CSC D70: Compiler Optimization Register Allocation

Prof. Gennady Pekhimenko
University of Toronto

Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons

Register Allocation and Coalescing

- Introduction
- Abstraction and the Problem
- Algorithm
- Spilling
- Coalescing

Reading: ALSU 8.8.4

Motivation

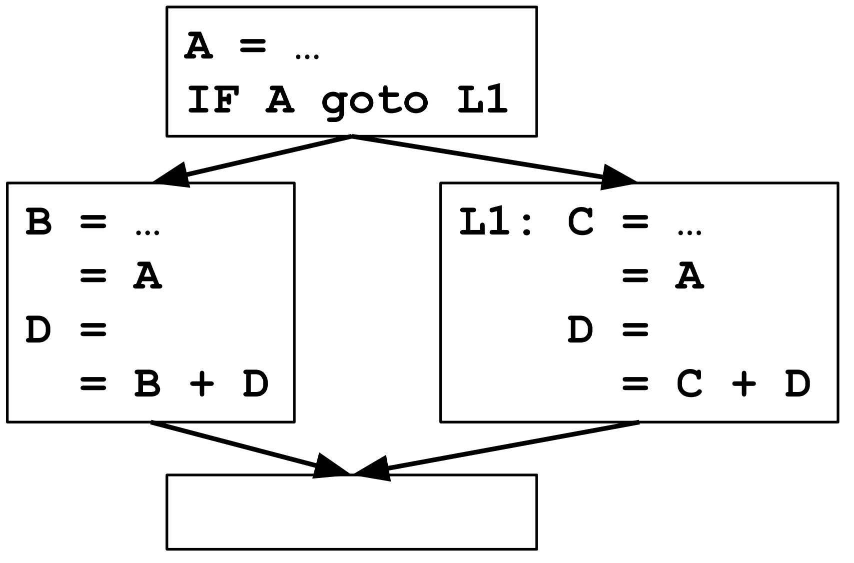
- **Problem**
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- **A very important optimization!**
  - Directly reduces running time
    - (memory access ⍰ register access)
  - Useful for other optimizations
    - e.g. CSE assumes old values are kept in registers.

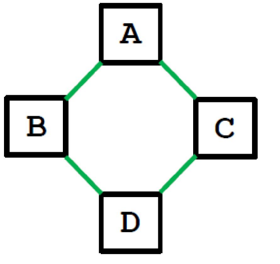
Goals

- Find an allocation for all pseudo-registers, if possible.

- If there are not enough registers in the machine, choose registers to spill to memory

Register Assignment Example





- Find an assignment (no spilling) with only 2 registers
  - A and D in one register, B and C in another one
- What assumptions?
  - After assignment, no use of A & (and only one of B and C used)

An Abstraction for Allocation & Assignment

- **Intuitively**
  - Two pseudo-registers **interfere** if at some point in the program they cannot both occupy the same register.
- **Interference graph**: an undirected graph, where
  - nodes = pseudo-registers
  - there is an edge between two nodes if their corresponding pseudo-registers interfere
- **What is not represented**
  - Extent of the interference between uses of different variables
  - Where in the program is the interference

Interfere many times vs. once

E.g., cold path vs. hot path

Register Allocation and Coloring

- A graph is **n-colorable** if:
  - every node in the graph can be colored with one of the n colors such that two adjacent nodes do not have the same color.
- Assigning n register (without spilling) = Coloring with n colors
  - assign a node to a register (color) such that no two adjacent nodes are assigned same registers (colors)
- Is spilling necessary? = Is the graph n-colorable?
- To determine if a graph is n-colorable is NP-complete, for n>2
  - Too expensive
  - Heuristics

Algorithm

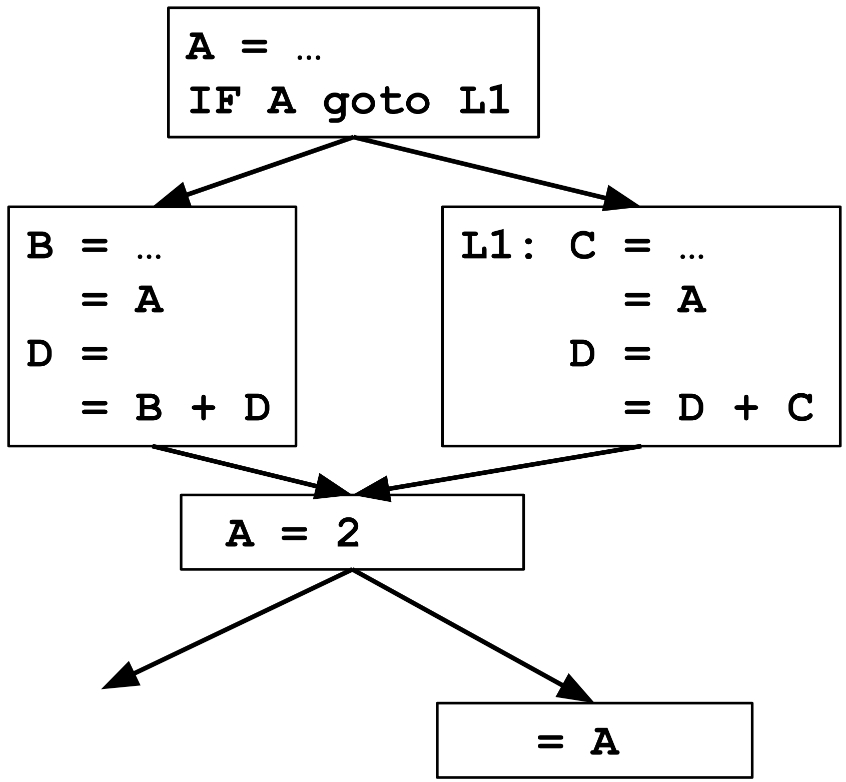
Step 1. Build an interference graph

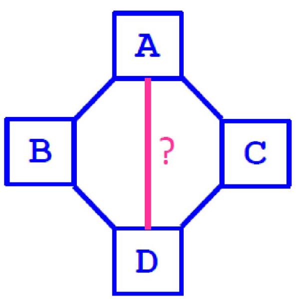
a. refining notion of a node
b. finding the edges

Step 2. Coloring

- use heuristics to try to find an n-coloring
  - Success:
    - colorable and we have an assignment
  - Failure:
    - graph not colorable, or
    - graph is colorable, but it is too expensive to color

Step 1a. Nodes in an Interference Graph





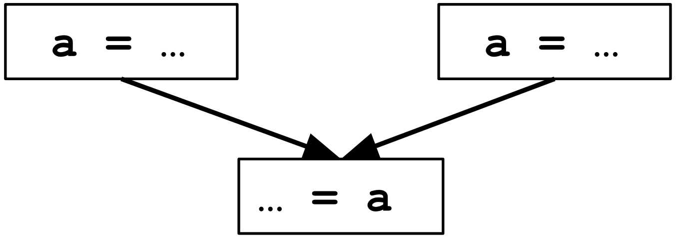
Interference Graph

Should we add A-D edge?

