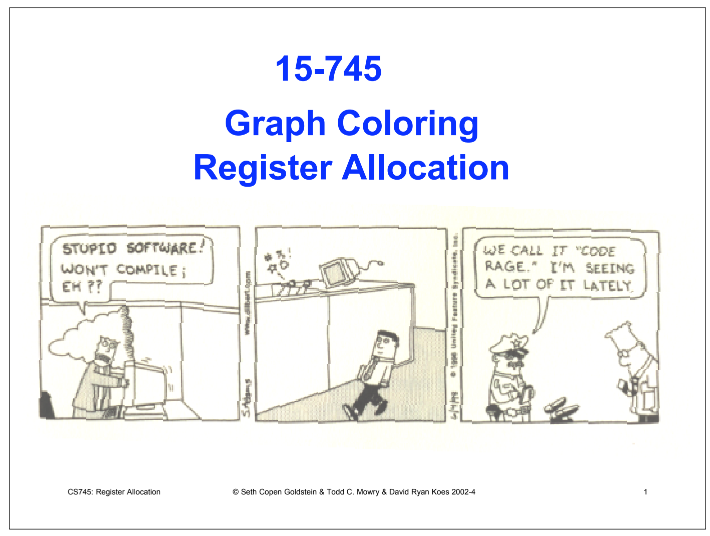


Intro to Global Register Allocation

Problem:

- **Allocation of variables (pseudo-registers) to hardware registers in a procedure**

One of the most important optimizations

- **Memory accesses are more costly than register accesses**
  - True even with caches
  - True even with CISC architectures
- **Important for other optimizations**
  - E.g., CSE assumes old values are kept in registers
- **When it does not work well, the performance impact is noticeable.**

Terminology

Allocation

- **decision to keep a pseudo-register in a hardware register**
- **prior to register allocation, we assume an infinite set of registers**
  - (aka “temps” or “pseudo-registers” or (bad) “variables”).

Spilling

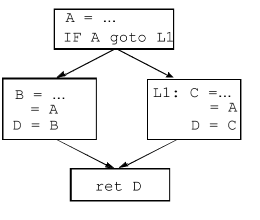
- **when allocation fails...**
- **a pseudo-register is spilled to memory, if not kept in a hardware register**

Assignment

- **decision to keep a pseudo-register in a *specific* hardware register**

What are the Problems?

- For this example:
  - What is the minimum number of registers needed to avoid spilling?
  - Given *n* registers in a machine, is spilling necessary?
  - Find an assignment for all pseudo-registers, if possible.
  - If there are not enough registers in the machine, how do we spill to memory?



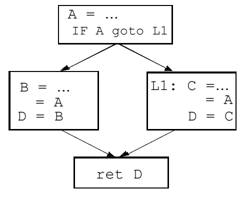
Abstraction for Reg Alloc & Assignment

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



Register Allocation and Coloring

- **A graph is *n-colorable* if every node in the graph can be colored with one of n colors such that two adjacent nodes do not have the same color.**
- Assigning n registers (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**

Simple Algorithm

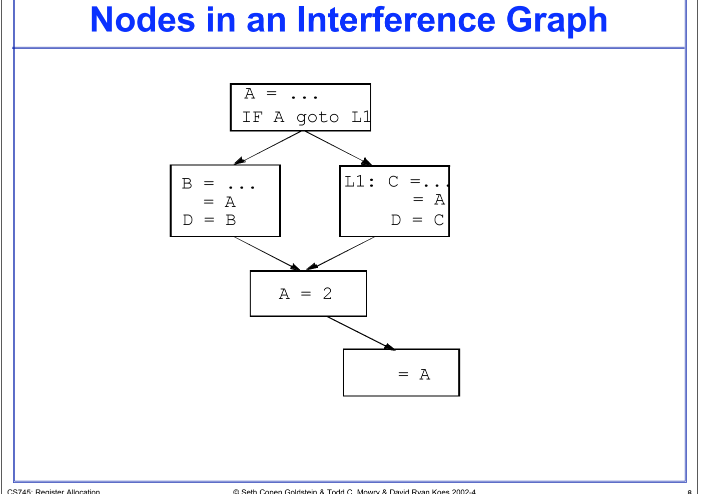
Build an interference graph

- refining notion of a node
- finding the edges

Coloring

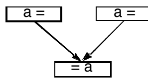
- use heuristics to try to find an n-coloring
  - **Success** ⇒ colorable and we have an assignment
  - **Failure** ⇒ graph not colorable, or graph is colorable, but we couldn't find a coloring

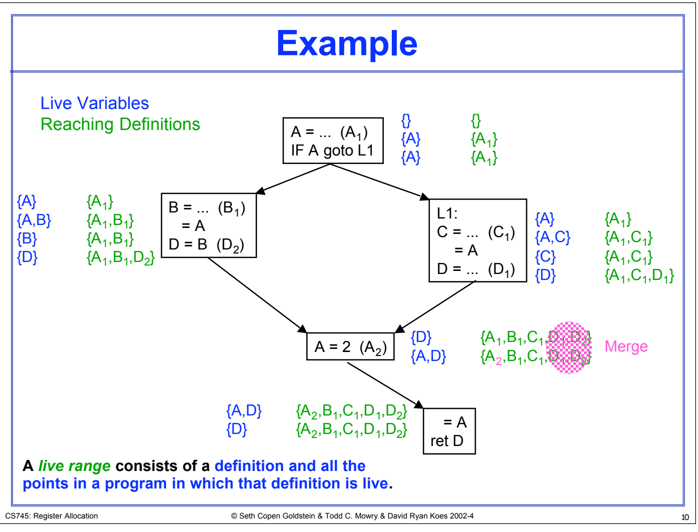
Nodes in an Interference Graph



Live Ranges & 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?
- **Two overlapping live ranges for the same variable must be merged**

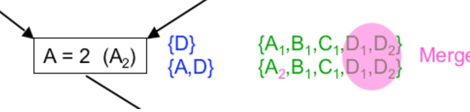




Merging Live Ranges

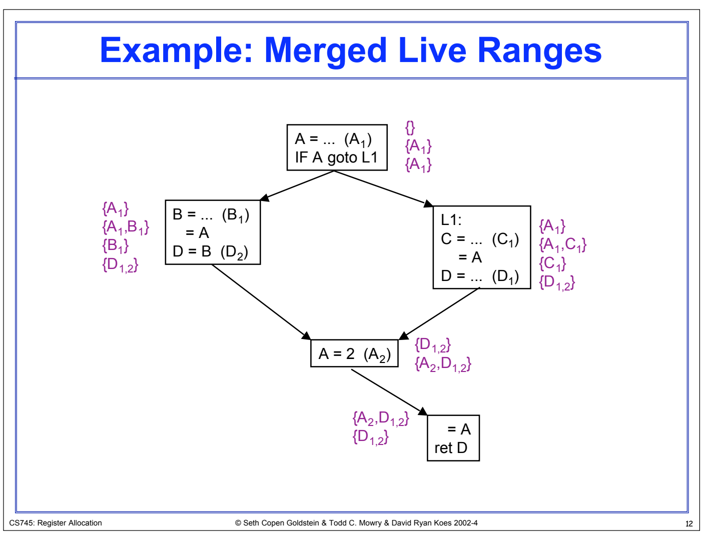
Merging definitions into equivalence classes:

