Lecture 23

Register Allocation:

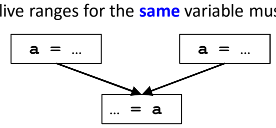
Coalescing

Review (2/17 lecture): Register Allocation without Spilling

- **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 [reviewed in next slide]
    - 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

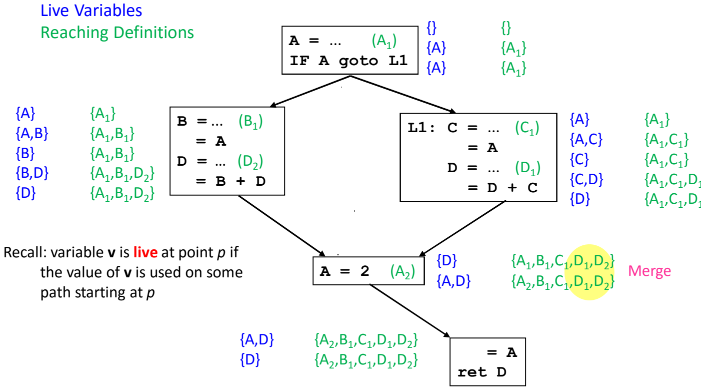
Review: Live Ranges and Merged Live Ranges

- **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 (e.g. end of an instruction) in which that definition is live.
  - How to compute a live range?
    - **live variables** & **reaching definitions**
- Two overlapping live ranges for the **same** variable must be merged



Review: Computing Live Ranges

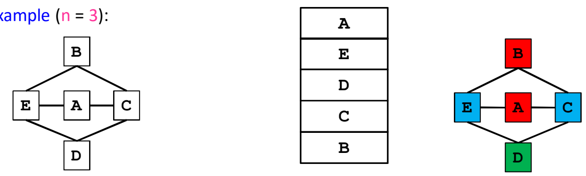
Live Variables
Reaching Definitions



Recall: variable **v** is live at point *p* if the value of **v** is used on some path starting at *p*

Review: Coloring Heuristic

- 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 (e.g., B, A, D different colors)

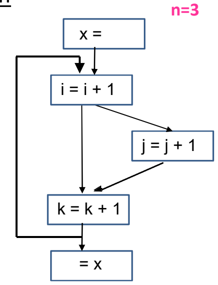
Review: Register Allocation with Spilling

- **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)
    - one estimate = [(# defs & uses)*$10^{\text{loop-nest-depth}}$]/degree
    - 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

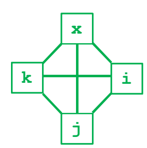
Review: A Spilling Algorithm

n=3

- 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



split x,
then can color



Register Coalescing Motivation: 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

```
X = A + B;
Y = C;
```

```
Y = X;
```

```
Z = Y + 4;
```

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

Another Example Where the Copy Instruction Remains

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

Can substitute **X** for **Y** here

```
Y = …;
```

```
C = Y + D;
```

But not here

- Copy target (**Y**) still live even after some successful copy propagations
- Bottom line:
  – 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?