No, since new def of A

Live Ranges and Merged Live Ranges

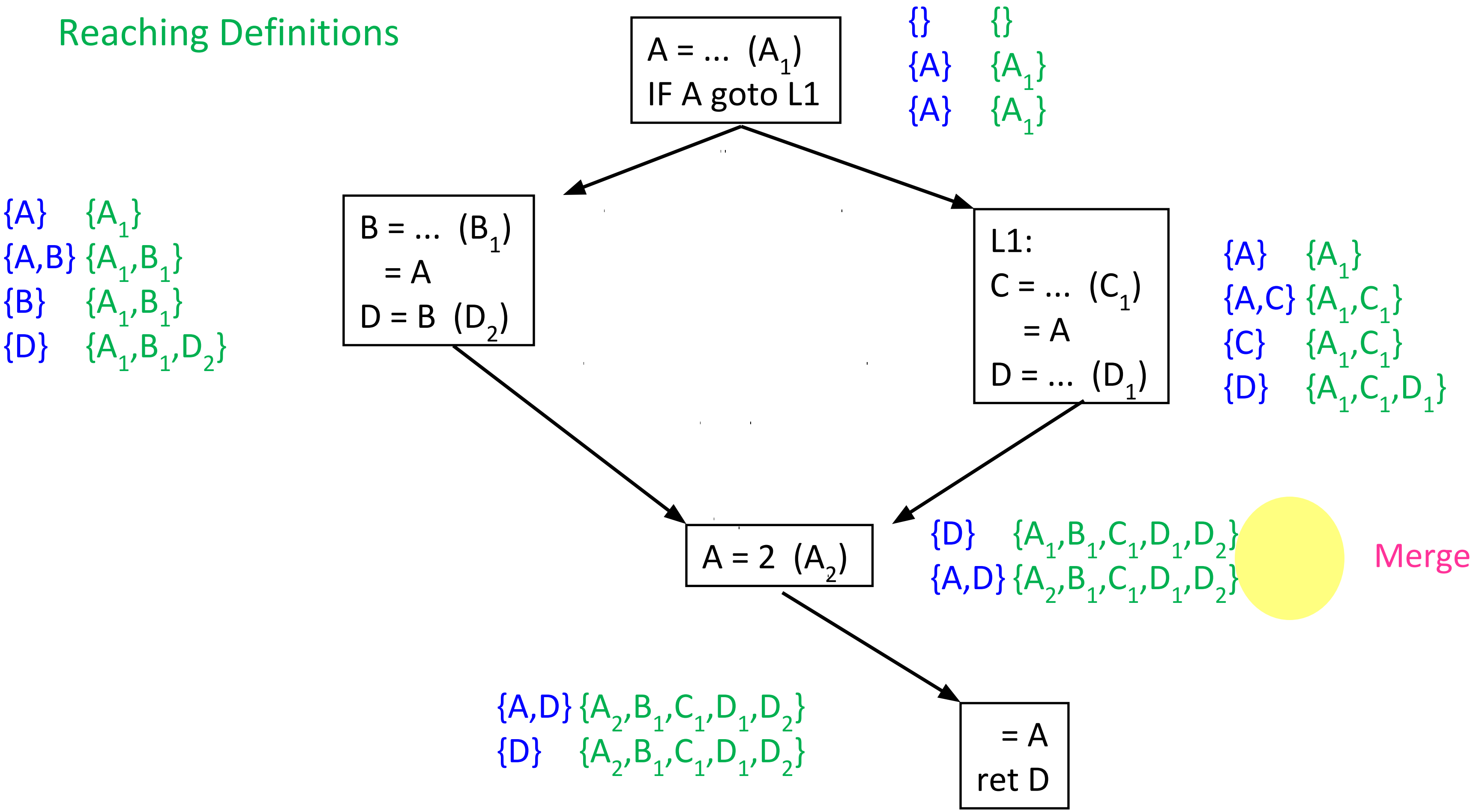
- **Motivation: to create an interference graph that is easier to color**
  - Eliminate interference in a variable's "dead" zones.
  - Increase flexibility in allocation:
    - can allocate same variable to different registers
- A **live range** consists of a definition and all the points in a program in which that definition is live.
  - How to compute a live range?
- Two overlapping live ranges for the **same** variable must be merged



Example (Revisited)

Live Variables

Reaching Definitions



Merging Live Ranges

- **Merging definitions into equivalence classes**
  - Start by putting each definition in a different equivalence class
  - Then, for each point in a program:
    - if (i) variable is live, and (ii) there are multiple reaching definitions for the variable, then:
      - merge the equivalence classes of all such definitions into one equivalence class
    - **Sounds familiar?**

- **From now on, refer to merged live ranges simply as live ranges**
  - merged live ranges are also known as “webs”

