Lecture 24

EECS 483: COMPILER CONSTRUCTION

Announcements

- HW6: Analysis and Optimization
  - Due on Thursday, May 2
- Final Exam
  - 4-6pm April 29
  - DOW1010, DOW1005, DOW2166
  - Same cheat sheet policy as midterm
- Guest Lectures
  - Steven Schaefer today
  - Professor New returns for final recap/review lecture on Monday.

REGISTER ALLOCATION

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs
- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed
- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

1. Compute liveness information: `live(x)`
   - recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `pal` be the set of usable registers
   - usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   - locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   - `used = {r | reg r = uid_loc(y)` s.t. `y` ∈ `live(x)}`
   - `available = pal - used`
   - If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n ;` n = n + 1
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

For HW6

- HW 6 implements two naive register allocation strategies:
  - `none`: spill all registers
  - `greedy`: uses linear scan
- Also offers choice of liveness
  - trivial: assume all variables are live everywhere
  - dataflow: use the dataflow algorithms

- Your job: do "better" than these.
- Quality Metric:
  - registers other than rbp count positively
  - rbp counts negatively (it is used for spilling)
  - shorter code is better

- Linear scan is OK
  - but… how can we do better?

GRAPH COLORING

Register Allocation

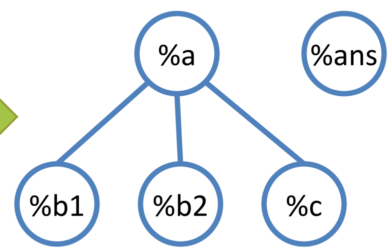
Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, “spill” a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

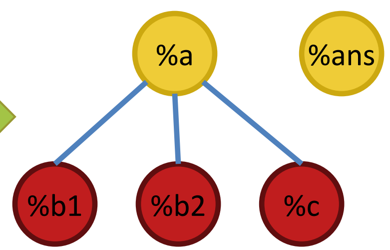
Interference Graphs

- Nodes of the graph are %uids
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

Register Allocation Questions

- Can we efficiently find a k-coloring of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using heuristics.
- How do we assign registers to colors?
  - If we do this in a smart way, we can eliminate redundant MOV instructions.
- What do we do when there aren't enough colors/registers?
  - We have to use stack space, but how do we do this effectively?

