Compilation 2024

Optimization

Aslan Askarov
aslan@cs.au.dk

Acknowledgments: Stephen Chong

Part 1

- Optimizations
  - Safety
  - Constant folding
  - Algebraic simplification
    - Strength reduction
  - Constant propagation
  - Copy propagation
  - Dead code elimination
  - Inlining and specialization
    - Recursive function inlining
  - Tail call elimination
  - Common subexpression elimination

Why do we need optimizations?

- To help programmers…
  - They write modular, clean, high-level programs
  - Compiler generates efficient, high-performance assembly
- Programmers don't write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. `A[i][j] = A[i][j] + 1`
- Architectural independence
  - Optimal code depends on features not expressed to the programmer
  - Modern architectures assume optimization
- Different kinds of optimizations:
  - Time: improve execution speed
  - Space: reduce amount of memory needed
  - Power: lower power consumption (e.g. to extend battery life)

Some caveats

- Optimization are code transformations:
  - They can be applied at any stage of the compiler
  - They must be safe – they shouldn’t change the meaning of the program.
- In general, optimizations require some program analysis:
  - To determine if the transformation really is safe
  - To determine whether the transformation is cost effective
- “Optimization” is misnomer
  - Typically no guarantee transformations will improve performance, nor that compilation will produce optimal code
- This course: most common and valuable performance optimizations
  - See Muchnick “Advanced Compiler Design and Implementation” for ~10 chapters about optimization

Constant Folding

- Idea: If operands are known at compile type, perform the operation statically.
- `int x = (2+3) * y` ➔ `int x = 5 * y`
- `b & false` ➔ `false`

Constant Folding

```
int x = (2+3) * y  ➔  int x = 5 * y
```

- What performance metric does it intend to improve?
  - In general, the question of whether an optimization improves performance is undecidable.
- At which compilation step can it be applied?
  - Intermediate Representation
  - Can be performed after other optimizations that create constant expressions.

Constant Folding

```
int x = (2+3) * y  ➔  int x = 5 * y
```

- When is it safely applicable?
  - For Boolean values, yes.
  - For integers, almost always yes.
    - An exception: division by zero.
  - For floating points, use caution.
    - Example: rounding
- General notes about safety:
  - Whether an optimization is safe depends on language semantics.
    - Languages that provide weaker guarantees to the programmer permit more optimizations, but have more ambiguity in their behavior.
  - Is there a formal proof for safety?

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules.
- Identities:
  - `a * 1 → a` `a * 0 → 0`
  - `a + 0 → a` `a – 0 → a`
  - `b | false → b` `b & true → b`
- Reassociation & commutativity:
  - `(a + b) + c → a + (b + c)`
  - `a + b → b + a`

Algebraic Simplification

- Combined with Constant Folding:
  - `(a + 1) + 2 → a + (1 + 2) → a + 3`
  - `(2 + a) + 4 → (a + 2) + 4 → a + (2 + 4) → a + 6`
- Iteration of these optimizations is useful…
  - How much?

Strength Reduction

- Replace expensive op with cheaper op:
  - `a * 4  →  a << 2`
  - `a * 7  →  (a << 3) – a`
  - `a / 32767  →  (a >> 15) + (a >> 30)`
- So, the effectiveness of this optimization depends on the architecture.

Constant Propagation

- If the value of a variable is known to be a constant, replace the use of the variable by that constant.
- Value of the variable must be propagated forward from the point of assignment.
  - This is a substitution operation.
- Example:

```
int x = 5;
int y = x * 2;
int z = a[y];
```

➔

```
int y = 5 * 2;
int z = a[y];
```

➔

```
int y = 10;
int z = a[y];
```

➔

```
int z = a[10];
```

- To be most effective, constant propagation can be interleaved with constant folding.

Constant Propagation

- For safety, it requires a data-flow analysis.
- What performance metric does it intend to improve?
- At which compilation step can it be applied?
- What is the computational complexity of this optimization?

Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;
if (x > 1) {
  x = x * f(x – 1);
}
```

➔

```
x = y;
if (y > 1) {
  x = y * f(y – 1);
}
```

- Can make the first assignment to x **dead code** (that can be eliminated).

Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x  = y * y // x is dead!
...        // x never used
x = z * z
```

➔

```
...
x = z * z
```

- A variable is **dead** if it is never used after it is defined.
  - Computing such **definition** and **use** information is an important component of compiler
- Dead variables can be created by other optimizations…
- Code for computing the value of a dead variable can be dropped.

Dead Code Elimination

- Is it always safely applicable?
  - Only if that code is **pure** (i.e. it has no externally visible side effects).
    - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket, ...
    - Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

Unreachable Code Elimination

- Basic blocks not reachable by any trace leading from the starting basic block are **unreachable** and can be deleted.
- At which compilation step can it be applied?
  - IR or assembly level
- What performance metric does it intend to improve?
  - Improves instruction cache utilization.

Common Subexpression Elimination

- Idea: replace an expression with previously stored evaluations of that expression.
- Example:

  ```
  [a + i*4] = [a + i*4] + 1
  ```

  - Common subexpression elimination removes the redundant add and multiply:

    ```
    t = a + i*4; [t] = [t] + 1
    ```

- For safety, you must be sure that the shared expression always has the same value in both places!

Unsafe Common Subexpression Elimination

- As an example, consider function:

```
void f(int[] a, int[] b, int[] c) {
   int j = …; int i = …; int k = …;
   b[j] = a[i] + 1;
   c[k] = a[i];
   return;
}
```

- The following optimization that shares expression `a[i]` is unsafe… Why?

```
void f(int[] a, int[] b, int[] c) {
   int j = …; int i = …; int k = …;
   t = a[i];
   b[j] = t + 1;
   c[k] = t;
   return;
}
```

Common Subexpression Elimination

- Almost always improves performance.
- But sometimes…
  - It might be less expensive to recompute an expression, rather than to allocate another register to hold its value (or to store it in memory and later reload it).

Loop-invariant Code Motion

- Idea: hoist invariant code out of a loop.

```
while (b) {
  z = y/x;
  …              // y, x not updated
}
```

➔

```
z = y/x;
while (b) {
  …              // y, x not updated
}
```

- What performance metric does it intend to improve?
- Is this always safe?