- **Start by putting each definition in a different equivalence class**
- **For each point in a program**
  - if variable is live, and there are multiple reaching definitions for the variable
  - merge the equivalence classes of all such definitions into a one equivalence class



From now on, refer to merged live ranges simply as live range

- Merged live ranges are also known as "webs"



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 program, enter an edge for every pair of live ranges at that point

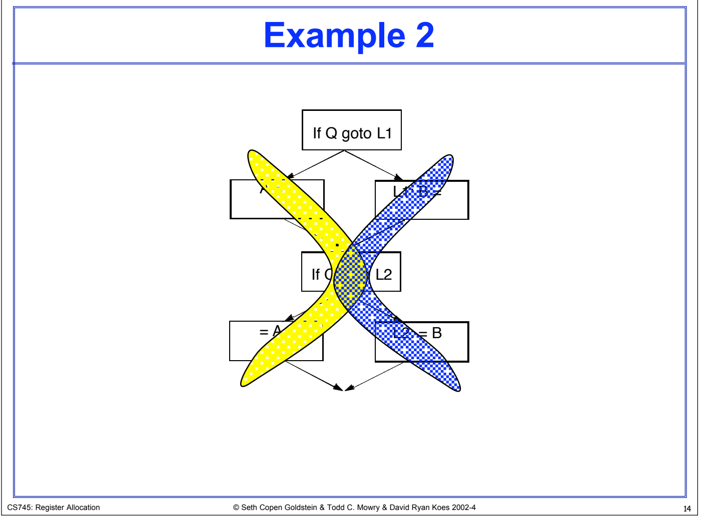
An optimized definition & algorithm for edges:

```
For each defining inst i
        Let x be live range of definition at inst i
        For each live range y present at end of inst i
                insert an edge between x and y
```

- Faster
- Better quality?

A = 2 (A_2) $\{D_{1,2}\}$ $\{A_2, D_{1,2}\}$ → Edge between A_2 and $D_{1,2}$

Example 2



Example: Interference Graph

A_1 — B

A_1 — C

A_2 — D $\{D_1, D_2\}$

So was it worth it to split the live ranges?

Coloring

- **Reminder:** coloring for n > 2 is NP-complete*
- **Observations**
  - a node with degree < n ⇒
    - ➢ can always color it successfully, given its neighbors' colors
  - a node with degree = n ⇒
  - a node with degree > n ⇒

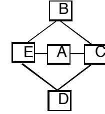
* [1] H. Bodlaender, J. Gustedt, and J. A. Telle, "Linear-time register allocation for a fixed number of registers," in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pp. 574–583, Society for Industrial and Applied Mathematics, 1998.
[2] S. Kannan and T. Proebsting, "Register allocation in structured programs," in *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pp. 360–368, Society for Industrial and Applied Mathematics, 1995.
[3] M. Thorup, "All structured programs have small tree width and good register allocation," *Inf. Comput.*, vol. 142, no. 2, pp. 159–181, 1998.

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

© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4

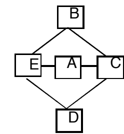
What Does Coloring Accomplish?

Done:

- **colorable**
- **also obtained an assignment (colors correspond to registers)**

Stuck (n = 2):

- **colorable or not?**



- **One solution: optimistically remove nodes and hope we get lucky…**

© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4

Checkpoint

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: (merged) live ranges
  - edges: presence of live range at time of definition
- **Register Allocation and Assignment problems**
  = n-colorability of interference graph
  ⇒ NP-complete
- **Heuristics to find an assignment for n colors**
  - successful: colorable, and finds assignment
  - unsuccessful: colorability unknown & no assignment

© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4

Discussion

What about when we can't k-color?

- **spill to memory**

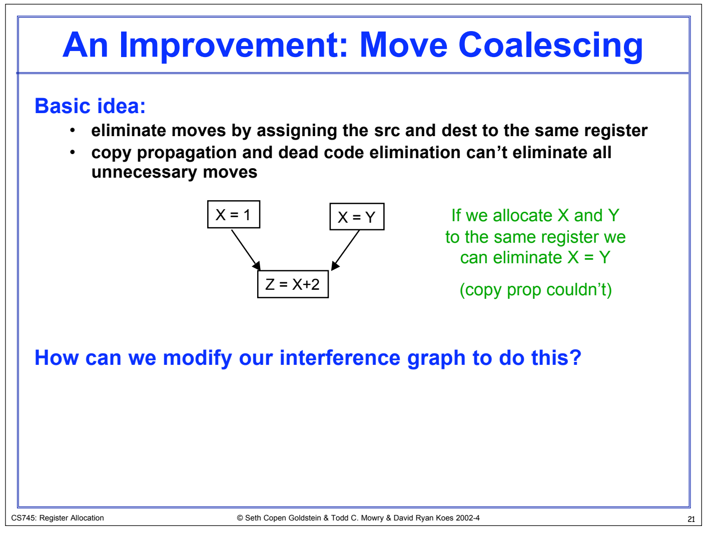
Is the minimum coloring always what we want?

- **Hint: no**

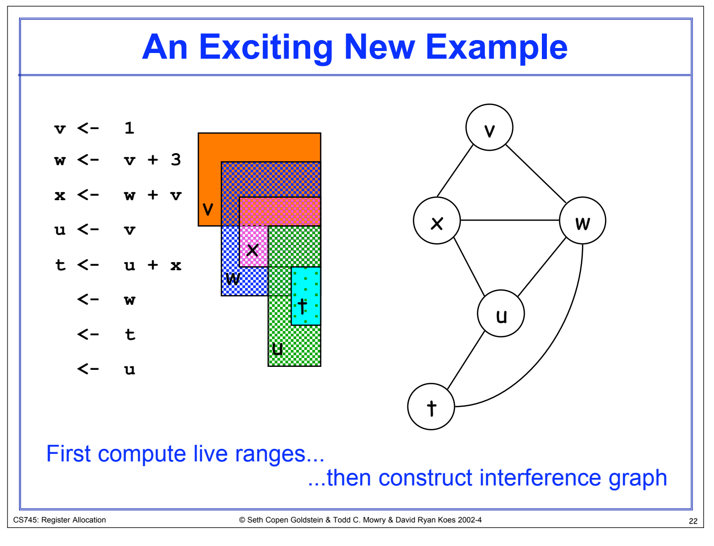
What about architecture strangeness?