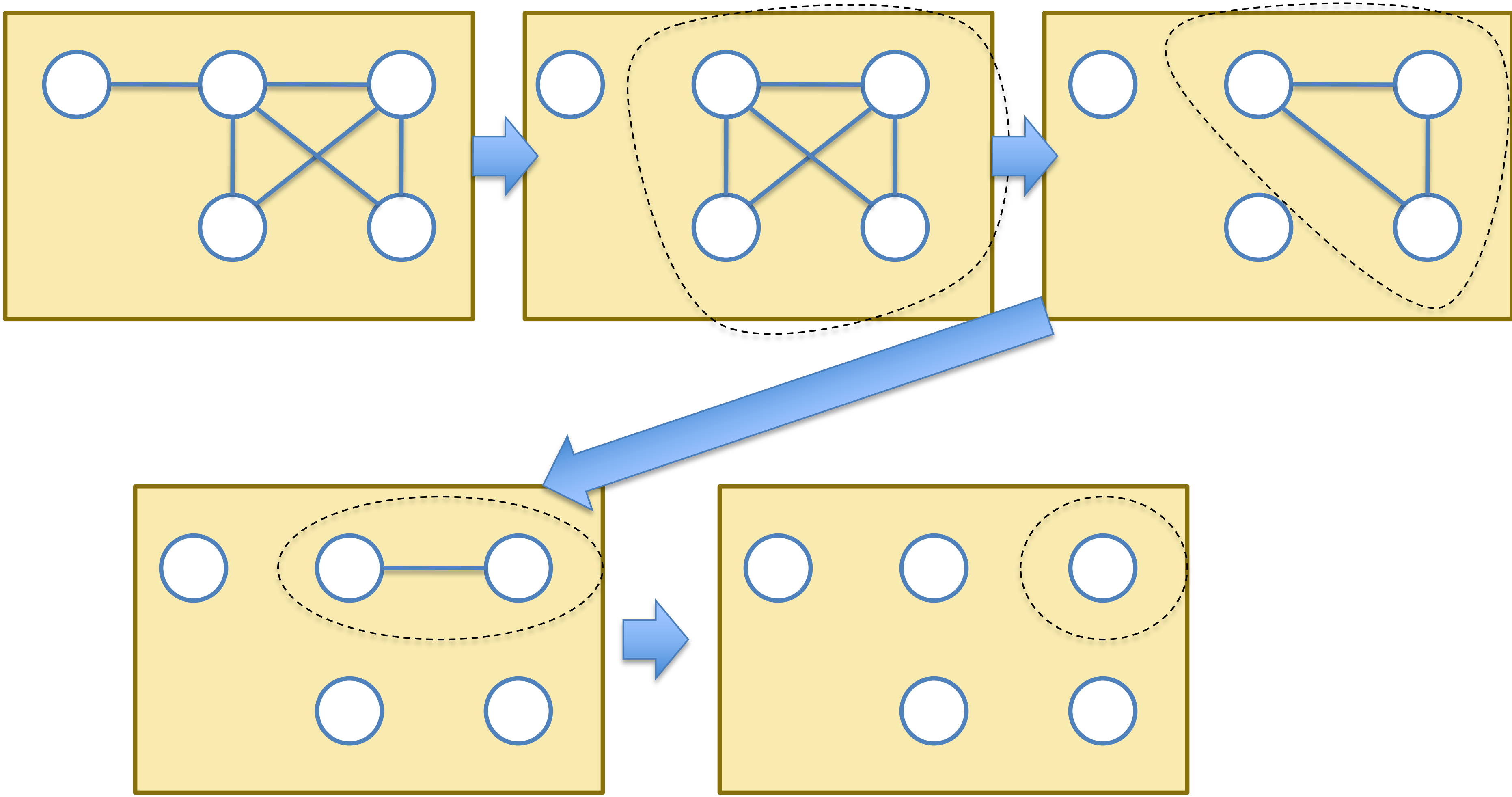
Coloring a Graph: Kempe's Algorithm

Kempe [1879] provides this algorithm for K-coloring a graph.

It's a recursive algorithm that works in three steps:

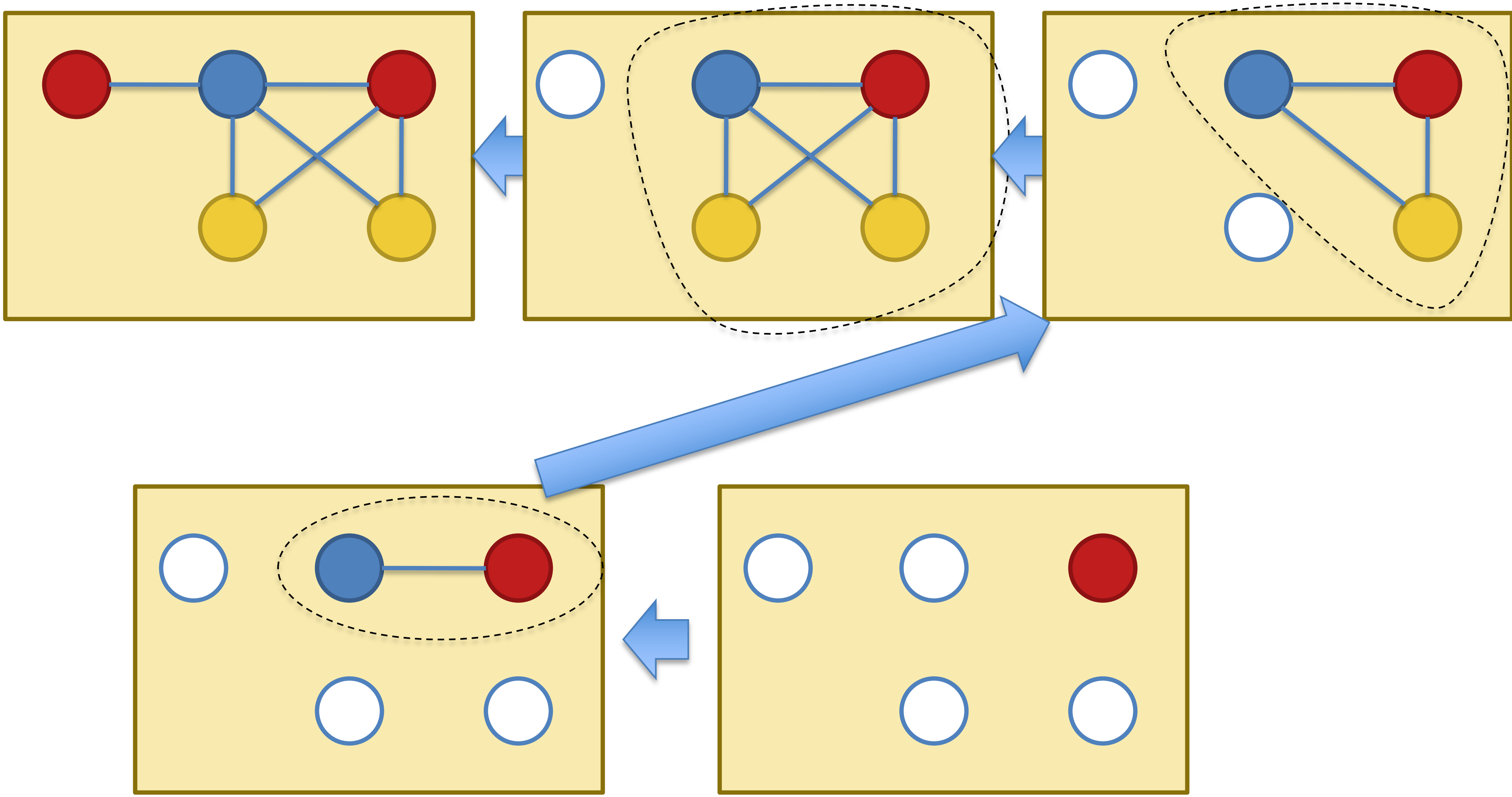
1. Find a node with degree < K and cut it out of the graph.
   - Remove the nodes and edges.
   - This is called *simplifying* the graph
2. Recursively K-color the remaining subgraph
3. When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