Optimization Example

```
let a = x ** 2 in
let b = 3 in
let c = x in
let d = c * c in
let e = b * 2 in
let f = a + d in
e * f
```

Copy and constant propagation

```
let a = x ** 2 in
let d = x * x in
let e = 3 * 2 in
let f = a + d in
e * f
```

Constant folding

```
let a = x ** 2 in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

Strength reduction

```
let a = x * x in
let d = x * x in
let e = 6 in
let f = a + d in
e * f
```

Common sub-expression elimination

```
let a = x * x in
let d = a in
let e = 6 in
let f = a + d in
e * f
```

Copy and constant propagation

```
let a = x * x in
let f = a + a in
6 * f
```

Loop Unrolling

- Idea: replace the body of a loop by several copies of the body and adjust the loop-control code.
- Example:
  - Before unrolling:

```
for(int i=0; i<100; i=i+1) {
  s = s + a[i];
}
```

  - After unrolling:

```
for(int i=0; i<99; i=i+2){
  s = s + a[i];
  s = s + a[i+1];
}
```

Loop Unrolling

- What performance metric does it intend to improve?
  - Reduces the overhead of branching and checking the loop-control.
    - But it yields larger loops, which might impact the instruction cache.
- Which loops to unroll and by what factor?
  - Some heuristics:
    - Body with straight-line code.
    - Simple loop-control.
  - Use profiled runs.
- It may improve the effectiveness of other optimizations (e.g., common-subexpression evaluation).

Inlining

- Replace call to a function with function body (rewrite arguments to be local variables).
- Example:

```
int g(int x) { return x + pow(x); }

int pow(int a) {
  int b = 1; int n = 0;
  while (n < a) {b = 2 * b};
  return b;
}
```

➔

```
int g(int x) {
  int a = x;
  int b = 1; int n = 0;
  while (n < a) {b = 2 * b};
  tmp = b;
  return x + tmp;
}
```

- Eliminates the stack manipulation, jump, etc.
- May need to rename variable names to avoid **name capture.**
  - Example of what can go wrong?
- Best done at the AST or relatively high-level IR.
  - Enables further optimizations.

Inlining Recursive Functions

- Consider recursive function:
```
f(x,y) = if x < 1 then y
         else x * f(x-1,y)
```
- If we inline it, we essentially just unroll one call:
  - `f(z,8) + 7`
    becomes
    `(if z < 0 then 8 else z*f(z-1,8)) + 7`
  - Can't keep on inlining definition of `f`; will never stop!
- But can still get some benefits of inlining by slight rewriting of recursive function...

Rewriting Recursive Functions for Inlining

- Rewrite function to use a loop pre-header

```
function f(a₁,...,aₙ) = e
```

becomes

```
function f(a₁,...,aₙ) =
   let function f′(a₁,...,aₙ) = e[f↦f']
   in f′(a₁,...,aₙ)
```

- Example:

```
function f(x,y) = if x < 1 then y else x * f(x-1,y)

function f(x,y) =
   let function f′(x,y) = if x < 1 then y
                          else x * f′(x-1,y)
   in f′(x,y)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
   let function f’(x,y) = if x < 1 then y
                          else x * f’(x-1,y)
   in f’(x,y)
```

- Remove **loop-invariant arguments**
  - e.g., `y` is invariant in calls to `f’`

```
function f(x,y) =
   let function f’(x) = if x < 1 then y
                        else x * f’(x-1)
   in f’(x)
```

Rewriting Recursive Functions for Inlining

```
function f(x,y) =
   let function f'(x) = if x < 1 then y
                                   else x * f'(x-1)
   in f'(x)
```

6+f(4,5) becomes:

```
6 +
(let function f'(x)=
 if x < 1 then 5
  else x * f'(x-1)
in f'(4))
```

Without rewriting f,

6+f(4,5) becomes:

```
6 +
(if 4 < 1 then 5
  else 4 * f(3,5))
```

Rewriting Recursive Functions for Inlining

- Now inlining recursive function is more useful!
  - Can *specialize* the recursive function!
    - Additional optimizations for the specific arguments can be enabled (e.g., copy propagation, dead code elimination).

When to Inline

- Code inlining might increase the code size.
  - Impact on cache misses.
- Some heuristics for when to inline a function:
  - Expand only function call sites that are called frequently
    - Determine frequency by execution profiler or by approximating statically (e.g., loop depth)
  - Expand only functions with small bodies
    - Copied body won't be much larger than code to invoke function
  - Expand functions that are called only once
    - Dead function elimination will remove the now unused function

Tail Call Elimination

- Consider two recursive functions:

```
let add(m,n) = if (m=0) then n else 1 + add(m-1,n)

let add(m,n) = if (m=0) then n else add(m-1,n+1)
```

- First function: after recursive call to `add`, still have computation to do (i.e., add `1`).
- Second function: after recursive call, nothing to do but return to caller.
  - This is a **tail call**.

Tail Call Elimination

```
let add(m,n) = if (m=0) then n else add(m-1,n+1)
```

Equivalent program in an imperative language

```
int add(int m, int n){
 if (m=0) then
  return n
 else
  return add(m-1,n+1)}
```

Tail Call Elimination