SSA Revisited: What Happens to Φ Functions

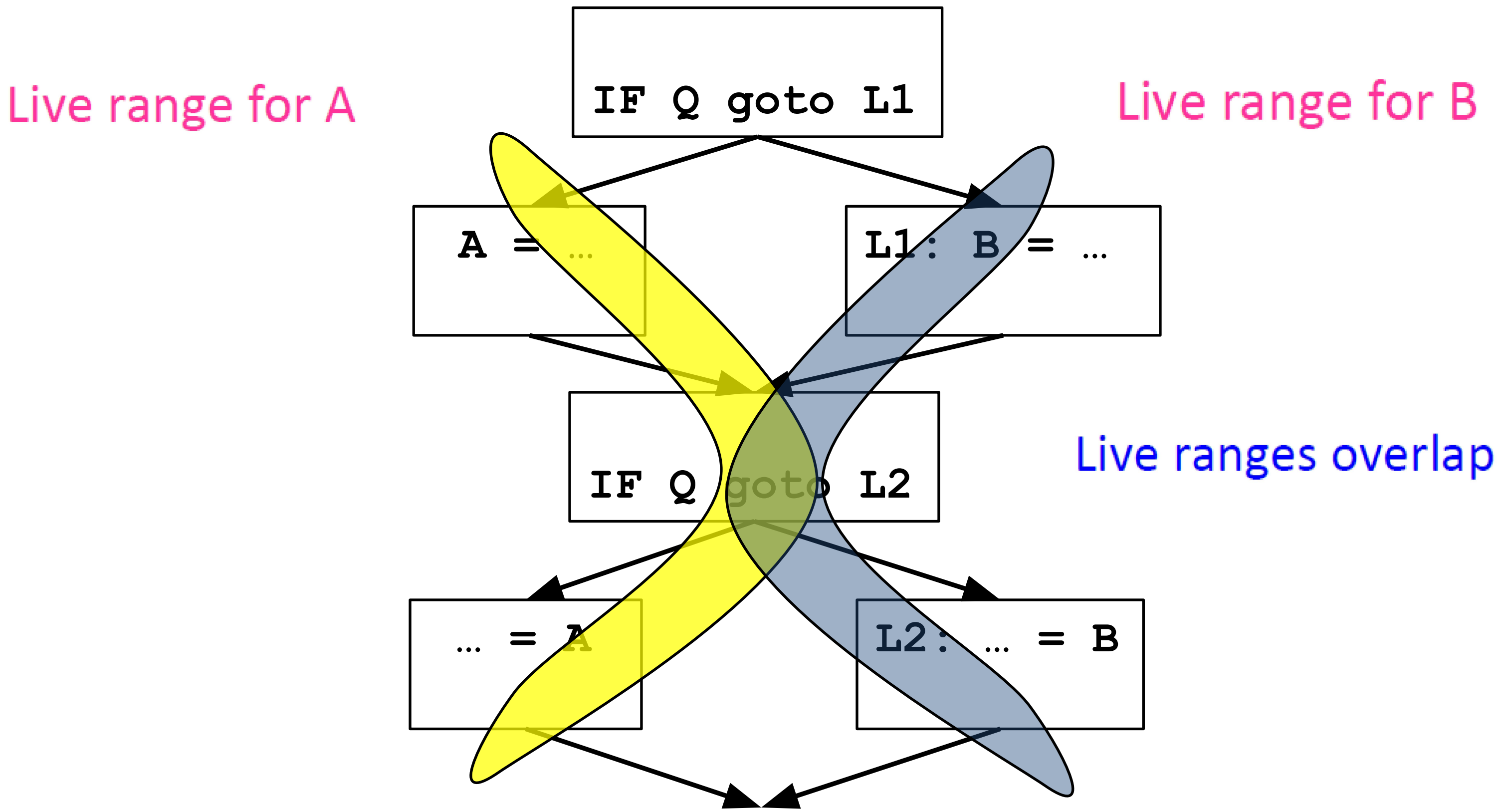
- Now we see why it is unnecessary to "implement" a Φ function
  - Φ functions and SSA variable renaming simply turn into merged live ranges
- When you encounter: $\mathbf{x_4 = \Phi(x_1, x_2, x_3)}$
  - merge $\mathbf{x_1}$, $\mathbf{x_2}$, $\mathbf{x_3}$, and $\mathbf{x_4}$ into the same live range
  - delete the $\mathbf{\Phi}$ function
- Now you have effectively converted back out of SSA form

Step 1b. Edges of Interference Graph

- **Intuitively**:
  - Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.
  - Algorithm:
    - At each point in the program:
      - enter an edge for every pair of live ranges at that point.

- **An optimized definition & algorithm for edges:**
  - Algorithm:
    - check for interference only at the start of each live range
  - Faster
  - Better quality

Live Range Example 2

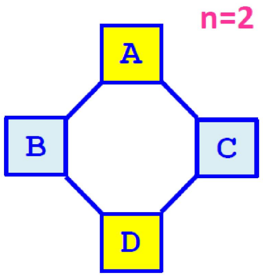


Because ranges overlap: Won't assign A and B to same register
(even though would have been ok: path sensitive vs. path insensitive analysis)

Step 2. Coloring

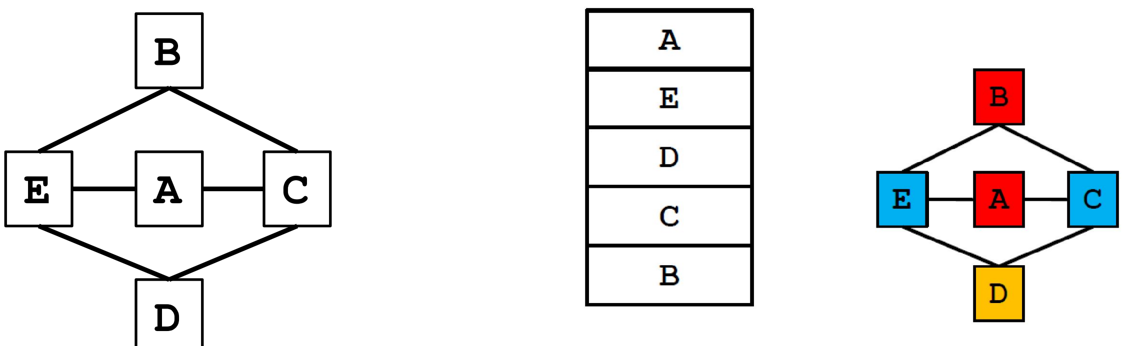
- **Reminder: coloring for n > 2 is NP-complete**

- **<u>Observations</u>:**
  - a node with degree < n ⇒
    - can always color it successfully, given its neighbors' colors
  - a node with degree = n ⇒
    - can only color if at least two neighbors share same color
  - a node with degree > n ⇒
    - maybe, not always



Coloring Algorithm

- Algorithm:
  - Iterate until stuck or done
    - Pick any node with degree < n
    - Remove the node and its edges from the graph
  - If done (no nodes left)
    - reverse process and add colors
- Example ($n = 3$):



- Note: degree of a node may drop in iteration
- Avoids making arbitrary decisions that make coloring fail

More details

- **Apply coloring heuristic**

  Build interference graph

  Iterate until there are no nodes left

      If there exists a node v with less than n neighbor

          push v on register allocation stack

      else

          return (coloring heuristics fail)

      remove v and its edges from graph

- **Assign registers**

  While stack is not empty

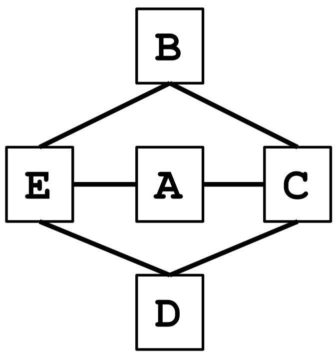
      Pop v from stack

      Reinsert v and its edges into the graph

      Assign v a color that differs from all its neighbors

What Does Coloring Accomplish?

- **Done**:
  - colorable, also obtained an assignment
- **Stuck**:
  - colorable or not?