- **subword registers (x86, 68k, ColdFire…)**
- **register pairing (HP PA-RISC, SPARC, x86)**
- **register classes (x86, 68k, ColdFire…)**

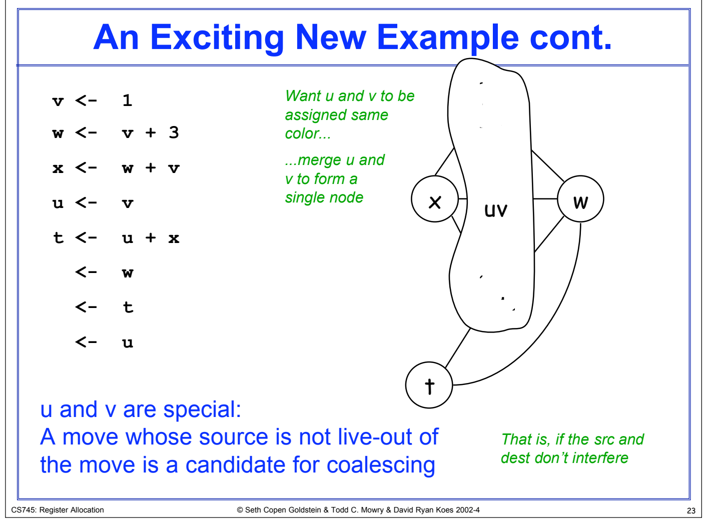
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4



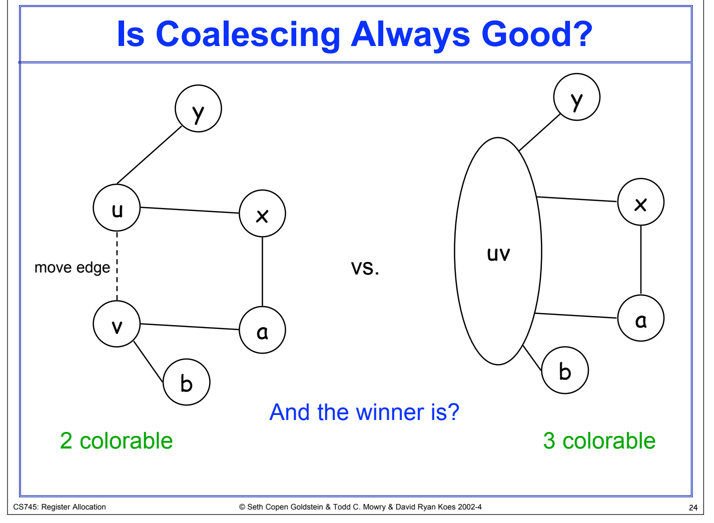
An Improvement: Move Coalescing
Basic idea:
• eliminate moves by assigning the src and dest to the same register
• copy propagation and dead code elimination can't eliminate all unnecessary moves
X = 1
X = Y
Z = X+2
If we allocate X and Y to the same register we can eliminate X = Y
(copy prop couldn't)
How can we modify our interference graph to do this?
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
21



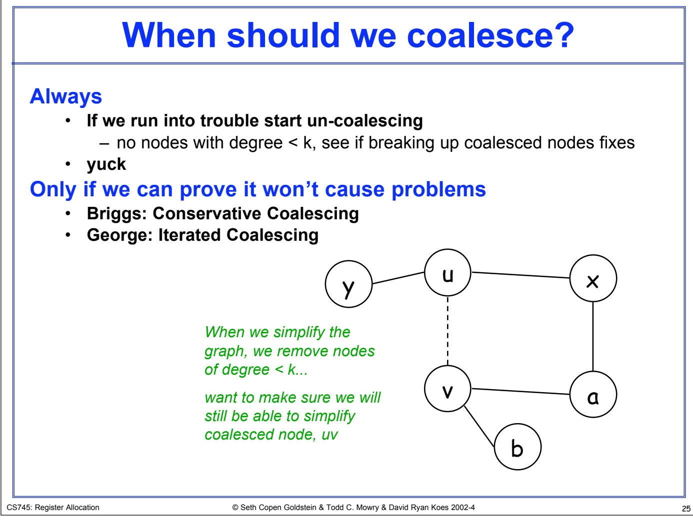
An Exciting New Example
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
<- w
<- t
<- u
v
w
x
t
u
First compute live ranges...
...then construct interference graph
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
22

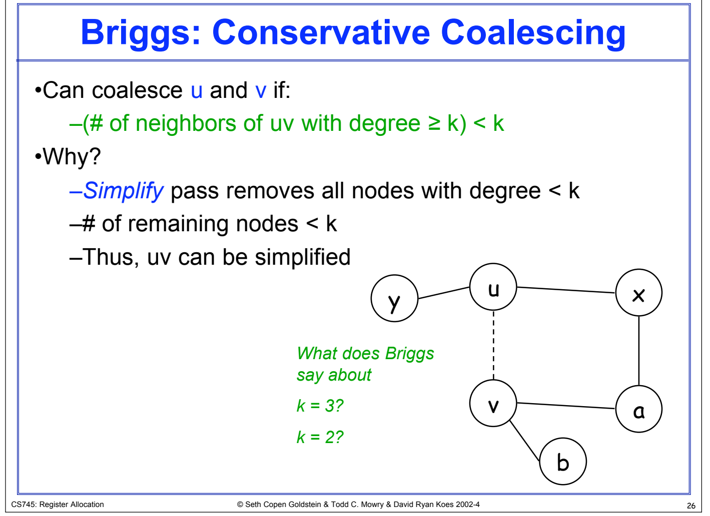


An Exciting New Example cont.
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
<- w
<- t
<- u
Want u and v to be assigned same color...
...merge u and v to form a single node
x
uv
w
t
u and v are special:
A move whose source is not live-out of the move is a candidate for coalescing
That is, if the src and dest don't interfere
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
23



Is Coalescing Always Good?
y
u
x
move edge
v
a
b
vs.
uv
2 colorable
And the winner is?
3 colorable
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
24