```
int add(int m, int n){
loop:
 if (m=0) then
  return n
 else
  m:=m-1;
  n:=n+1;
  goto loop }
```

Tail Call Elimination

- Steps for applying tail call elimination to a recursive procedure:
  - Replace recursive call by updating the parameters.
  - Branch to the beginning of the procedure.
  - Delete the **`return`**.
- Reuse stack frame!
  - Don’t need to allocate new stack frame for recursive call.
- Values of arguments (`n`, `m`) remain in registers.
- Combined with inlining, a recursive function can become as cheap as a while loop.
- Even for non-recursive functions: if last statement is function call (tail call), can still reuse stack frame.

Some Optimizations

High level

AST

- Inlining
- Function specialization

IR

- Constant folding
- Constant propagation
- Value numbering

Mid level

Canonical IR

- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction

Low level

Abstract assembly

- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling

Assembly

- Cache optimization

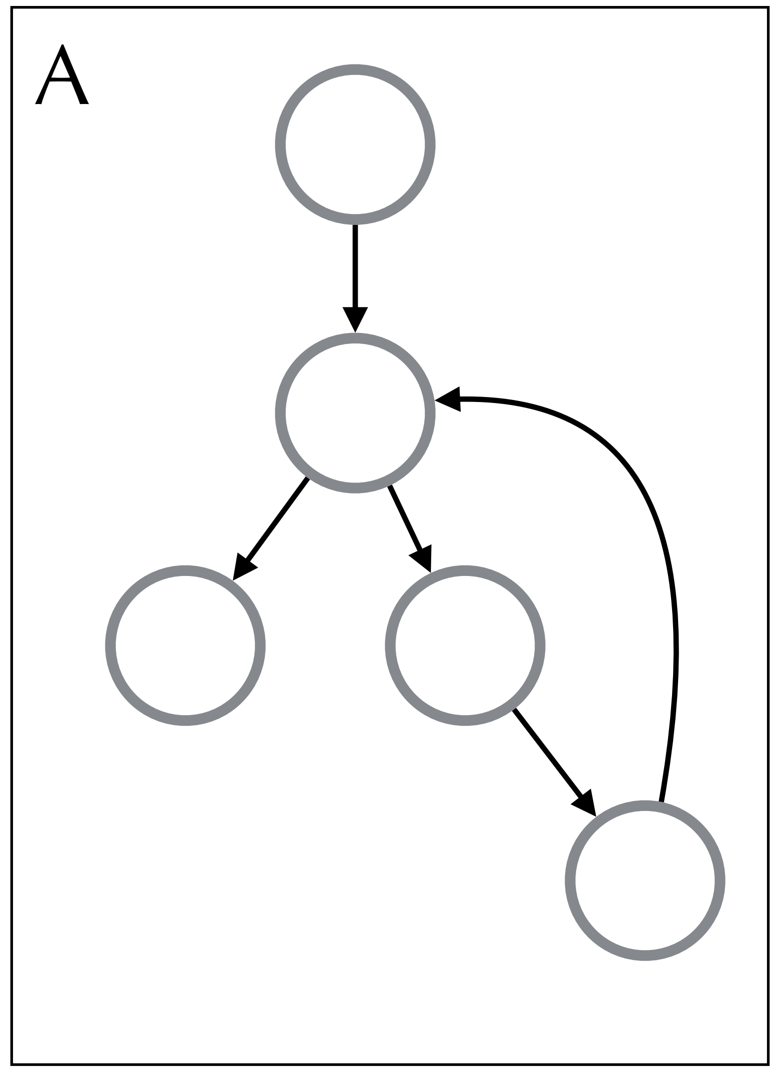
Writing Fast Programs In Practice

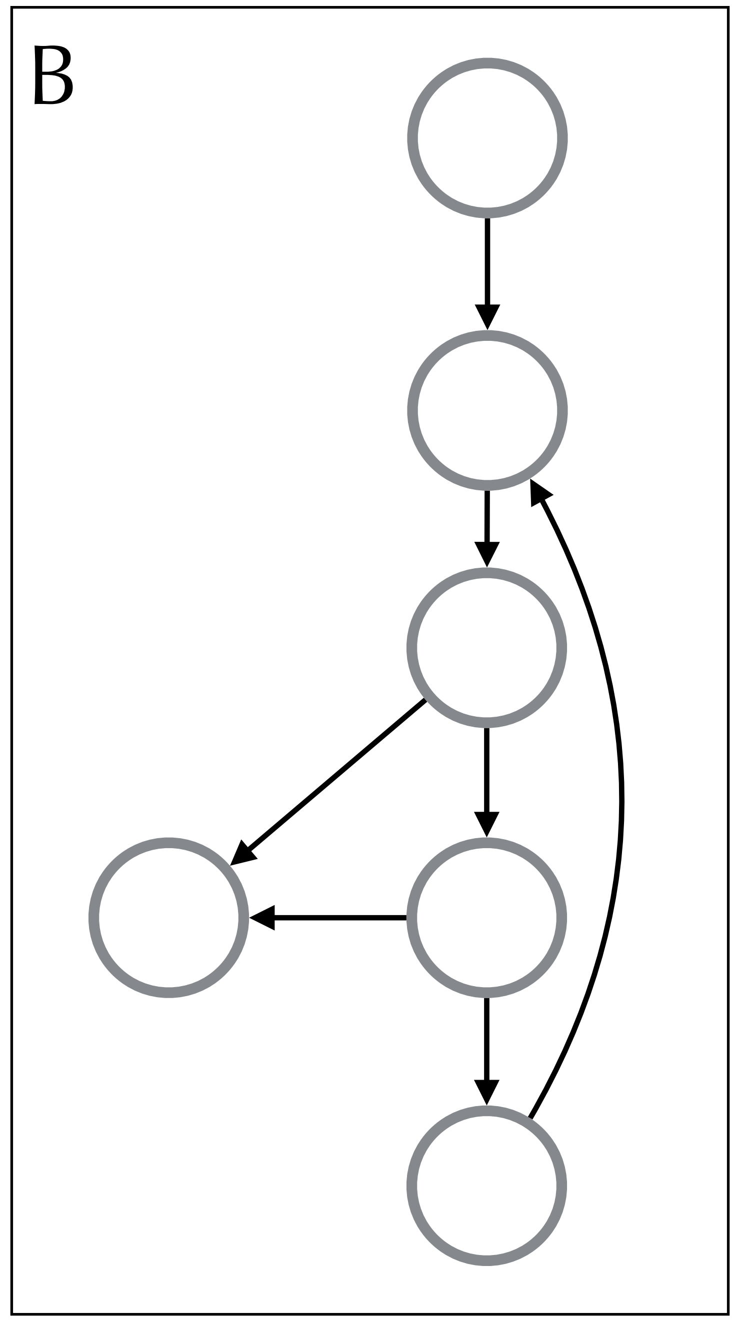
- Pick the right algorithms and data structures.
  - These have a much bigger impact on performance that compiler optimizations.
  - Reduce # of operations
  - Reduce memory accesses
  - Minimize indirection – it breaks working-set coherence
- **Then** turn on compiler optimizations.
- Profile to determine program hot spots.
- Evaluate whether the algorithm/data structure design works.

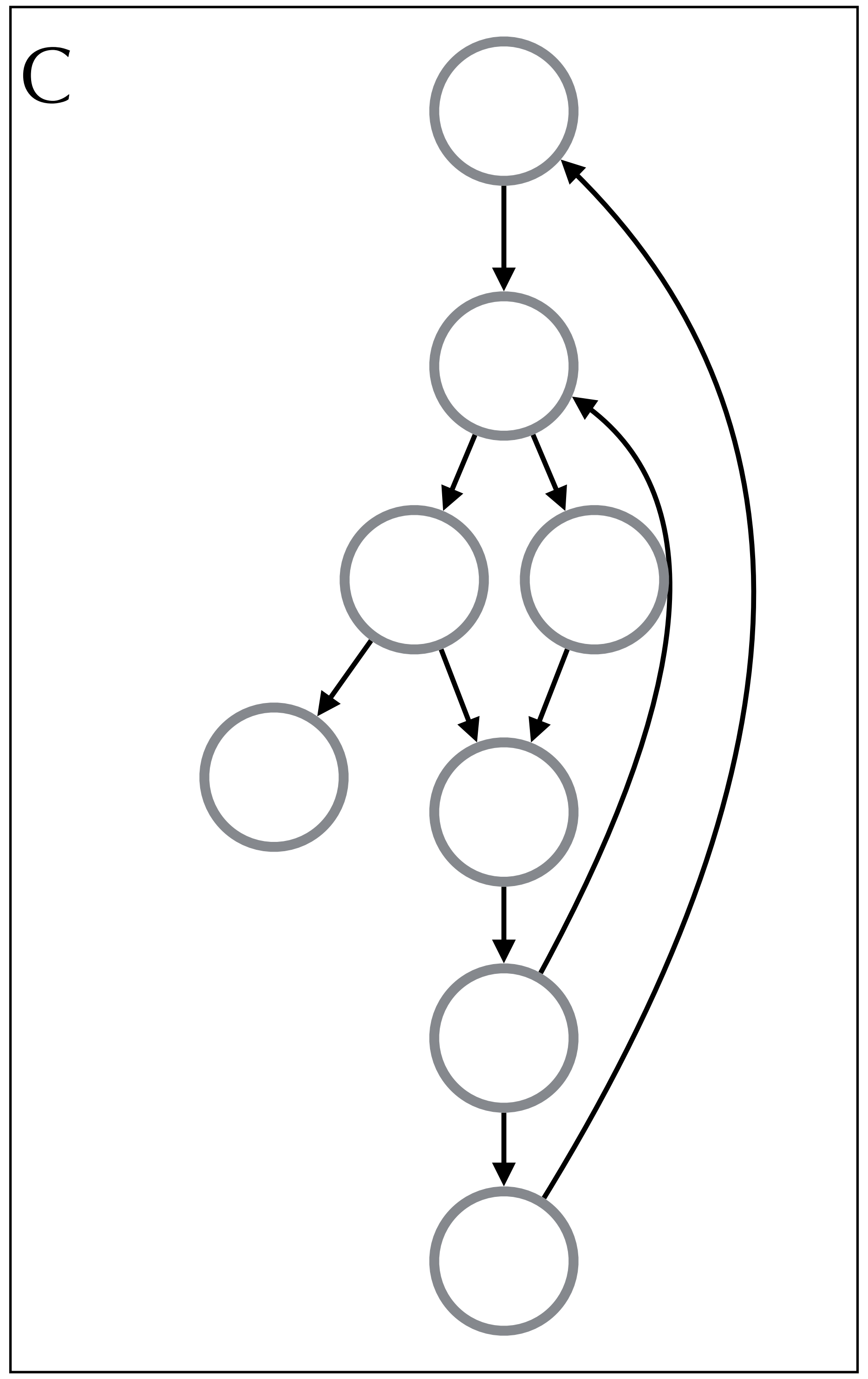
Loop optimization

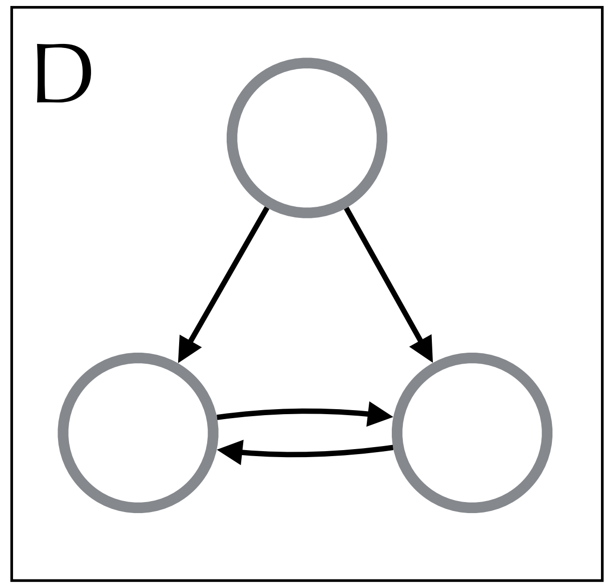
Quiz

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









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