Extending Coloring: Design Principles

- **A pseudo-register is**
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- **Objective function**
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- **Heuristics**
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

Chaitin: Coloring and Spilling

- **Identify spilling**

  Build interference graph
  Iterate until there are no nodes left
      If there exists a node v with less than n neighbor
          place v on stack to register allocate
      else
          v = node with highest degree-to-cost ratio
          mark v as spilled
      remove v and its edges from graph

- **Spilling may require use of registers; change interference graph**

  While there is spilling
  rebuild interference graph and perform step above

- **Assign registers**

  While stack is not empty
      Remove v from stack
      Reinsert v and its edges into the graph
      Assign v a color that differs from all its neighbors

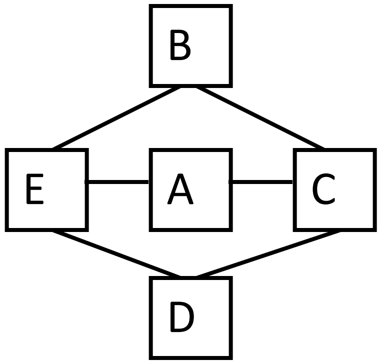
Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?

- One Heuristic:
  - spill cheapest live range (aka "web")
  - Cost = [(# defs & uses)*$10^{\text{loop-nest-depth}}$]/degree

Quality of Chaitin’s Algorithm

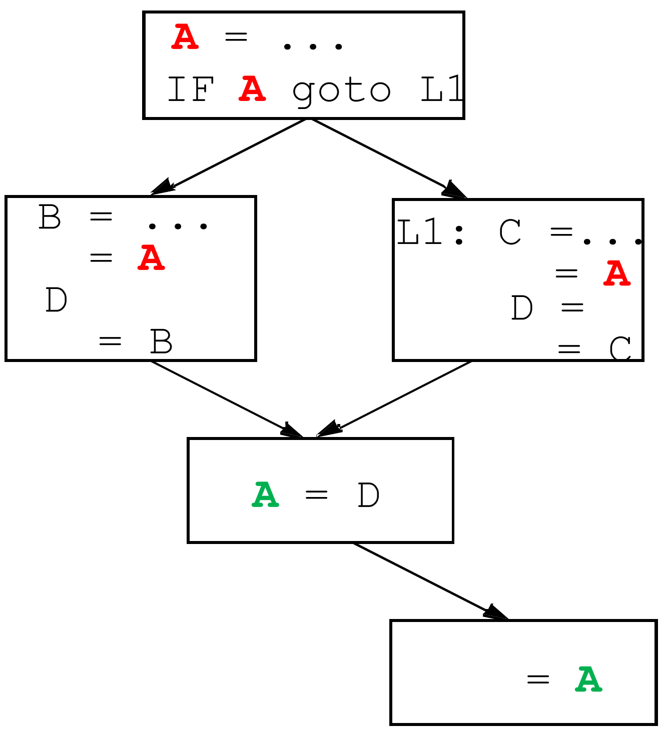
- Giving up too quickly
- N=2

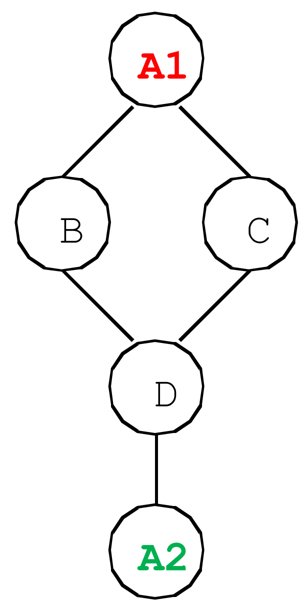


- An optimization: “Prioritize the coloring”
  - Still eliminate a node and its edges from graph
  - Do not commit to “spilling” just yet
  - Try to color again in assignment phase.

Splitting Live Ranges

- Recall: Split pseudo-registers into live ranges to create an interference graph that is easier to color
  - Eliminate interference in a variable's "dead" zones.
  - Increase flexibility in allocation:
    - can allocate same variable to different registers





Insight

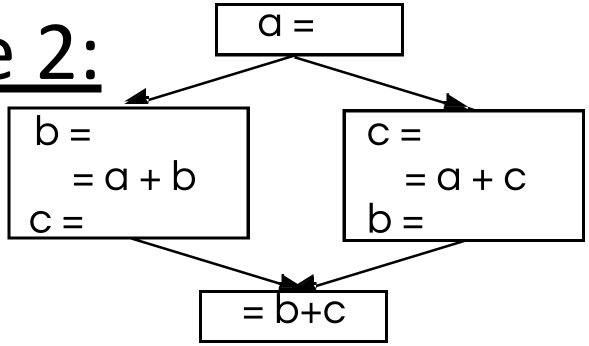
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's "nearly dead" zones.
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - # active live ranges at a program point can be > # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - # active live ranges cannot be > # registers

Examples

Example 1:

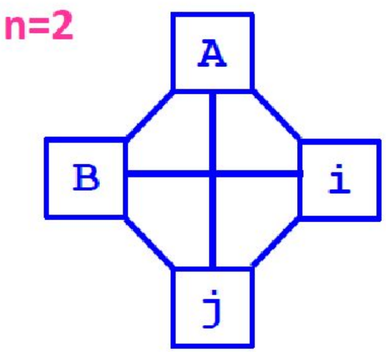
```
FOR i = 0 TO 10
FOR j = 0 TO 10000
    A = A + ...
 (does not use B)
FOR j = 0 TO 10000
    B = B + ...
 (does not use A)
```

Example 2:

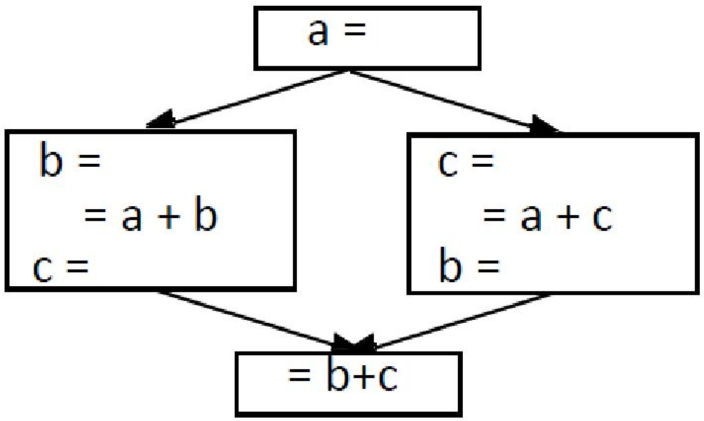


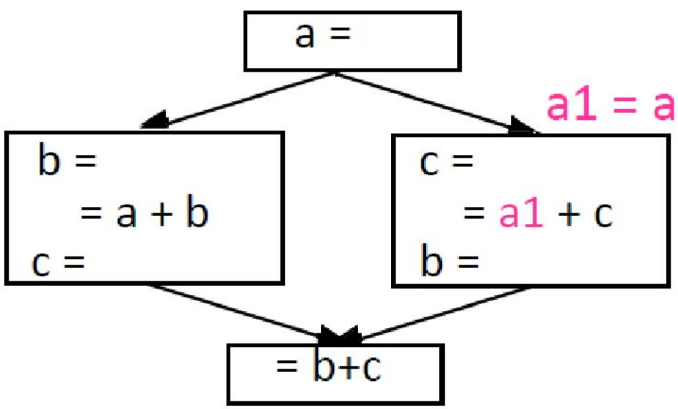
Example 1

```
                FOR i = 0 TO 10
        spill     FOR j = 0 TO 10000
          B           A = A + ...
spill                 (does not use B)
  i               FOR j = 0 TO 10000
        spill         B = B + ...
          A           (does not use A)
```