George: Iterated Coalescing

Can coalesce u and v if

foreach neighbor t of u

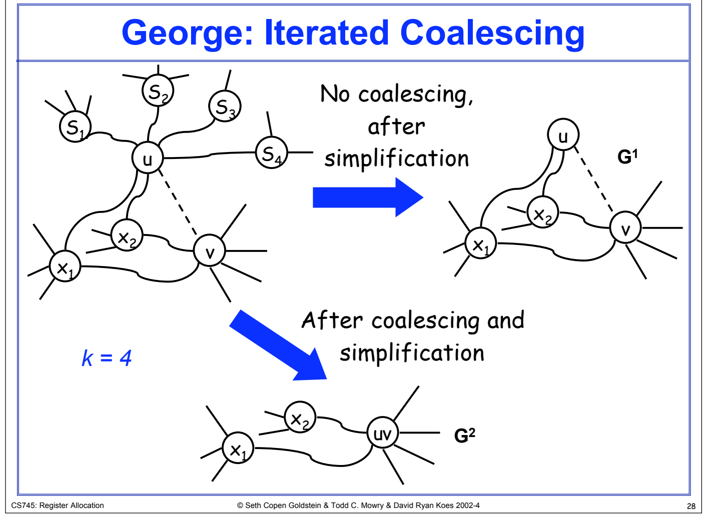
- t interferes with v, or, *doesn't change degree*
- degree of t < k *removed by simplification*

Resulting node uv will (after simplification) have degree equal to degree of v

Why?

- **let S be set of neighbors of u with degree < k**
- **If no coalescing, simplify removes all nodes in S, call that graph G^1**
- **If we coalesce we can still remove all nodes in S, call that graph G^2**
- **G^2 is a subgraph of G^1**

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 27



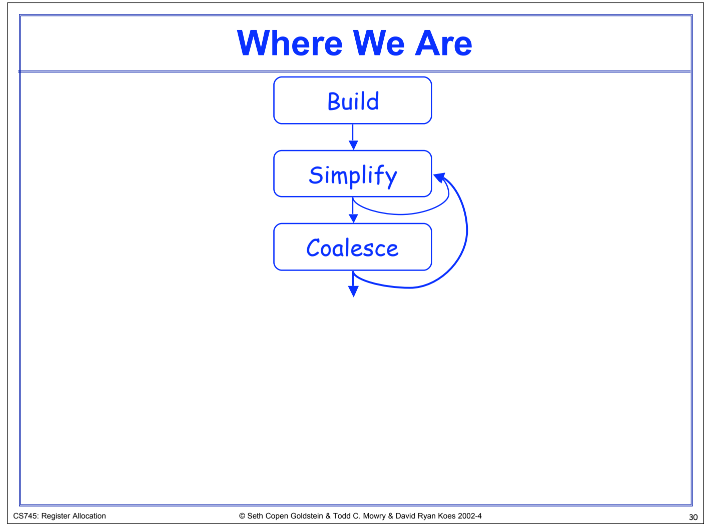
Why Two Methods?

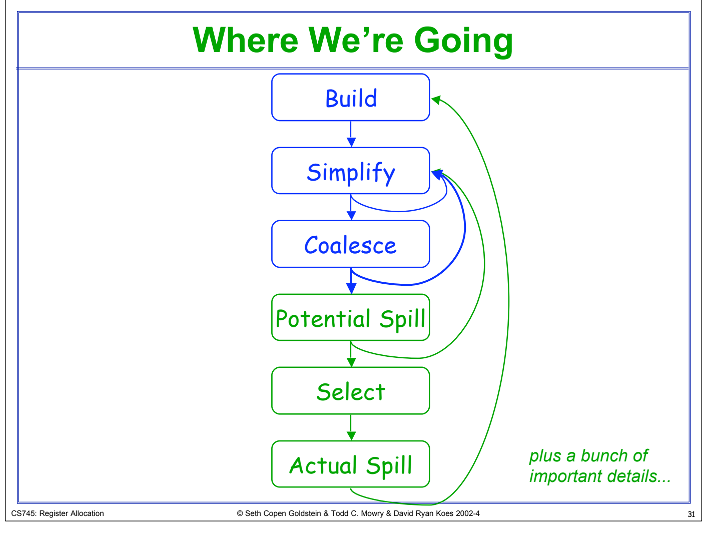
- Why not?
- With Briggs, one needs to look at all neighbors of a & b
- With George, only need to look at neighbors of a.

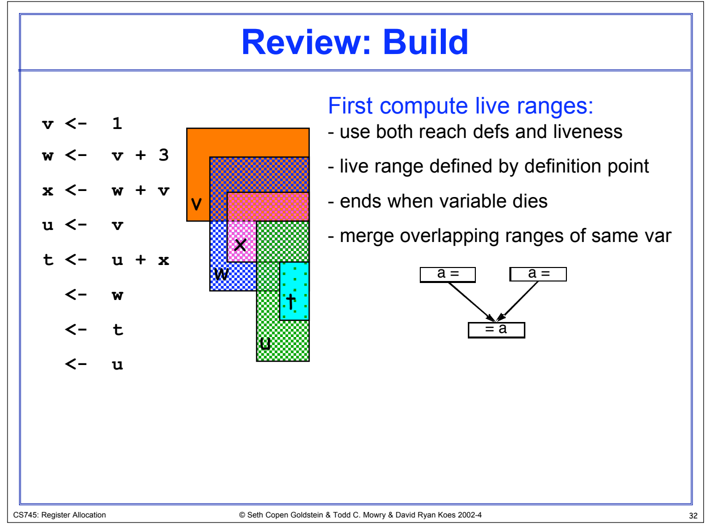
So:

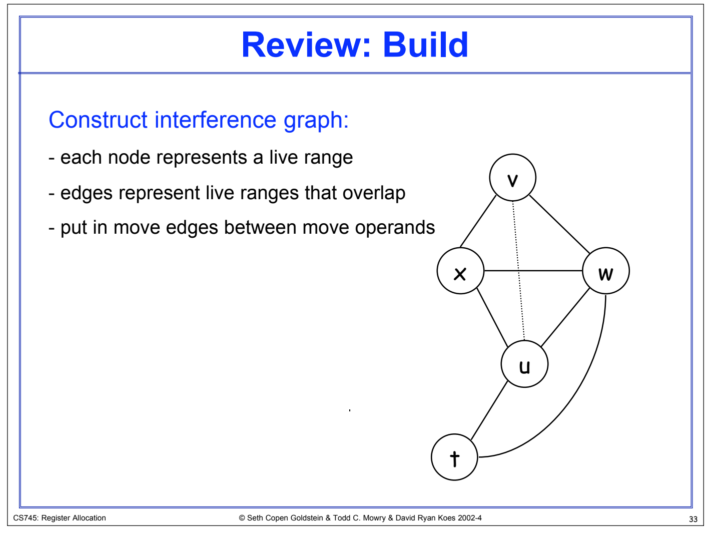
- Use George if one of a & b has very large degree
- Use Briggs otherwise