```
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:  ...
```

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

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

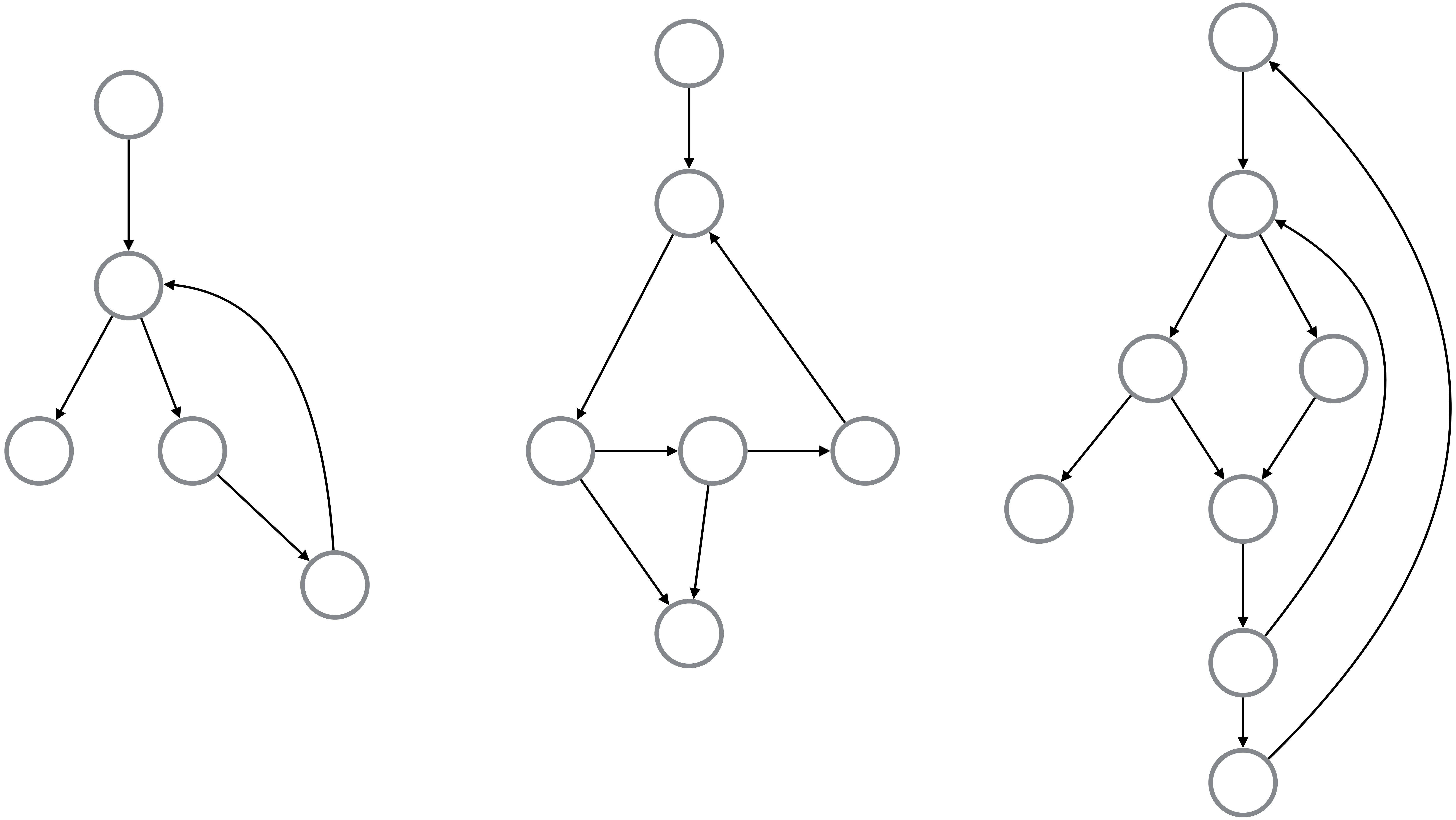
```
L0:   s  := 0
      t2 := a
      t5 := t2 + 400
      jump L2