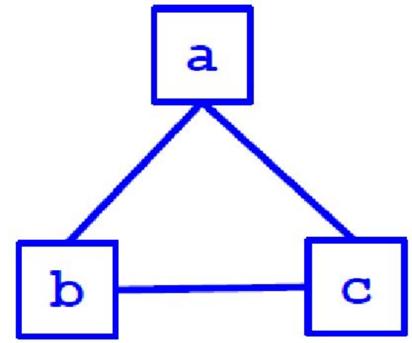


Example 2

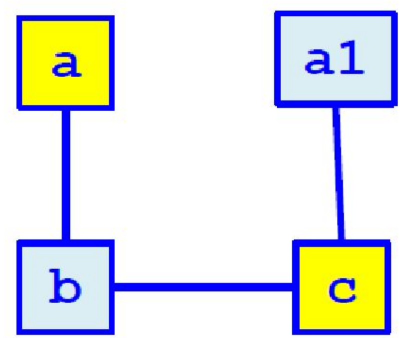




n=2



Can't 2-color



Can 2-color
("a" gets 2 regs)

Live Range Splitting

- When do we apply live range splitting?
- Which live range to split?
- Where should the live range be split?
- How to apply live-range splitting with coloring?
  - Advantage of coloring:
    - defers arbitrary assignment decisions until later
  - When coloring fails to proceed, may not need to split live range
    - degree of a node >= n does not mean that the graph definitely is not colorable
  - Interference graph does not capture positions of a live range

One Algorithm

- Observation: spilling is absolutely necessary if
  - number of live ranges active at a program point > n
- Apply live-range splitting before coloring
  - Identify a point where number of live ranges > n
  - For each live range active around that point:
    - find the outermost “block construct” that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

Summary

- **Problems:**
  - Given n registers in a machine, is spilling avoided?
  - Find an assignment for all pseudo-registers, whenever possible.
- **Solution:**
  - Abstraction: an **interference graph**
    - nodes: live ranges
    - edges: presence of live range at time of definition
  - Register Allocation and Assignment problems
    - equivalent to **n-colorability** of interference graph
      - ⍰ NP-complete
  - Heuristics to find an assignment for n colors
    - successful: colorable, and finds assignment
    - not successful: colorability unknown & no assignment

CSC D70: Compiler Optimization Register Coalescing

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons

Let's Focus on Copy Instructions

```
X = A + B;
…
Y = X;
…
Z = Y + 4;
```

```
X = A + B;
…
// deleted
…
Z = X + 4;
```

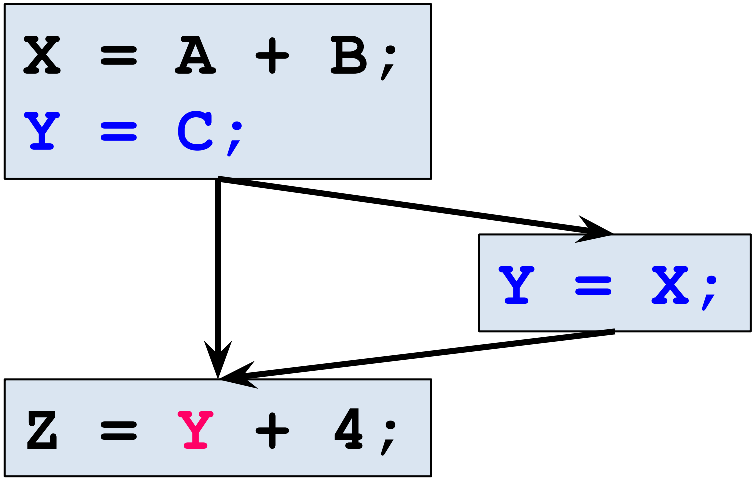
2. Dead Code Elimination

1. Copy Propagation

- Optimizations that help optimize away copy instructions:
  - Copy Propagation
  - Dead Code Elimination

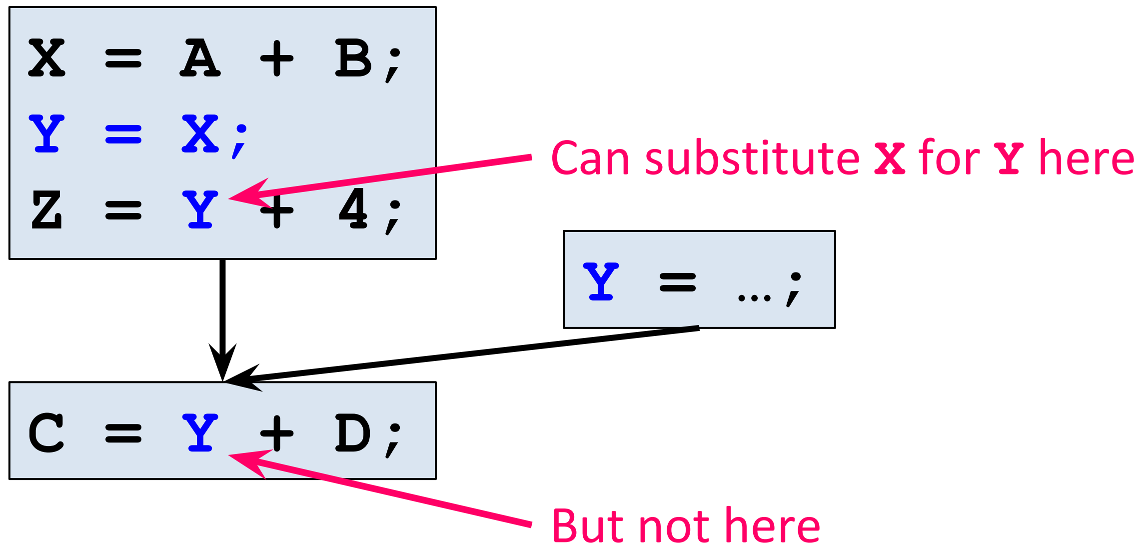
- Can all copy instructions be eliminated using this pair of optimizations?