Recursing Down the Simplified Graphs

Example: 3-color this Graph

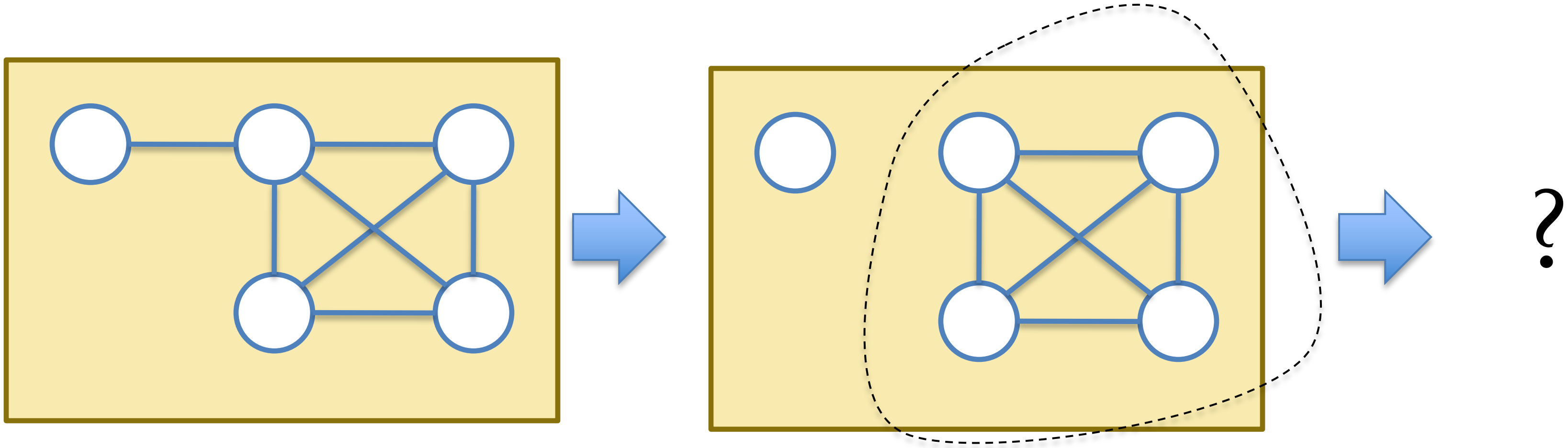


Assigning Colors on the way back up.

Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least K neighbors.
  - This can happen even when the graph is K-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:



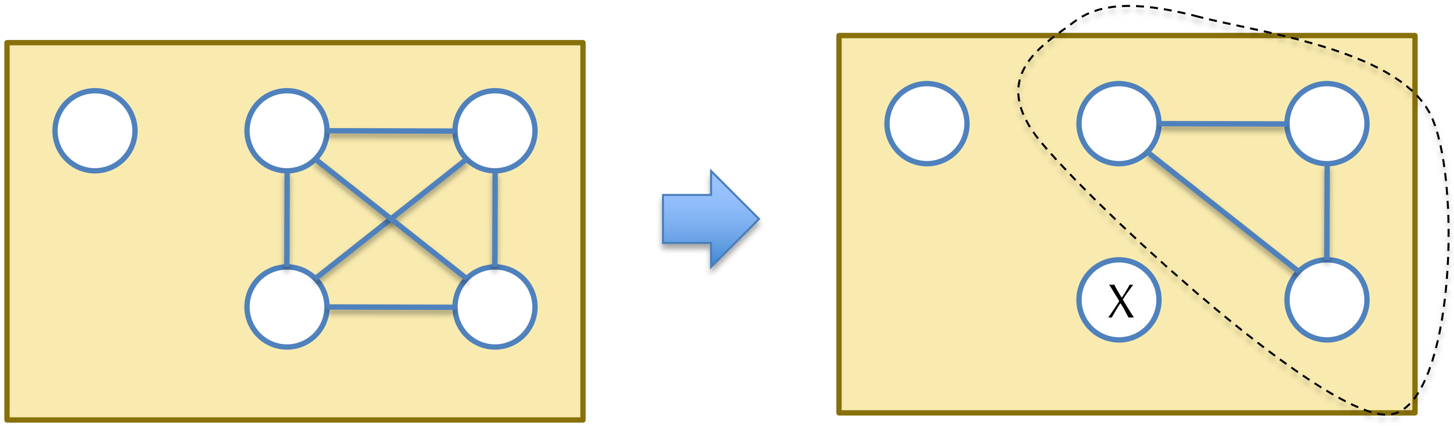
Spilling

- Idea: If we can't K-color the graph, we need to store one temporary variable on the stack.
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference
    (since removing it will make the graph easier to color)
- In practice: some weighted combination of these criteria

- When coloring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively coloring

Spilling, Pictorially

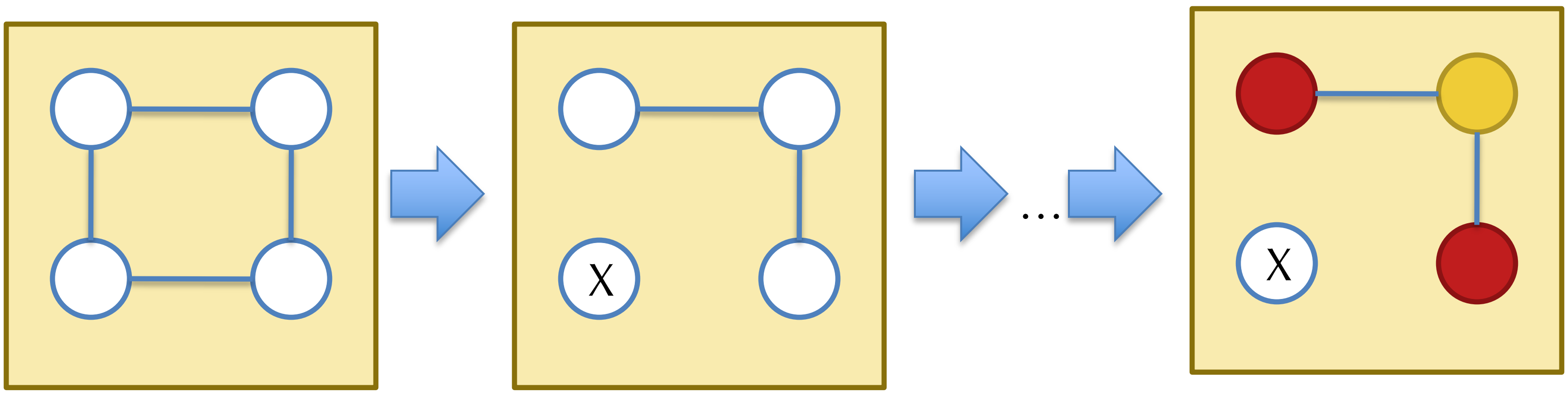
- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring



Optimistic Coloring

- Sometimes it is possible to color a node marked for spilling.
  - If we get "lucky" with the choices of colors made earlier.