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`

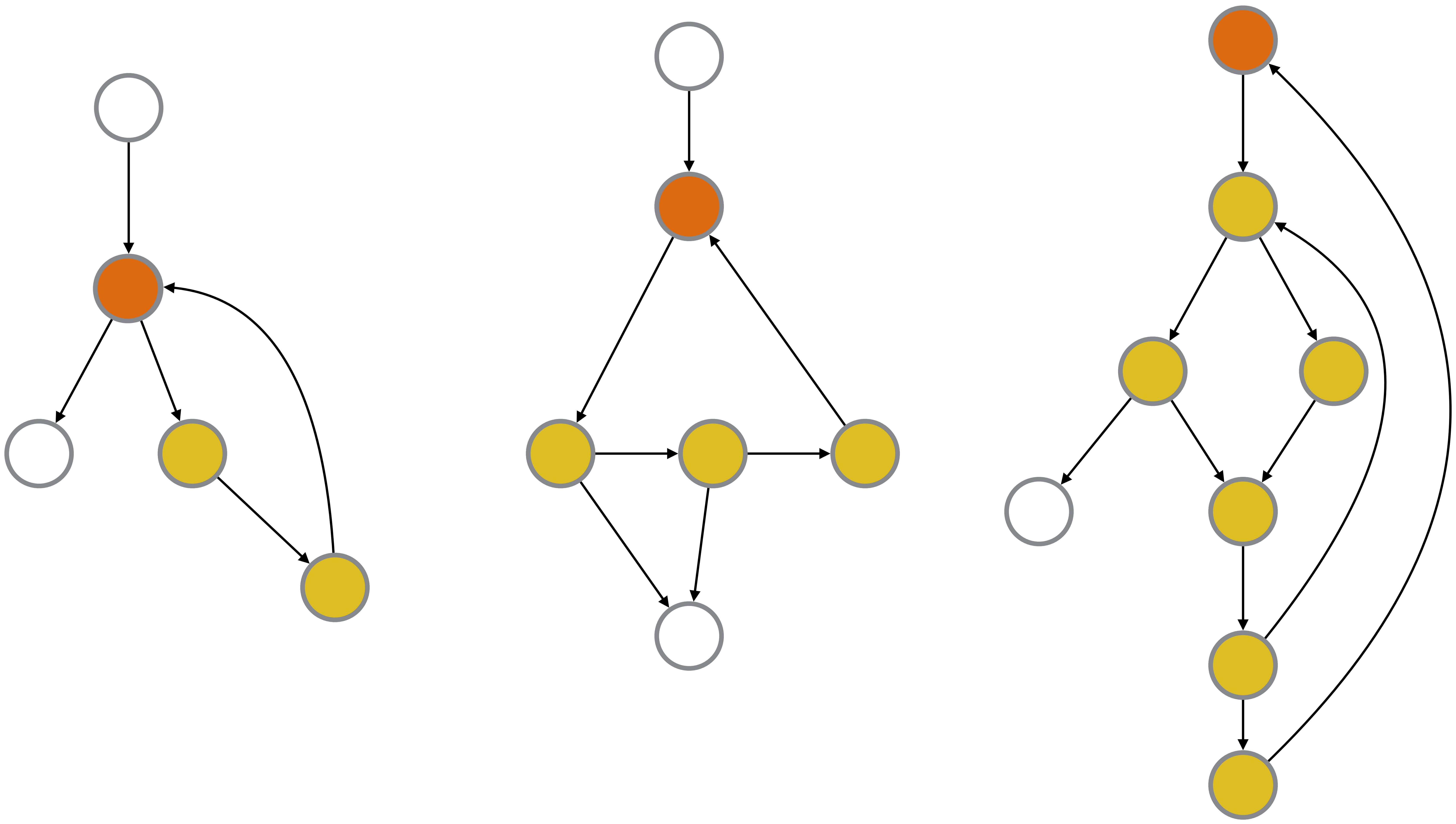
Loop Analysis

- 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 $S \setminus \{h\}$

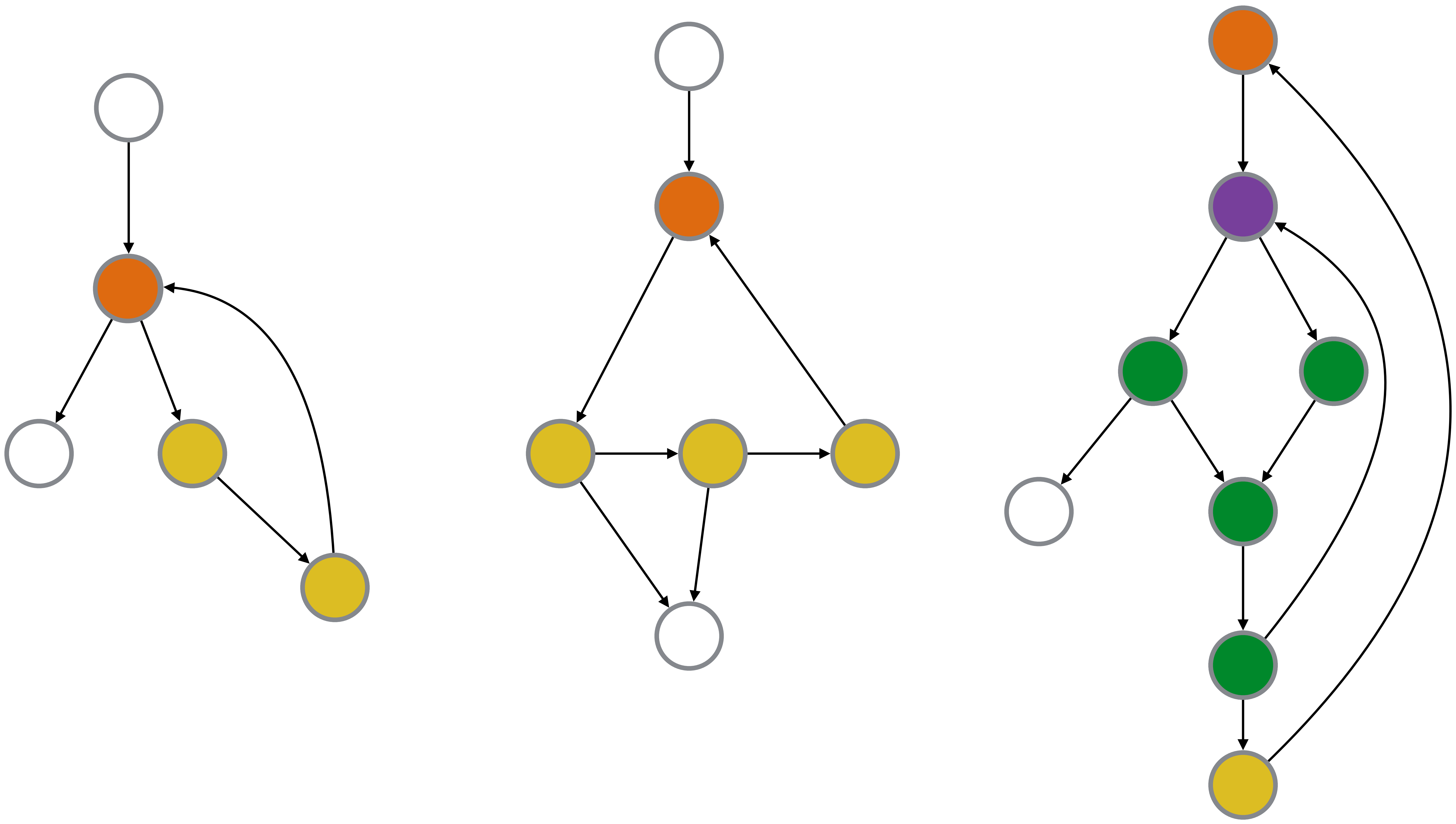
Examples