Example Where Copy Propagation Fails



- Use of copy target has multiple (conflicting) reaching definitions

Another Example Where the Copy Instruction Remains



- Copy target (**Y**) still live even after some successful copy propagations

- <u>Bottom line:</u>
  - copy instructions may still exist when we perform register allocation

Copy Instructions and Register Allocation

- What clever thing might the register allocator do for copy instructions?

```
…
Y = X;
…
```

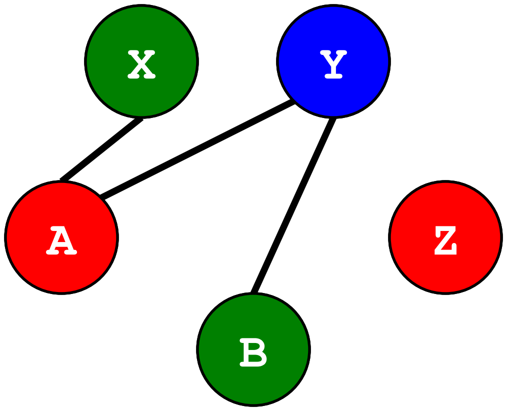
```
…
r7 = r7;
…
```

- If we can assign both the source and target of the copy to the same register:
  - then we don't need to perform the copy instruction at all!
  - the copy instruction can be removed from the code
    - even though the optimizer was unable to do this earlier

- <u>One way to do this:</u>
  - treat the copy source and target as the same node in the interference graph
    - then the coloring algorithm will naturally assign them to the same register
  - this is called "coalescing"

Simple Example: Without Coalescing

```
X = …;
A = 5;
Y = X;
B = A + 2;
Z = Y + B;
return Z;
```

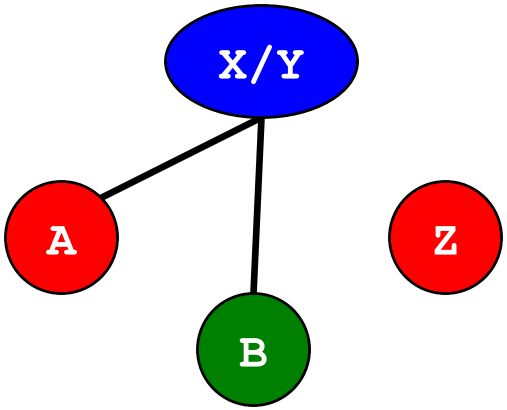


Valid coloring with 3 registers

- Without coalescing, **`X`** and **`Y`** can end up in different registers
  - cannot eliminate the copy instruction

Example Revisited: With Coalescing

```
X = …;
A = 5;
~~Y = X;~~
B = A + 2;
Z = Y + B;
return Z;
```

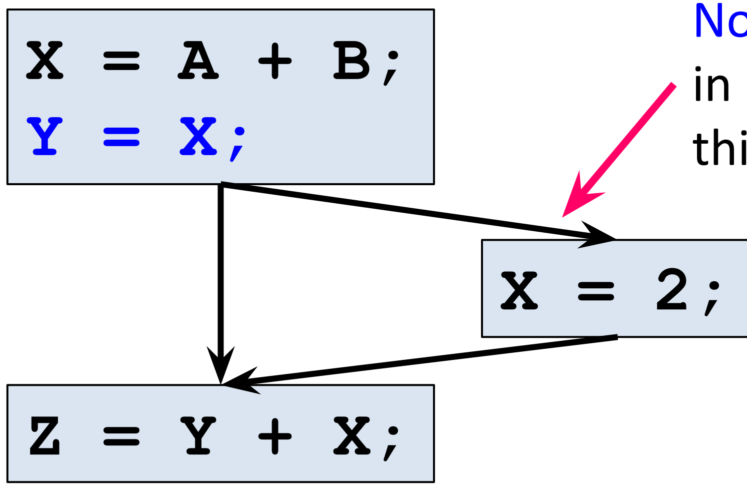


Valid coloring with 3 registers

- With coalescing, **X** and **Y** are now guaranteed to end up in the same register
  - the copy instruction can now be eliminated

- Great! So should we go ahead and do this for every copy instruction?

Should We Coalesce `X` and `Y` In This Case?



No! That would result in incorrect behavior if this branch is taken.

- It is legal to coalesce **`X`** and **`Y`** for a "**`Y = X`**" copy instruction iff:
  - initial definition of **`Y`**'s live range is this copy instruction, AND
  - the live ranges of **`X`** and **`Y`** do not interfere otherwise

- But just because it is legal doesn't mean that it is a good idea…

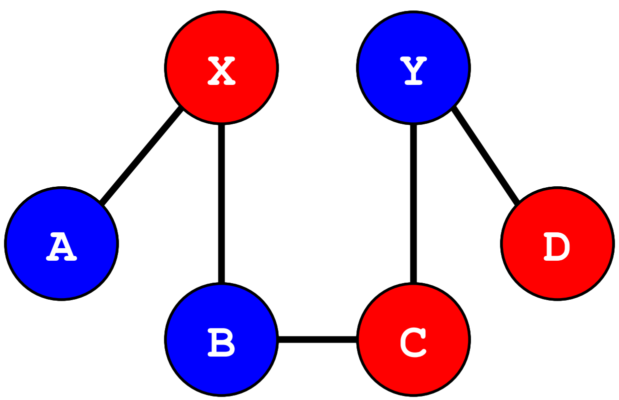
Why Coalescing May Be Undesirable

```
X = A + B;
…  // 100 instructions
Y = X;
…  // 100 instructions
Z = Y + 4;
```

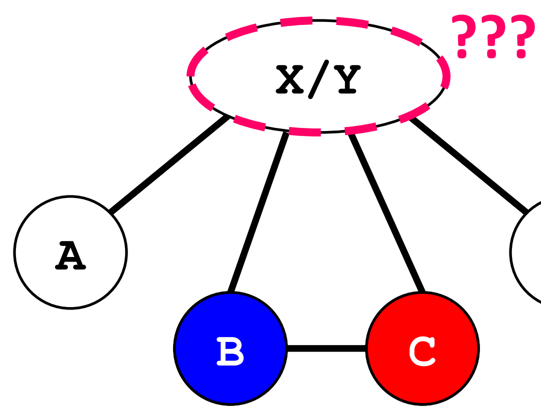
- What is the likely impact of coalescing **X** and **Y** on:
  - live range size(s)?
    - recall our discussion of live range splitting
  - colorability of the interference graph?
- Fundamentally, coalescing adds further constraints to the coloring problem
  - doesn't make coloring easier; may make it more difficult
- If we coalesce in this case, we may:
  - save a copy instruction, BUT
  - cause significant spilling overhead if we can no longer color the graph

When to Coalesce

- Goal when coalescing is legal:
  - coalesce *unless* it would make a colorable graph non-colorable
- The bad news:
  - predicting colorability is tricky!
    - it depends on the shape of the graph
    - graph coloring is NP-hard
- <u>Example</u>: assuming 2 registers, should we coalesce **X** and **Y**?