- Example: When 2-coloring this graph:



- Even though the node was marked for spilling, we can color it.
- So: on the way down, mark for spilling, but don't actually spill…

Accessing Spilled Registers

- If optimistic coloring fails, we need to generate code to move the spilled temporary to & from memory.
- Option 1: Reserve registers specifically for moving to/from memory.
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  - Pro: Only need to color the graph once.
  - Not good on 32bit x86 because there are too few registers & too many constraints on how they can be used.
  - OK on 64bit x86 and other processors. (We use this for HW6)

- Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.
  - Pro: Need to reserve fewer registers.
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph

Example Spill Code

- Suppose temporary t is marked for spilling to stack slot located at [rbp+offs]
- Rewrite the program like this:

```
t = a op b;
...

x = t op c
...

y = d op t
```

```
t = a op b                      // defn. of t
        Mov [rbp+offs], t
        ...
Mov t37, [rbp+offs]// use 1 of t
        x = t37 op c
        ...
Mov t38, [rbp+offs]// use  2 of t
        y = d op t38
```

- Here, t37 and t38 are freshly generated temporaries that replace t for different uses of t.
- Rewriting the code in this way breaks t's live range up:

    t, t37, t38 are only live across one edge

Precolored Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The "Call" instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colors.
  - Pre-colored nodes can't be removed during simplification.
  - Trick: Treat pre-colored nodes as having "infinite" degree in the interference graph – this guarantees they won't be simplified.
  - When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes.

Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:

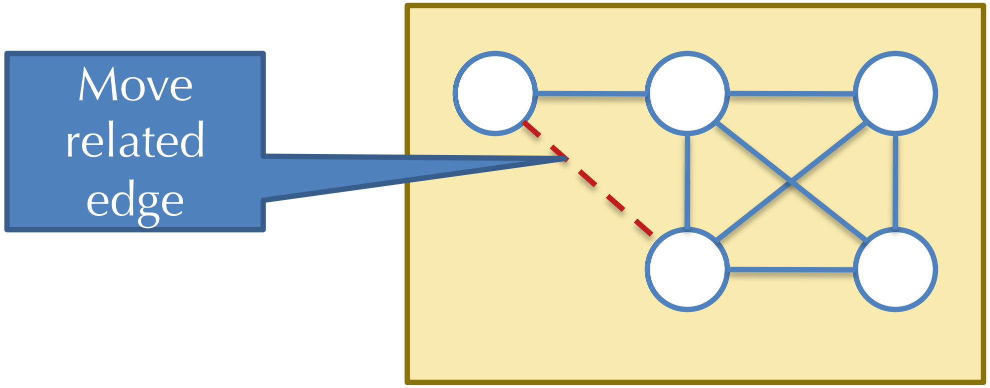
```
movq t1, t2
```

  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

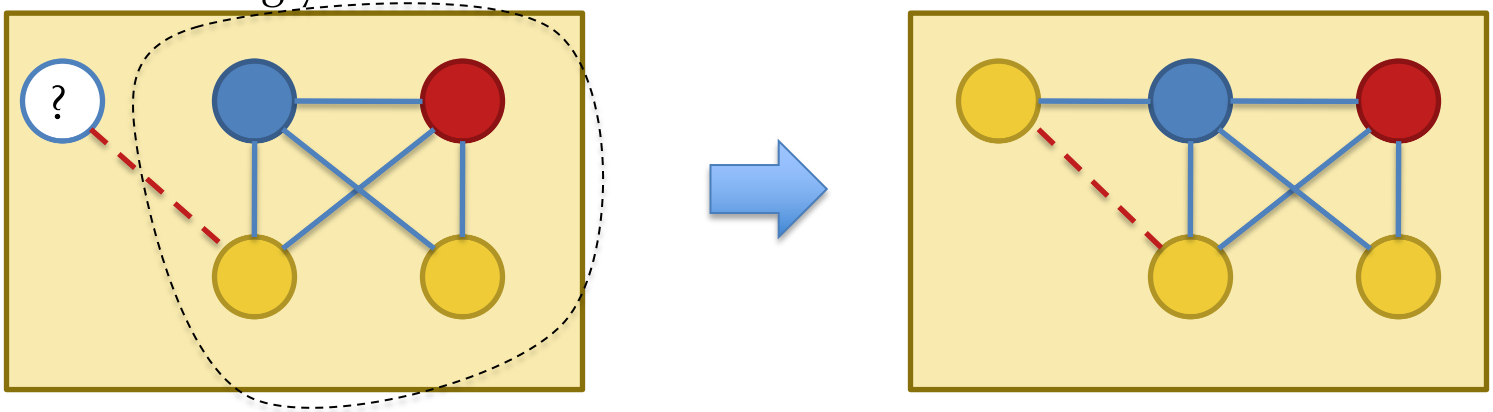
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible, pick a color of an already colored node reachable by a move-related edge.

Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

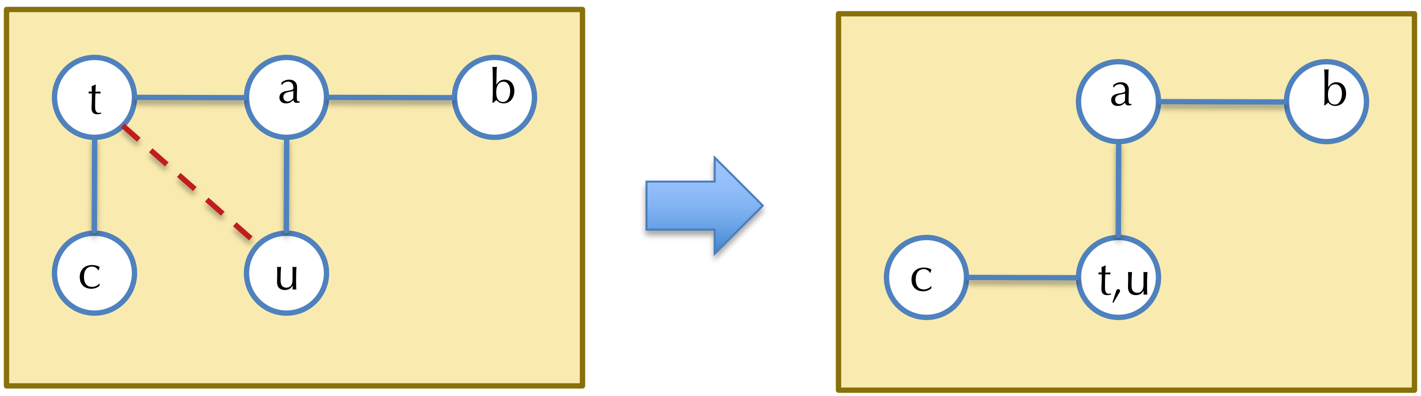


- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

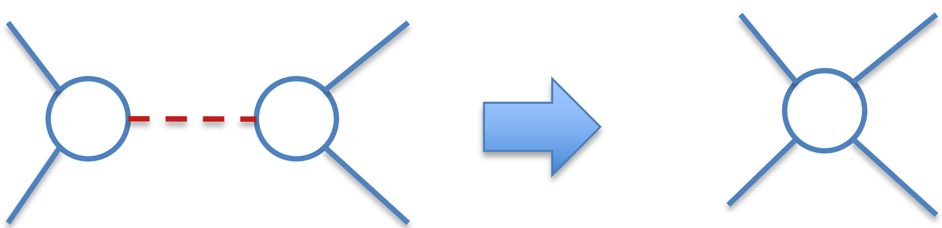


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

1. *Brigg's strategy*: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors (with degree ≥ k).

2. *George's strategy:* We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

Last details

- After register allocation, the compiler should do a peephole optimization pass to remove redundant moves.
- Some architectures specify calling conventions that use registers to pass function arguments.
  - It's helpful to move such arguments into temporaries in the function prelude so that the compiler has as much freedom as possible during register allocation. (Not an issue on X86, though.)

Chaitin's Algorithm

Chaitin's algorithm is efficient (O(|V| + |E|), simple to implement

Chaitin's Algorithm

Chaitin's algorithm is efficient ($O(|V| + |E|)$), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**

Chaitin's Algorithm

Chaitin's algorithm is efficient (O(|V| + |E|), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**
- For every graph, there is a elimination ordering such that Chaitin's algorithm produces an optimal coloring
  - therefore finding this optimal elimination ordering for a general graph is NP-complete