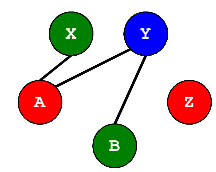
```
…
Y = X;
…
```

…
~~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
- One way to do this:
  – 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;
```

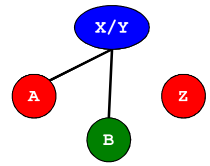


Valid coloring with 3 registers

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

Example Revisited: With Coalescing

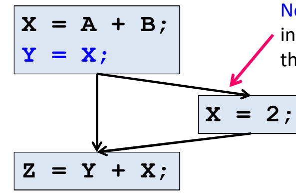
`X = …;`
`A = 5;`
~~`Y = X;`~~
`B = A + 2;`
`Z = Y + B;`
`return Z;`



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…

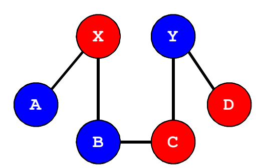
Why Coalescing May Be Undesirable, Even If Legal

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

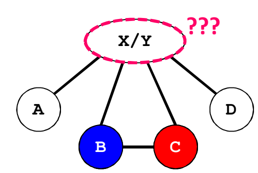
- What is the likely impact of coalescing **X** and **Y** on:
  - live range size(s)?
    - recall our discussion of live range splitting
  - colorability of the interference graph?
- Fundamentally, coalescing adds further constraints to the coloring problem
  - doesn't make coloring easier; may make 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
- Example: assuming 2 registers, should we coalesce **X** and **Y**?



2-colorable

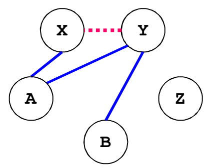


Not 2-colorable

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;
```

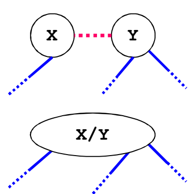


How Do We Know When Coalescing Will Not Cause Spilling?

- 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
- 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

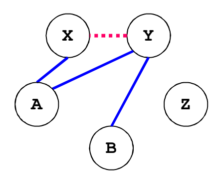
Degree of coalesced node can be no larger than 3

  - 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**?



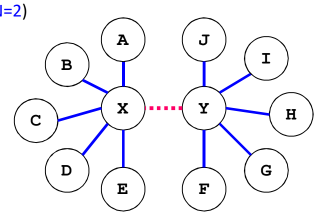
(|**X**| + |**Y**|) = (1 + 2) = 3
(Not less than N)

- Notice: **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

- Example: (N=2)



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

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

Yes: X/Y gets 1 color,
A-J get 1 color

Building on This Insight

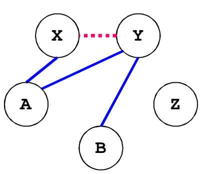
- When would coalescing cause the stack pushing (aka "simplification") to get stuck?
  1. coalesced node must have a degree >= N
     - otherwise, it can be pushed on the stack, and we are not stuck
  2. AND it must have at least N neighbors that each have a degree >= N
     - otherwise, all neighbors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

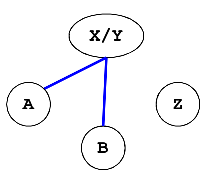
- To coalesce more aggressively (and safely), let's exploit this second requirement
  - which involves looking at the degree of a coalescing candidate's neighbors
    - not just the degree of the coalescing candidates themselves

Briggs' 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

- Example: (N = 2)



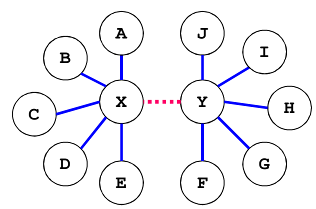


X/Y
B
A
Z

Briggs's Algorithm

- Nodes **X** and **Y** can be coalesced if:
  - (number of neighbors of **X**/**Y** with degree >= N) < N

- More extreme example: (N = 2)

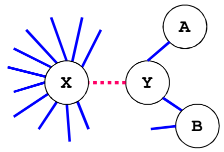


X/Y
J
I
H
G
F
E
D
C
B
A

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)

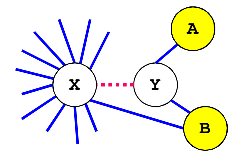


- With Briggs's algorithm, we would inspect all neighbors both **X** and **Y**
  - but **X** has a lot of neighbors!
- Can we get away with just inspecting the neighbors of **Y**?
  - showing that coalescing makes coloring no worse than it was given **X**?

George's Algorithm

- Coalescing **X** and **Y** does no harm if:
  - foreach neighbor **T** of **Y**, either:
    1. degree of **T** is <N, or ← *similar to Briggs: **T** will be pushed before **X/Y***
    2. **T** interferes with **X** ← *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
- Three Coalescing Algorithms:
  - Simplest: based solely on 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 lower degree node **Y** (degree and interference with **X**)

Wednesday's Class

- Thread-level Speculation