2-colorable

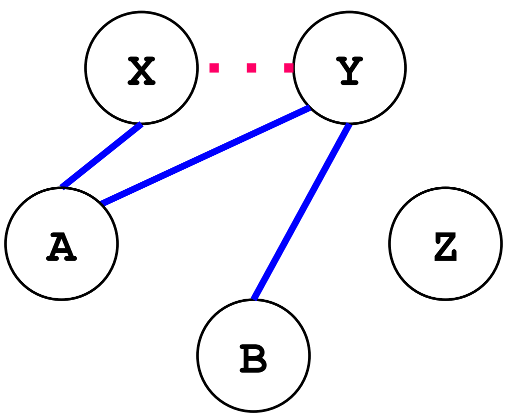


Not 2-colorable

Representing Coalescing Candidates in the Interference Graph

- To decide whether to coalesce, we augment the interference graph
- Coalescing candidates are represented by a new type of interference graph edge:
  - dotted lines: coalescing candidates
    - *try* to assign vertices the same color
      - (unless that is problematic, in which case they can be given different colors)
  - solid lines: interference
    - vertices *must* be assigned different colors

```
X = …;
A = 5;
Y = X;
B = A + 2;
Z = Y + B;
return Z;
```

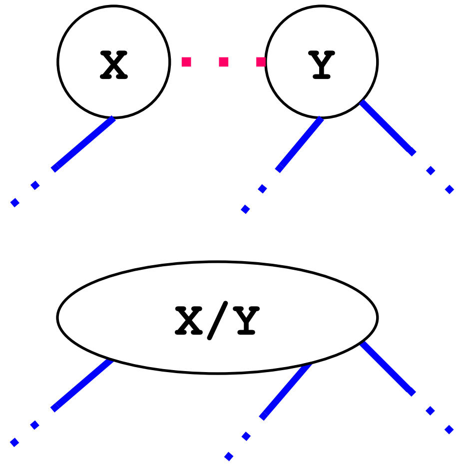


How Do We Know When Coalescing Will Not Cause Spilling?

- Key insight:
  - Recall from the coloring algorithm:
    - we can always successfully N-color a node if its degree is < N
- To ensure that coalescing does not cause spilling:
  - check that the degree < N invariant is still locally preserved after coalescing
    - if so, then coalescing won't cause the graph to become non-colorable
  - no need to inspect the entire interference graph, or do trial-and-error
- Note:
  - We do NOT need to determine whether the full graph is colorable or not
  - Just need to check that coalescing does not cause a colorable graph to become non-colorable

Simple and Safe Coalescing Algorithm

- We can safely coalesce nodes **X** and **Y** if (|**X**| + |**Y**|) < N
  - Note: |**X**| = degree of node **X** counting interference (not coalescing) edges
- Example:



(|**X**| + |**Y**|) = (1 + 2) = 3

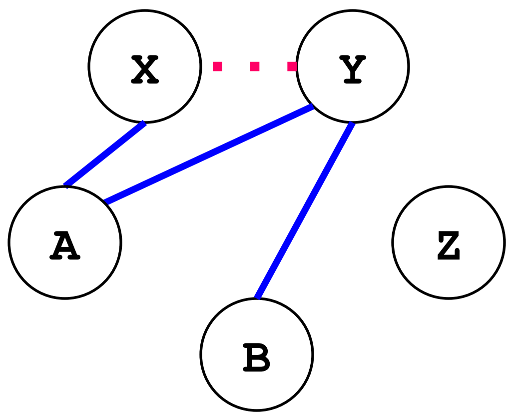
Degree of coalesced node can be no larger than 3

  - if N >= 4, it would always be safe to coalesce these two nodes
    - this cannot cause new spilling that would not have occurred with the original graph
  - if N < 4, it is unclear

How can we (safely) be more aggressive than this?

What About This Example?

- Assume N = 3
- Is it safe to coalesce **X** and **Y**?



(|**X**| + |**Y**|) = (1 + 2) = 3

(Not less than N)

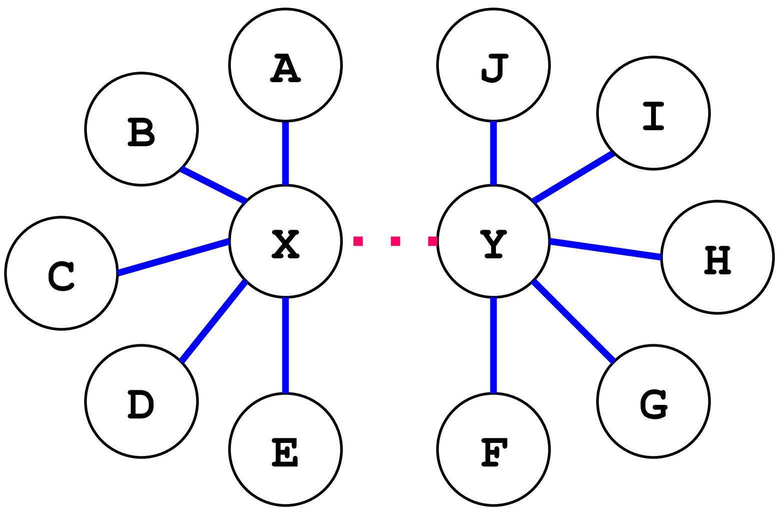
- <u>Notice</u>: **X** and **Y** share a common (interference) neighbor: node **A**
  - hence the degree of the coalesced **X/Y** node is actually 2 (not 3)
  - therefore coalescing **X** and **Y** *is* guaranteed to be safe when N = 3
- How can we adjust the algorithm to capture this?

Another Helpful Insight

- Colors are not assigned until nodes are popped off the stack
  - nodes with degree < N are pushed on the stack first
  - when a node is popped off the stack, we know that it can be colored
    - because the number of potentially conflicting neighbors must be < N
- Spilling only occurs if there is no node with degree < N to push on the stack

- <u>Example</u>: (N=2)

Another Helpful Insight



|**X**| = 5
|**Y**| = 5

2-colorable after coalescing **X** and **Y**?