Examples

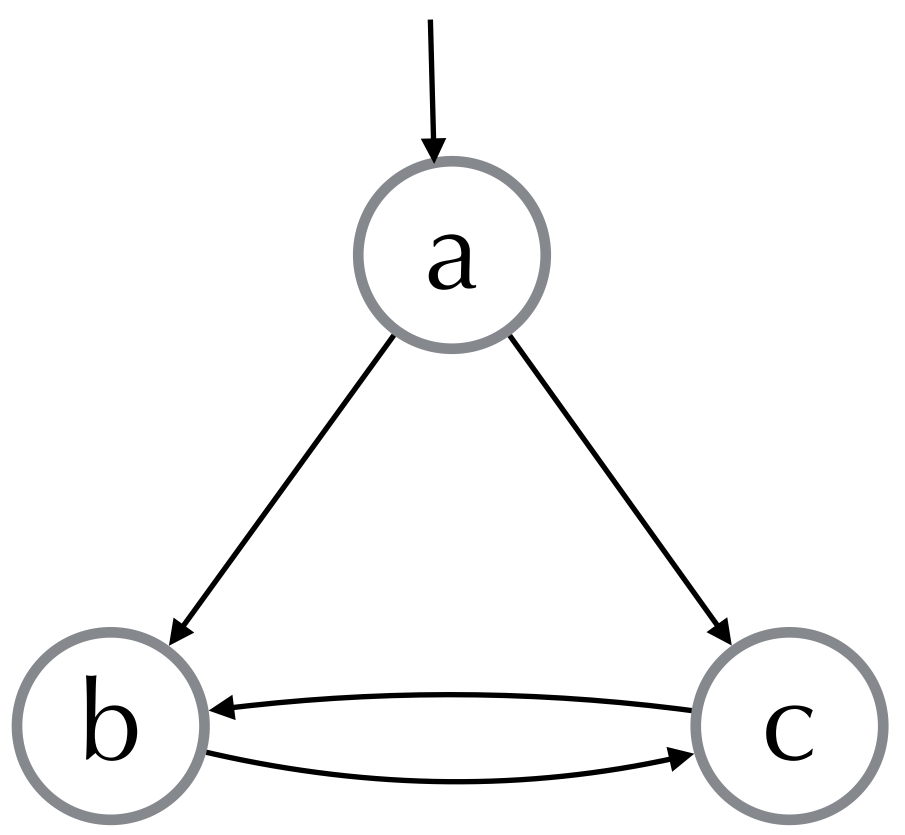


Examples



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

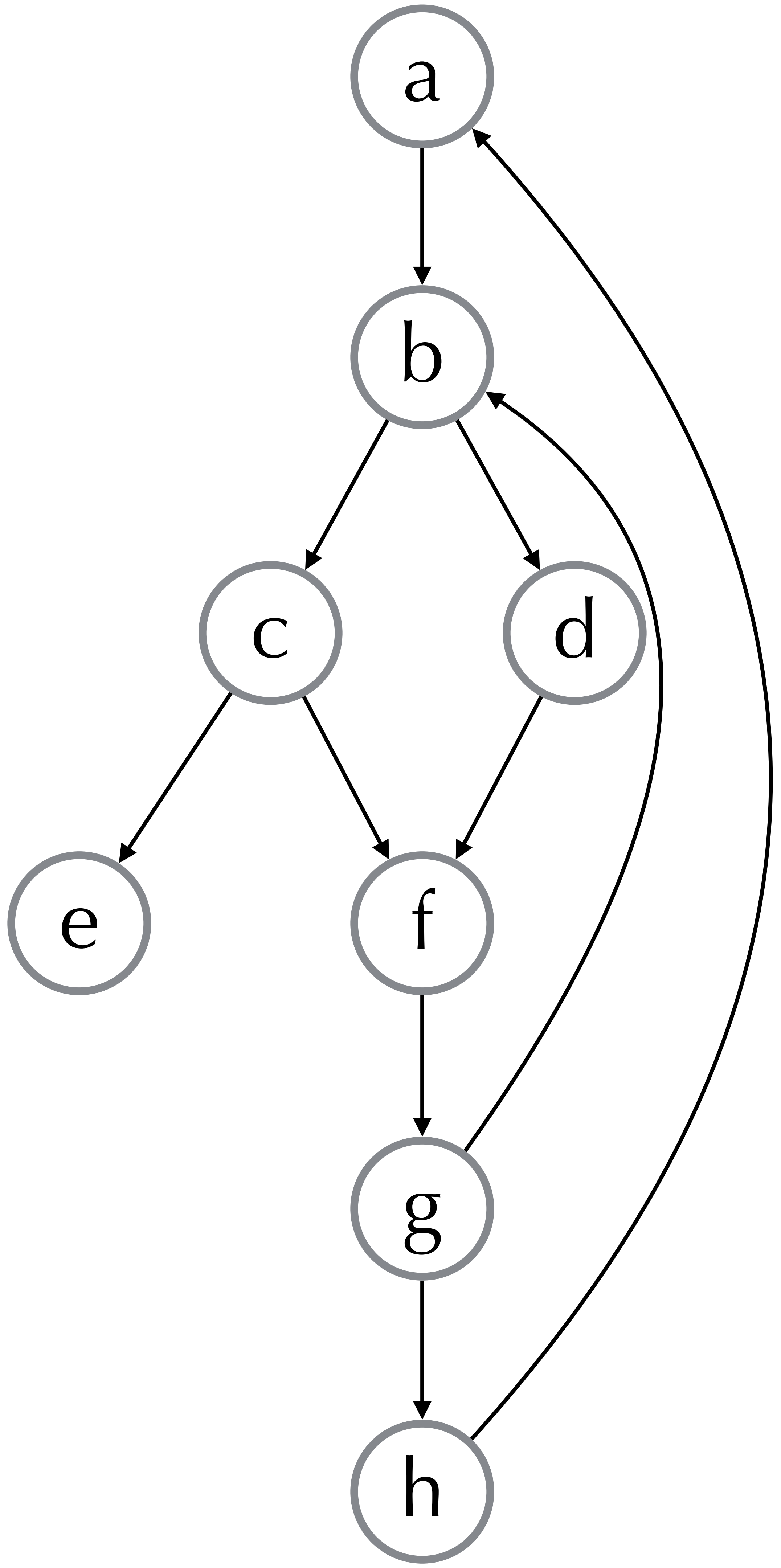
Finding Loops

- **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?



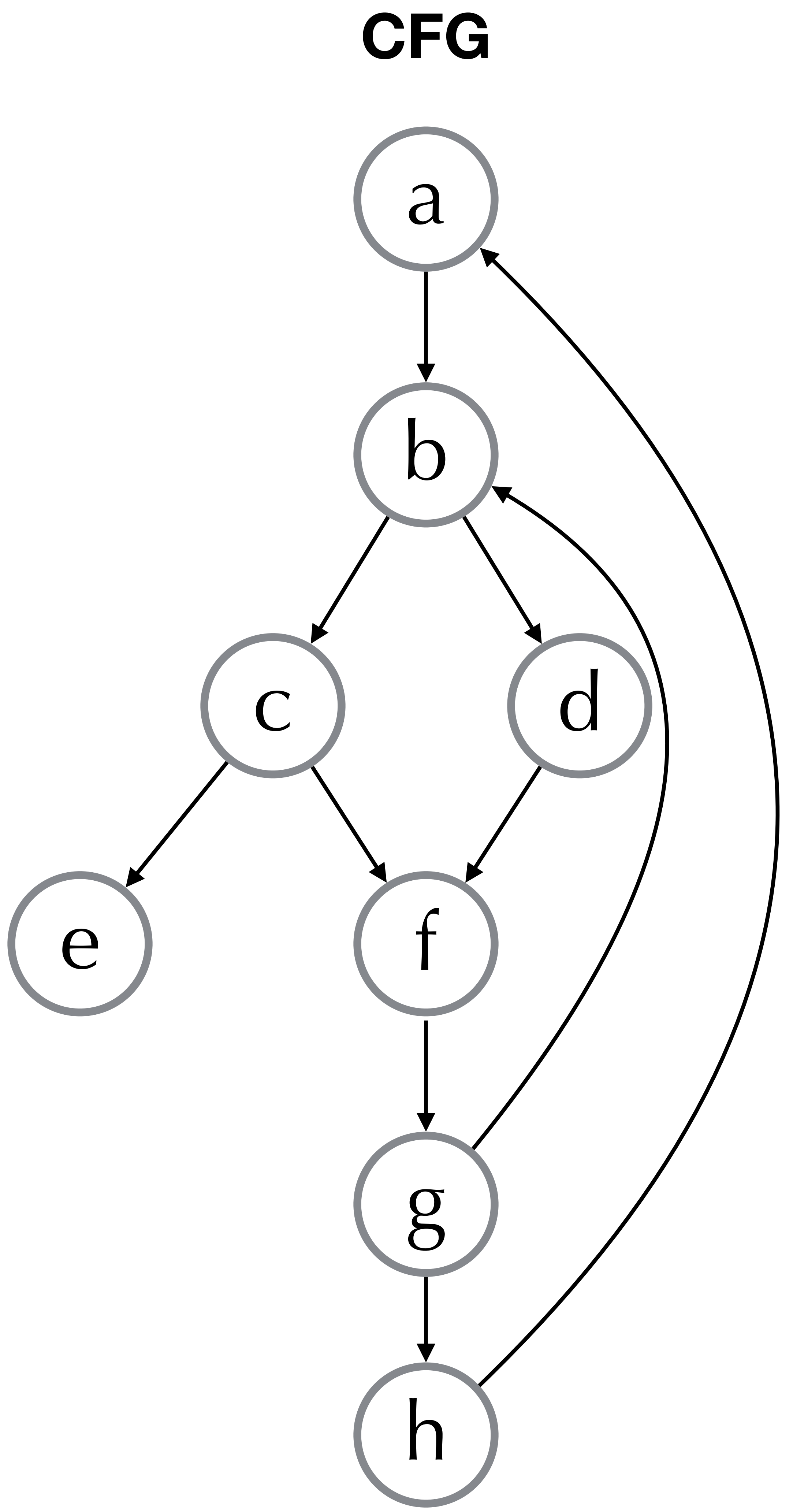
Calculating Dominators

- $D[n]$: 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

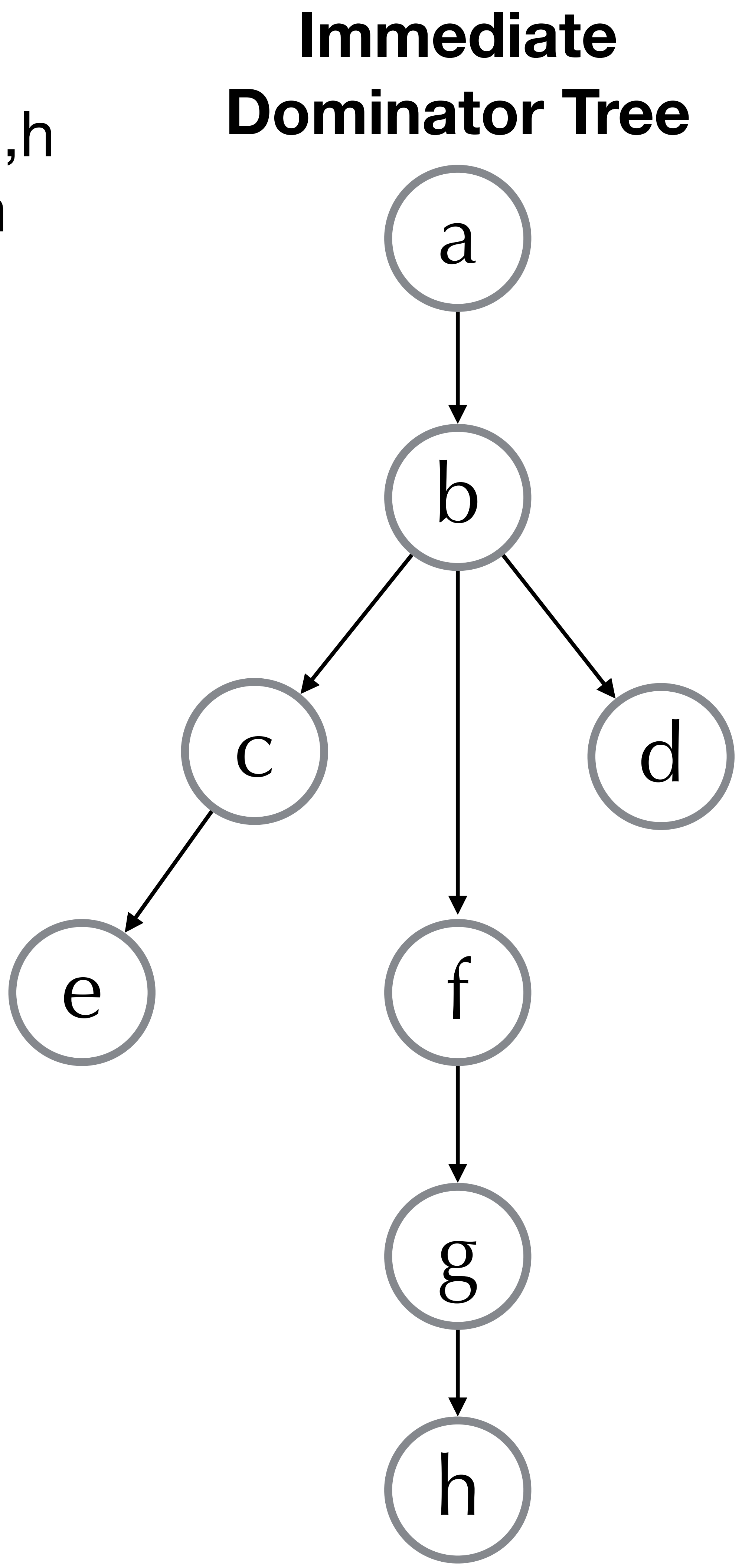
- Don’t actually need to keep set of all dominators for each node
- Instead, construct a **dominator tree**
  - Insight: if both *d* and *e* dominate *n*, then either *d* dominates *e* or vice versa
  - So that means that node *n* has a “closest” or **immediate dominator**

Example



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



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 Invariants

- An assignment `x := v`$_1$ `op 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:= v`$_1$ `op v`$_2$ 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

- 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:  ...
```

- 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];
}
```

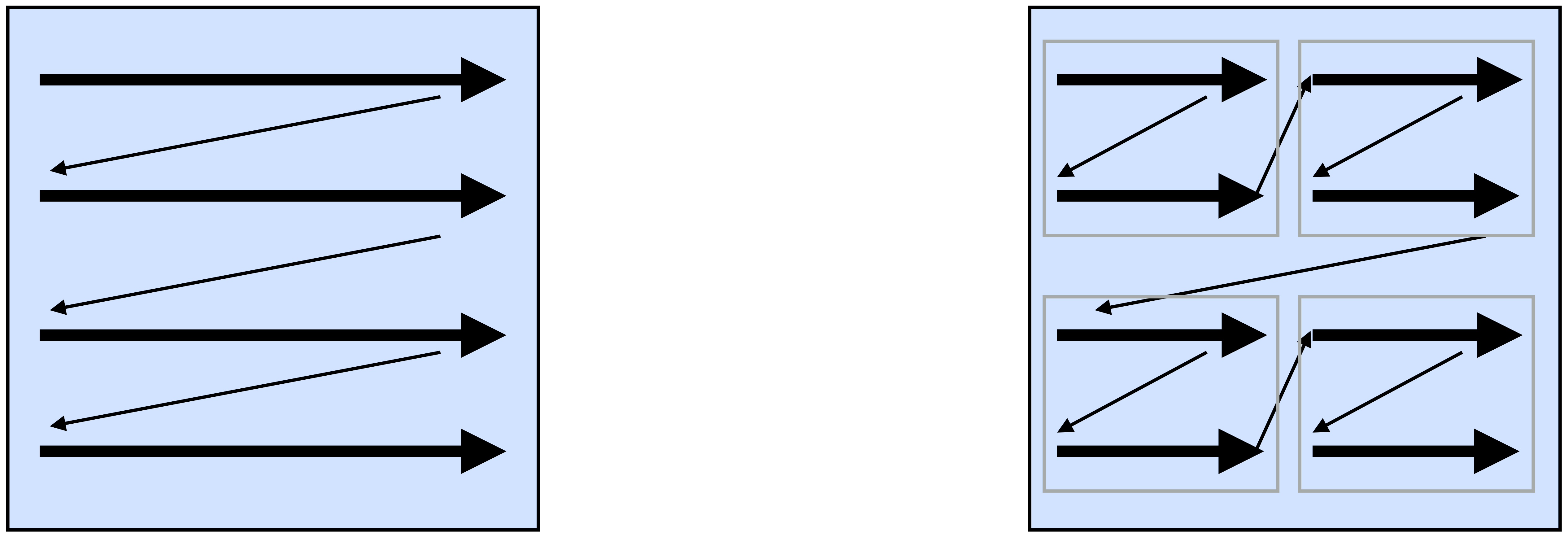
- After

```
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?