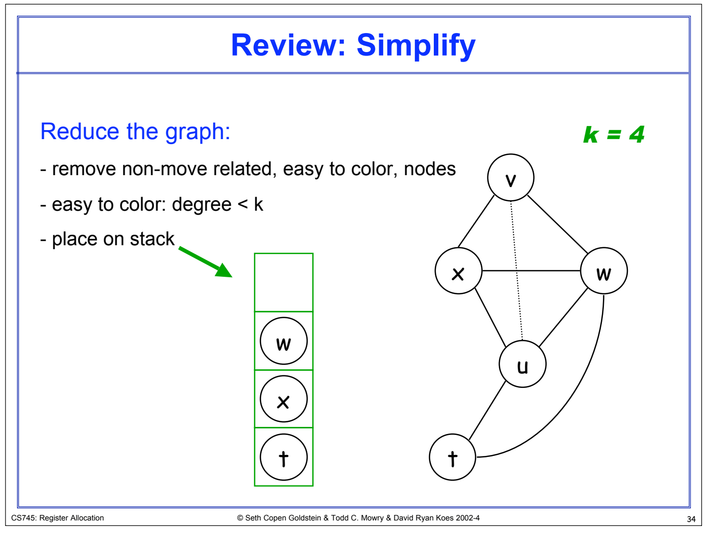
 © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4

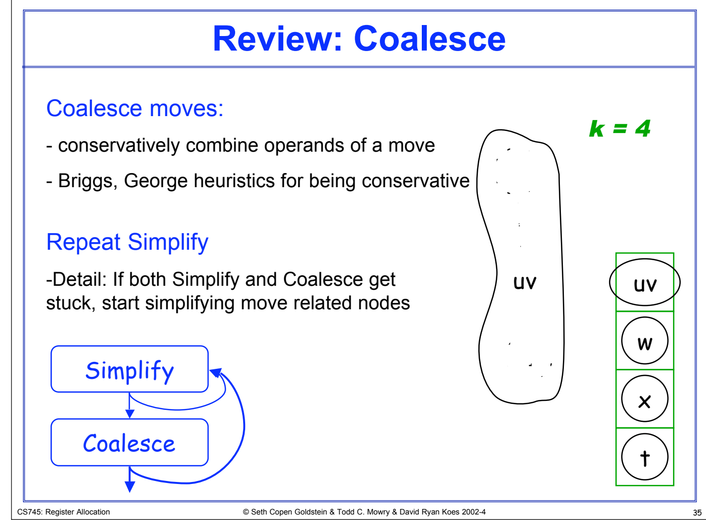


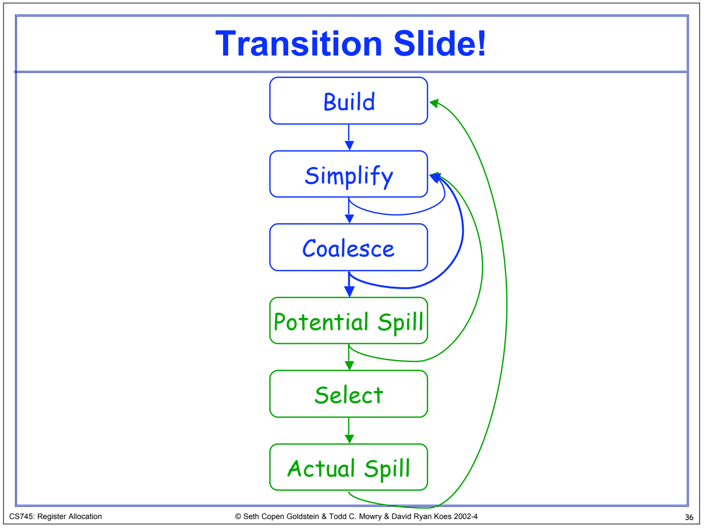


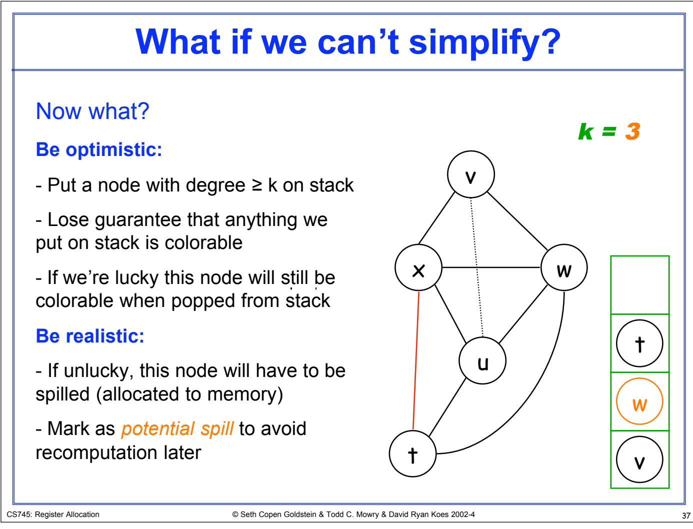


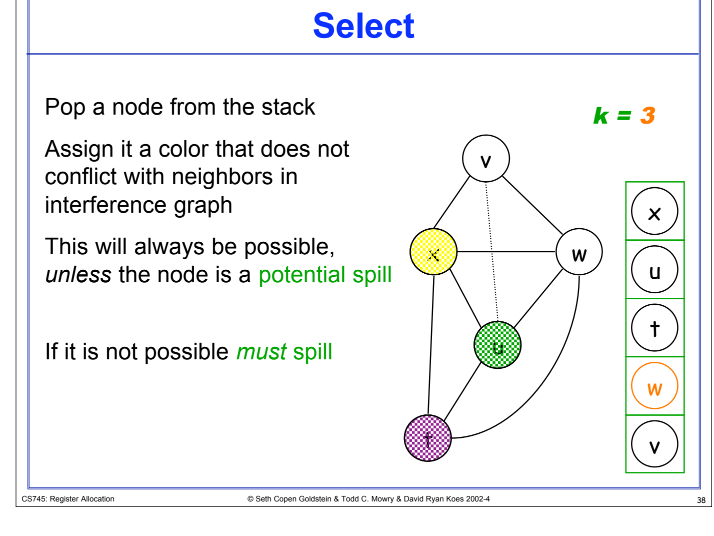

Review: Build
Construct interference graph:
- each node represents a live range
- edges represent live ranges that overlap
- put in move edges between move operands
v
x
w
u
t
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
33