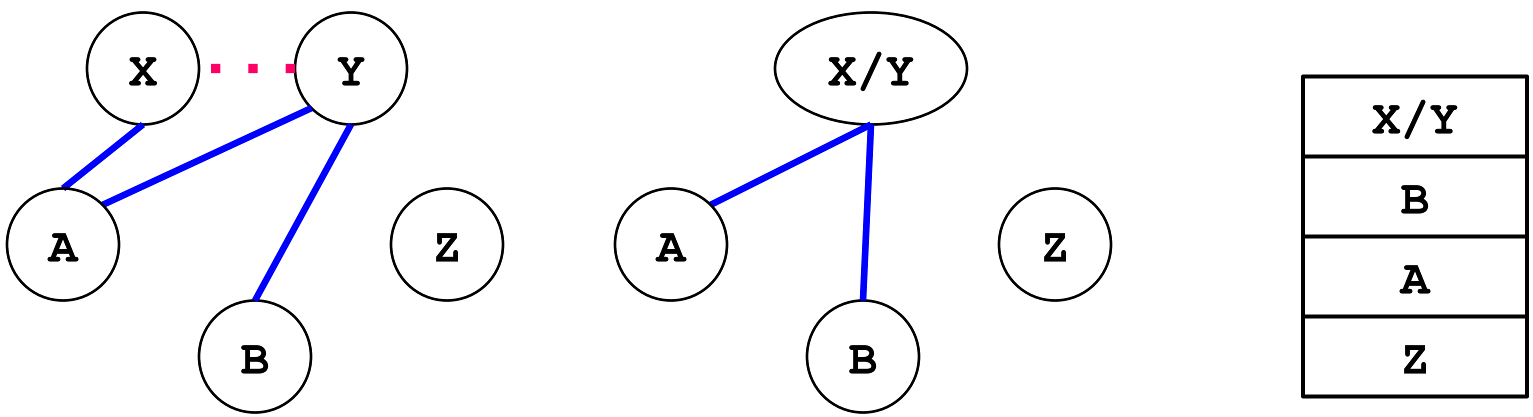
Building on This Insight

- When would coalescing cause the stack pushing (aka "simplification") to get stuck?
  1. coalesced node must have a degree >= N
     - otherwise, it can be pushed on the stack, and we are not stuck
  2. AND it must have at least N neighbors that each have a degree >= N
     - otherwise, all neighbors with degree < N can be pushed before this node
       - reducing this node's degree below N (and therefore we aren't stuck)

- To coalesce more aggressively (and safely), let's exploit this second requirement
  - which involves looking at the degree of a coalescing candidate's neighbors
    - not just the degree of the coalescing candidates themselves

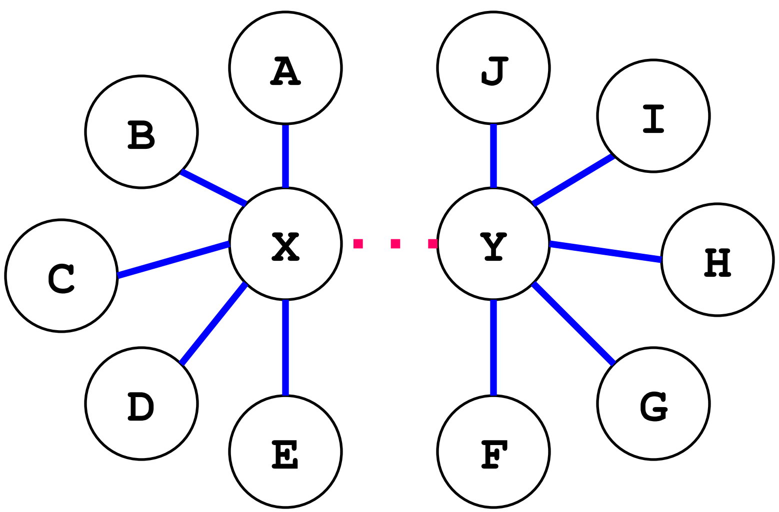
Briggs's Algorithm

- Nodes **X** and **Y** can be coalesced if:
  - (number of neighbors of **X/Y** with degree >= N) < N
- Works because:
  - all other neighbors can be pushed on the stack before this node,
  - and then its degree is < N, so then it can be pushed
  - <u>Example</u>: (N = 2)



Briggs's Algorithm

- Nodes **`X`** and **`Y`** can be coalesced if:
  - (number of neighbors of **`X/Y`** with
  - degree >= N) < N
- <u>More extreme example</u>: (N = 2)

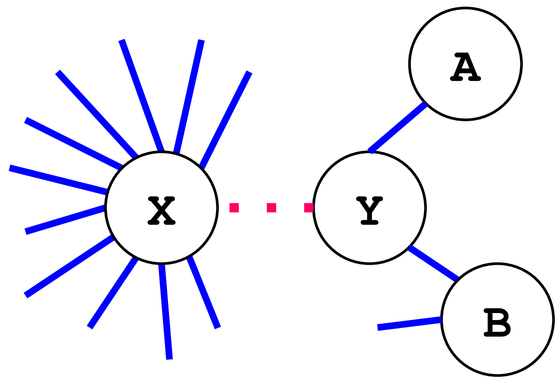


X/Y
J
I
H
G
F
E
D
C
B
A

George's Algorithm

Motivation:

- imagine that **`X`** has a very high degree, but **`Y`** has a much smaller degree
  - (perhaps because **`X`** has a large live range)



- With Briggs's algorithm, we would inspect all neighbors both **`X`** and **`Y`**
  - but **`X`** has a lot of neighbors!
- Can we get away with just inspecting the neighbors of **`Y`**?
  - showing that coalescing makes coloring no worse than it was given **`X`**?

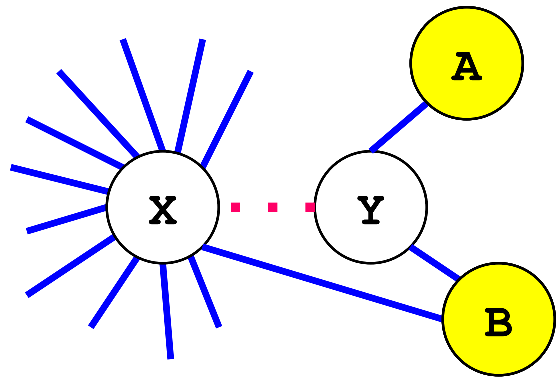
George's Algorithm

- Coalescing **X** and **Y** does no harm if:
  - foreach neighbor **T** of **Y**, either:
    1. degree of **T** is <N, or
    2. **T** interferes with **X**

 similar to Briggs: ***T*** will be pushed before ***X/Y***

 hence no change compared with coloring ***X***

- Example: (N=2)



Summary

- *Coalescing* can enable register allocation to eliminate copy instructions
  - if both source and target of copy can be allocated to the same register
- However, coalescing must be applied with care to avoid causing register spilling
- Augment the interference graph:
  - dotted lines for coalescing candidate edges
  - try to allocate to same register, unless this may cause spilling
- Coalescing Algorithms:
  - simply based upon degree of coalescing candidate nodes (**X** and **Y**)
  - Briggs's algorithm
    - look at degree of neighboring nodes of **X** and **Y**
  - George's algorithm
    - asymmetrical: look at neighbors of **Y** (degree and interference with **X**)

CSC D70: Compiler Optimization Register Allocation & Coalescing

Prof. Gennady Pekhimenko
University of Toronto

Winter 2020

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons