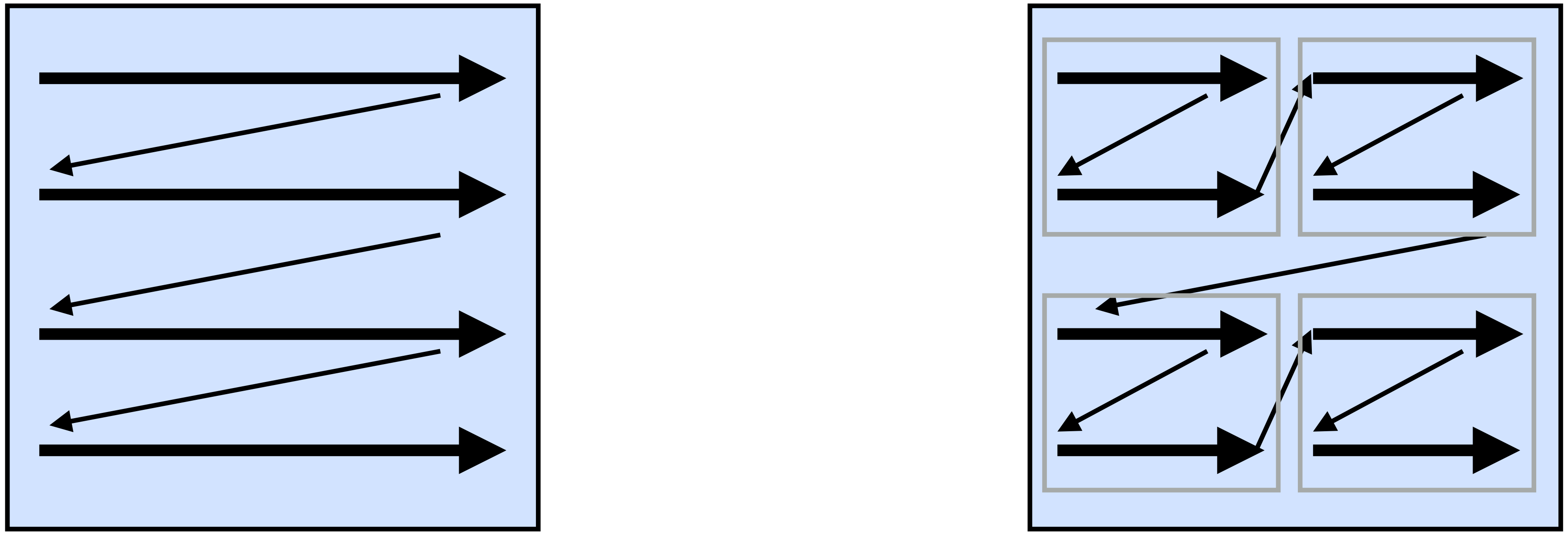
- After

```
b[0] = a[0];
for (int i = 1; i < n; ++i) {
  b[i] = a[i] + b[i-1];
}
```

- What are the potential benefits? Costs?

Loop Tiling

- For nested loops, change iteration order



Loop Tiling

- Before

```
for (i = 0; i < n; i++) {
  c[i] = 0;
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

- After:

```
for (i = 0; i < n; i += 4) {
    c[i] = 0;
    c[i + 1] = 0;
    for (j = 0; j < n; j += 4) {
      for (x = i; x < min(i + 4, n); x++) {
        for (y = j; y < min(j + 4, n); y++) {
          c[x] = c[x] + a[x][y] * b[y];
        }
      }
    }
  }
```

- What are the potential benefits? Costs?

Loop Parallelization

- Before

```
for (int i = 0; i < n; ++i) {
  a[i] = b[i] + c[i]; // a, b, and c do not overlap
}
```

- After

```
for (int i = 0; i < n % 4; ++i) a[i] = b[i] + c[i];
for (; i < n; i = i + 4) {
  __some4SIMDadd(a+i,b+i,c+i);
}
```

- What are the potential benefits? Costs?