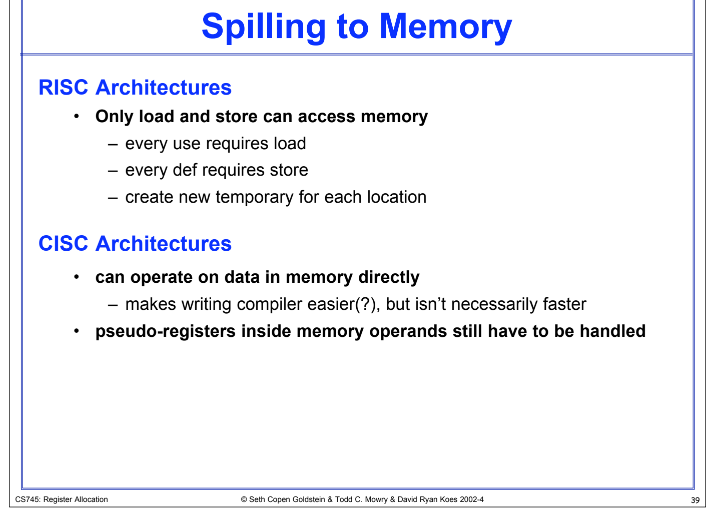

Review: Simplify
Reduce the graph:
k = 4
- remove non-move related, easy to color, nodes
- easy to color: degree < k
- place on stack
w
x
t
v
x
w
u
t
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
34

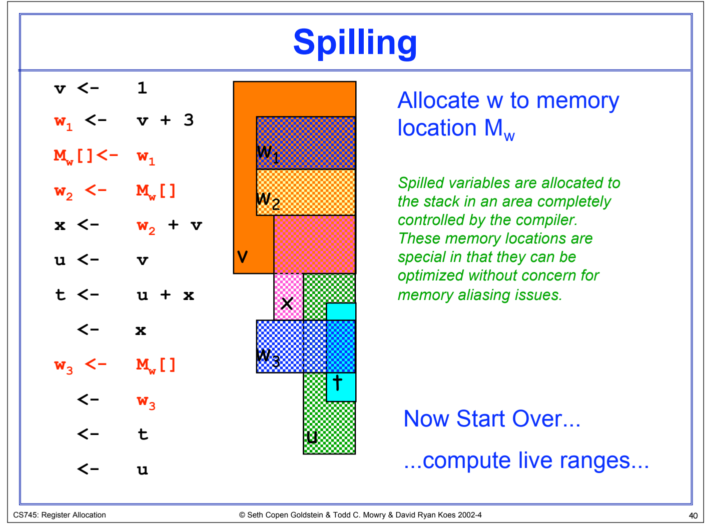

Review: Coalesce
Coalesce moves:
k = 4
- conservatively combine operands of a move
- Briggs, George heuristics for being conservative
Repeat Simplify
-Detail: If both Simplify and Coalesce get stuck, start simplifying move related nodes
uv
uv
w
x
t
Simplify
Coalesce
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
35

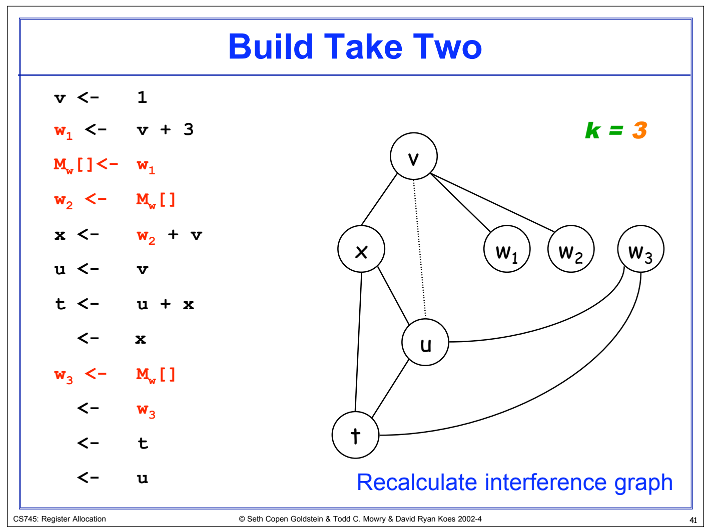

Transition Slide!
Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
36

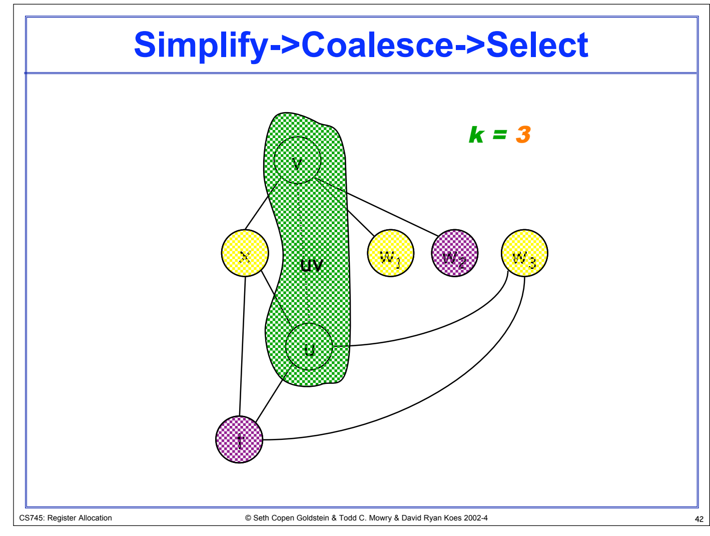

What if we can't simplify?
Now what?
Be optimistic:
- Put a node with degree ≥ k on stack
- Lose guarantee that anything we put on stack is colorable
- If we're lucky this node will still be colorable when popped from stack
Be realistic:
- If unlucky, this node will have to be spilled (allocated to memory)
- Mark as potential spill to avoid recomputation later
k = 3
v
x
w
u
t
t
w
v
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
37


Select
Pop a node from the stack
Assign it a color that does not conflict with neighbors in interference graph
This will always be possible, unless the node is a potential spill
If it is not possible must spill
k = 3
v
x
w
u
t
x
u
t
w
v
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
38


Spilling to Memory
RISC Architectures
Only load and store can access memory
every use requires load
every def requires store
create new temporary for each location
CISC Architectures
can operate on data in memory directly
makes writing compiler easier(?), but isn't necessarily faster
pseudo-registers inside memory operands still have to be handled
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
39


Spilling
v <- 1
w1 <- v + 3
Mw[]<- w1
w2 <- Mw[]
x <- w2 + v
u <- v
t <- u + x
<- x
w3 <- Mw[]
<- w3
<- t
<- u
Allocate w to memory location Mw
Spilled variables are allocated to the stack in an area completely controlled by the compiler. These memory locations are special in that they can be optimized without concern for memory aliasing issues.
Now Start Over...
...compute live ranges...
CS745: Register Allocation
© Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4
40





Spilling

We have to start from scratch every time we spill

- **Suggestions?**
  - Fewer iterations?
  - Faster iterations?