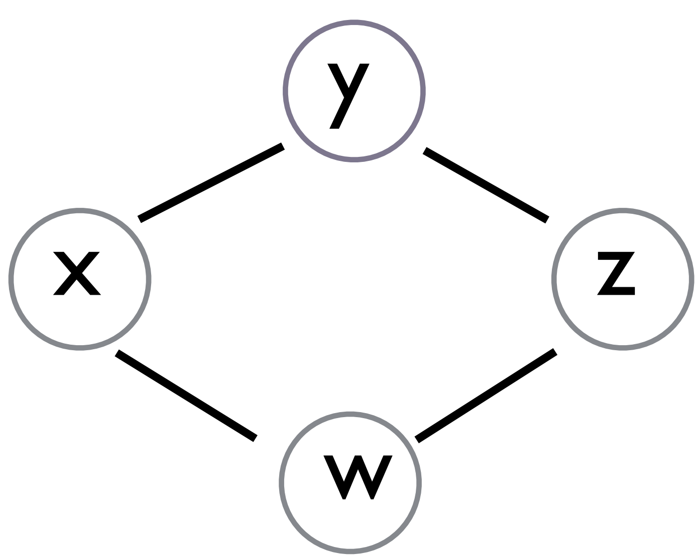
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

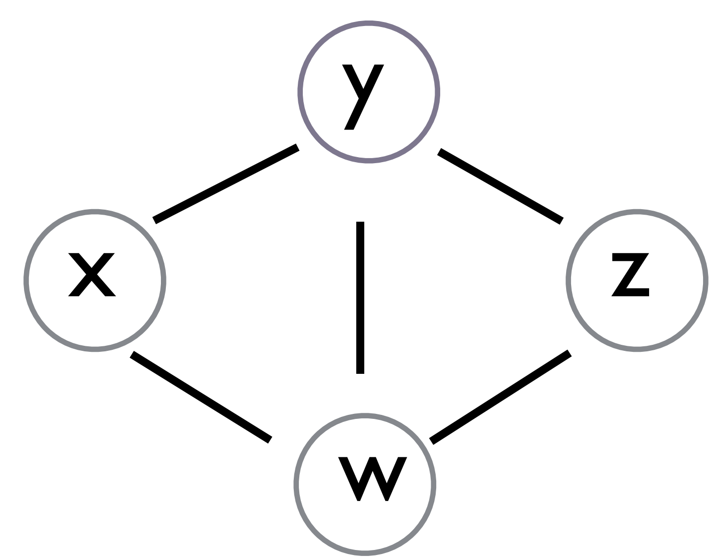
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

- The interference graphs of an SSA program are all **chordal**
  - Every cycle >= 4 nodes has a **chord**



Not chordal



chordal

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order

Coloring Chordal Graphs

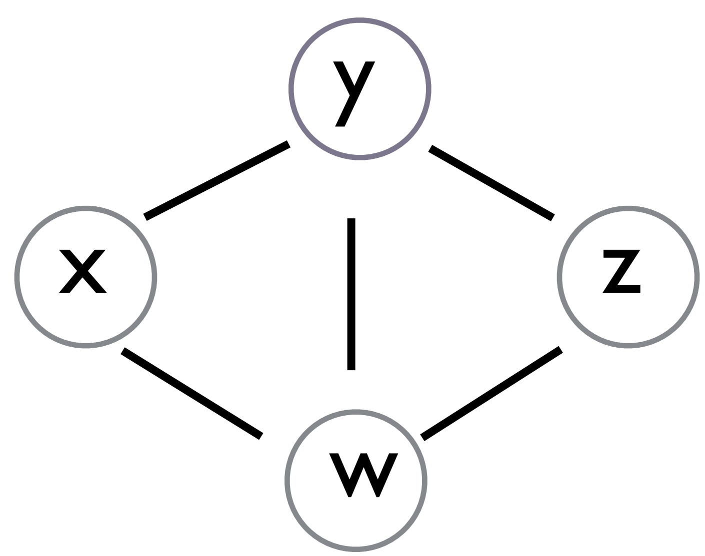
Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO



x,y,z,w
not perfect: N(w) non-clique

w,x,y,z
perfect

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

- a variable x dominates y if is in scope when y is defined (includes simultaneous defs)

Every SSA Interference Graph is Chordal

Theorem: A graph is chordal iff it has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors later in the order

SSA programs have a simple PEO:

"in-scope" or "dominance" relation

  - a variable x dominates y if is in scope when y is defined (includes simultaneous defs)

- x's definition is "closer to the root" of the AST than y
- easy to compute: pre-order traversal of the nodes