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 43

What to Spill?

When choosing potential spill node want:

- **A node that makes graph easier to color**
  - Fewer spills later
- **A node that isn't "expensive" to spill**
  - First nodes pushed on stack are last to be colored
    - ➢ more likely to be spilled
  - An expensive node would slow down the program if spilled
- **We can apply heuristics both when choosing potential spill nodes and when choosing actual spill nodes**
  - not required to spill node that we popped off stack and can't color

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 44

A Spill Heuristic

Pick node (live range) *n* that minimizes:

$$\frac{\sum_{def \in n} 10^{depth(def)} + \sum_{use \in n} 10^{depth(use)}}{\text{degree}(n)}$$

This heuristic prefers nodes that:

- **Are used infrequently**
- **Aren't used inside of loops**
- **Have a large degree**

Could use any one of several other heuristics as well...

Reducing Stack Frame Size

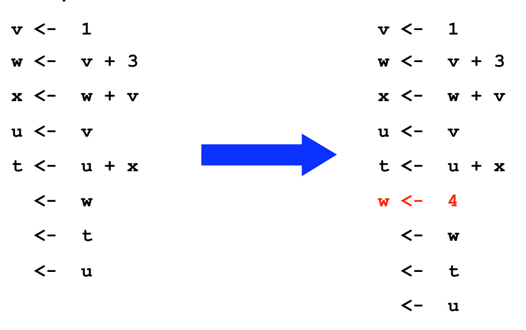
- How do you allocate spilled live ranges?
  - every live range gets its own location on the stack frame
  - or we can be smarter...
- What about `mov a,b` where both a & b have been spilled?
- Use graph-coloring with aggressive coalescing!
- Use liveness info to create an interference graph of the spilled nodes
- Always coalesce
- Simplify/Select
- Colors map to frame locations

Is it worth it?

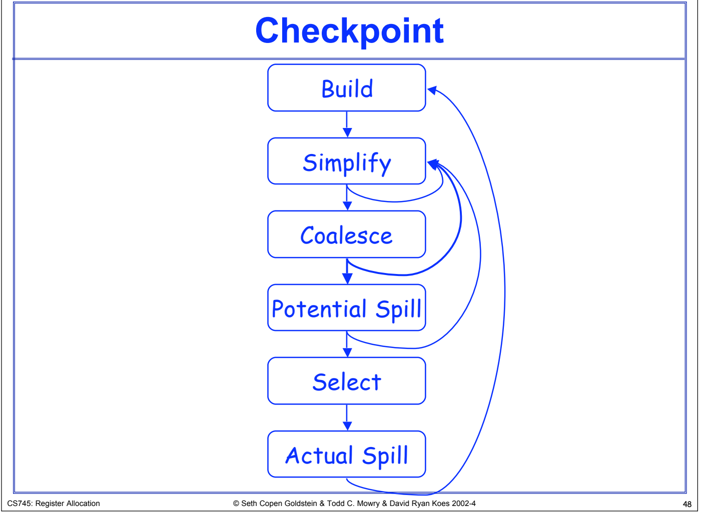
Rematerialization

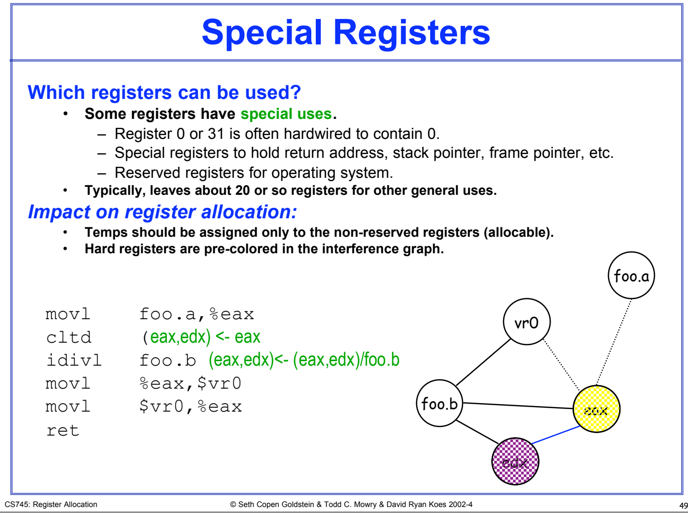
An alternative to spilling

- **Recompute value of variable instead of store/load to memory**
- **Example:**



Checkpoint





Register Usage Conventions

Certain registers are used for specific purposes defined by the *standard calling convention*.

- **4-6 argument registers.**
  - The first 4-6 arguments to procedures/functions are always passed in these registers.
- **~8 callee-save registers.**
  - These registers must be preserved across procedure calls. Thus, if a procedure wants to use a callee-save register, it must first save the old value and then restore it before returning.
- **The remainder are caller-save registers.**
  - These are not preserved across procedure calls. Thus, a procedure is free to use them without saving first.
  - Includes the argument registers.

How do we support these?

- **neat trick for handling callee save**
- **call instruction**

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 50

Allocating Callee-Save Registers

Move callee-save reg to temp at start of procedure
Move it back at end of procedure
What happens if there is no register pressure?
What happens if there is a lot of register pressure?

```
entry:    define r
       temp <- r
       …
exit:  r <- temp
       use r
```

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 51

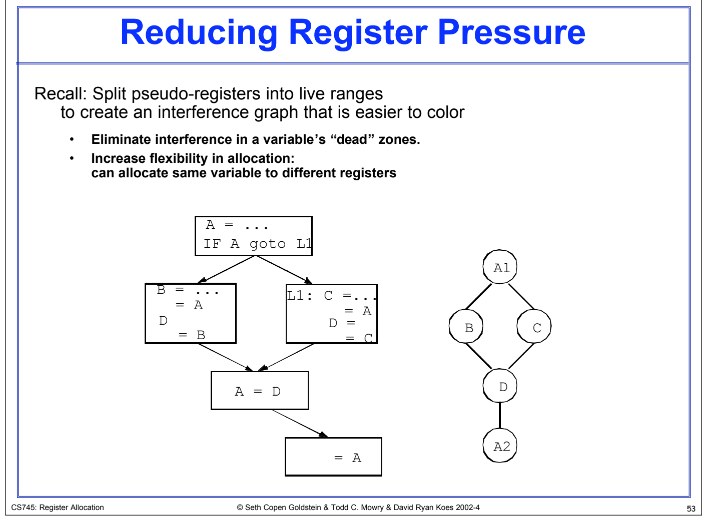
Allocating to callee-save registers

CALL instruction "modifies" all caller-save regs

entry: define r_e
  t_1 <- r_e
  x <-
  …
  call
    (r1,r2,r3 <-)
    (<- r1,r2,r3)
  …
  <- x
exit: r_e <- t_1
  use r_e

t_1, r1, r2, x, r3, r_e

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 52

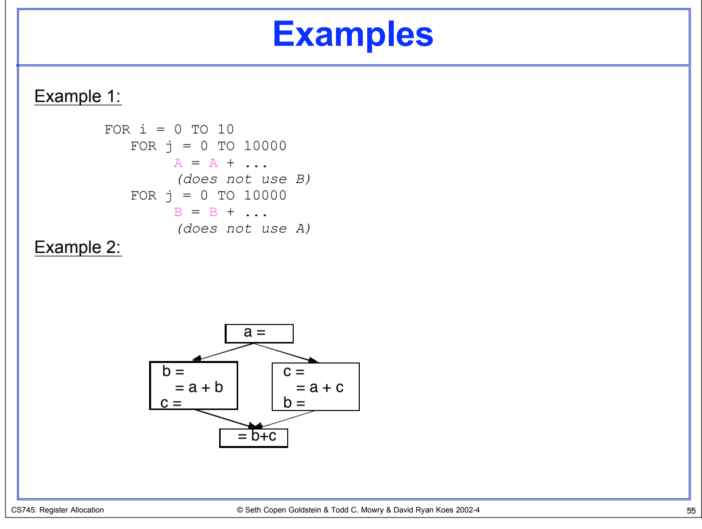


Insight

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

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 54



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**

CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 56

One Algorithm

Observation: Spilling is absolutely necessary if
- number of live ranges active at a program point > n *not degree in graph*

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

Alternative Allocators

Graph allocator, as described, has issues
- **What are they?**

Alternative: Single pass graph coloring
- **Build, Simplify, Coalesce as before**
- **In select, if can't color with register, color with stack location**
  - Keep going
- **Requires second, reload phase**
  - "fixes" spilled variables
  - Requires that we reserve a register
  - Can get messy

Claim: Does a pretty good job
- **Why?**
  - Key is order nodes are colored…

Advantages? Disadvantages?

Alternative Allocators

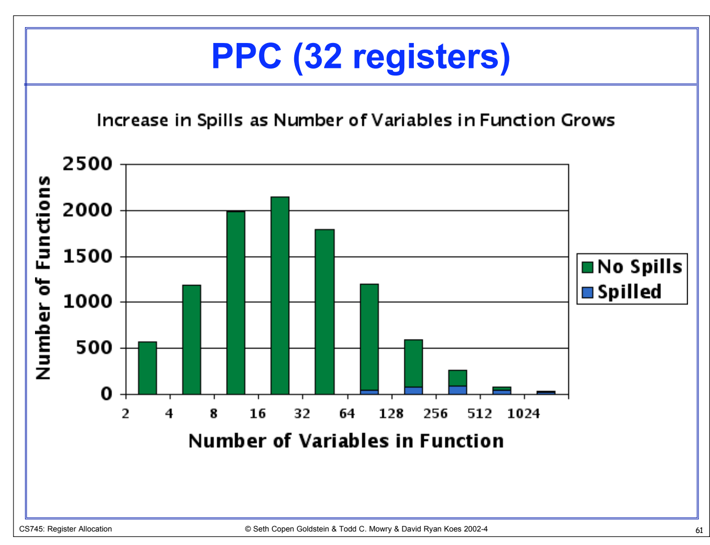
Local/Global Allocation

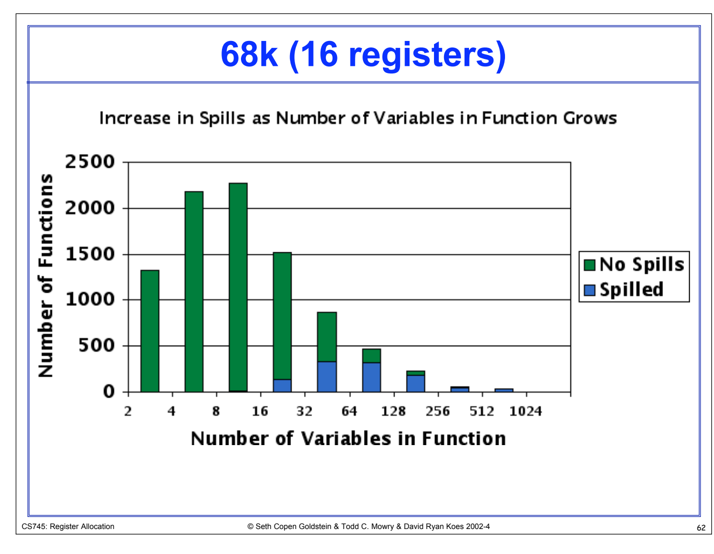
gcc's approach, unless -fnew-ra

- **Allocate "local" pseudo-registers**
  - Lifetime contained within basic block
  - Register sufficiency no longer NP-Complete!
- **Allocate global pseudo-registers**
  - Single pass global coloring
- **Reload pass to fix spills (allocator does not generate spill code)**
- **Can also do global then local (Morgan)**
- **Advantages? Disadvantages?**

How good is it in practice?

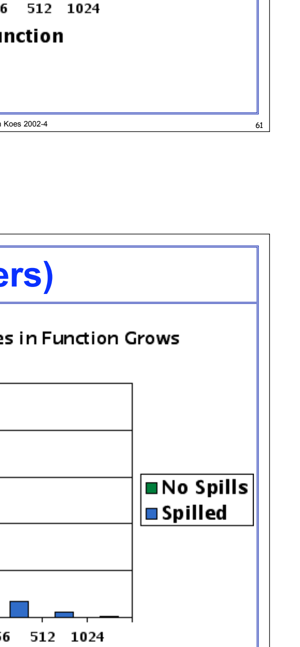
- Used `gcc -fnew-ra` to compile >10,000 functions from Mediabench, Spec95, Spec2000, and micro-benchmarks
- Recorded for which functions graph coloring had to spill





x86 (8 registers)

Increase in Spills as Number of Variables in Function Grows



CS745: Register Allocation © Seth Copen Goldstein & Todd C. Mowry & David Ryan Koes 2002-4 63