Administration

- Announcements
  - Final exam is scheduled: Apr 18 2018, 7:00pm
- Minijava project
  - Phase 4 due next week
  - The contest is on!!
Register Allocation

- The analysis we have done so far produces an interference graph
- How do we colour it with 4 colours?
Core of the Algorithm

• The core of the algorithm is simple. It works by reducing the colouring problem to a smaller equivalent problem using the following observation:
  – Any node that has fewer than $K$ neighbours is certain to be colourable, provided that we can colour the graph that remains after this node is removed.
  – Why is this observation true?
Core of the Algorithm

1. Build the interference graph

2. Simplify:
   - Find a node $n$ that has degree $< K$
   - push $n$ on a stack
   - remove $n$ from the graph
   - repeat step 2 until the graph is empty

3. Colour:
   - Pop a node $n$ from the stack
   - Assign $n$ a valid colour (in the original graph, thereby considering all of $n$’s neighbours)
   - repeat step 3 until the stack is empty
When does the Algorithm fail?

1. Build the interference graph

2. Simplify:
   - Find a node $n$ that has degree $\leq K$
   - push $n$ on a stack
   - remove $n$ from the graph
   - repeat step 2 until the graph is empty

3. Colour:
   - Pop a node $n$ from the stack
   - Assign $n$ a valid colour (in the original graph, thereby considering all of $n$’s neighbours)
   - repeat step 3 until the stack is empty
When does the Algorithm fail?

1. Build the interference graph
2. Simplify:
   - Find a node $n$ that has degree $< K$
   - push $n$ on a stack
   - remove $n$ from the graph
   - repeat step 2 until the graph is empty
3. Colour:
   - Pop a node $n$ from the stack
   - Assign $n$ a valid colour (in the original graph, thereby considering all of $n$’s neighbours)
   - repeat step 3 until the stack is empty
What to do when the Algorithm fails?

- When Simplify can’t find a node with degree < K:
  - Pick a “potential” spill node (s).
  - Any node will do, although some choices are better than others
  - Then “pretend” s has degree less than K and proceed as normal:
    - Push s onto the stack
    - Remove s from the graph
    - Continue step 2 (Simplify)
When does the Algorithm fail now?

1. Build the interference graph
2. Simplify:
   - Find a node \( n \) that has degree < \( K \) (or not)
   - push \( n \) on a stack
   - remove \( n \) from the graph
   - repeat step 2 until the graph is empty
3. Colour:
   - Pop a node \( n \) from the stack
   - Assign \( n \) a valid colour (in the original graph, thereby considering all of \( n \)'s neighbours)
   - repeat step 3 until the stack is empty
When does the Algorithm fail?

1. Build the interference graph

2. Simplify:
   - Find a node $n$ that has degree $< K$ (or not)
   - push $n$ on a stack
   - remove $n$ from the graph
   - repeat step 2 until the graph is empty

3. Colour:
   - Pop a node $n$ from the stack
   - Assign $n$ a valid colour (in the original graph, thereby considering all of $n$’s neighbours)
   - repeat step 3 until the stack is empty
If we had a “potential” spill?

- In the colouring step:
  - Pop a node $n$ from the stack
  - Assign $n$ a valid colour (in the original graph, thereby considering all of $n$’s neighbours)
  - Repeat step 3 until the stack is empty
- This may not work any more, although we might get lucky and have it work! How?
- If the potential spill node isn’t colourable, then what?
  - It becomes an “actual” spill
Handling “actual” spills

- Don’t colour the spilled temp, just mark it as a spill.
- Continue the select phase to find other actual spills.
- Since we didn’t colour all the temps, the code won’t work.
- So, rewrite the assembly code:
  - allocate an InFrame location for each spilled Temp
  - before each use, load the Temp from memory
  - after each def, store the Temp into memory
  - Start over with the new assembly code
Handling “actual” spills – Implementation Notes

- Don’t colour the spilled temp, just mark it as a spill.
  - The sample implementation does this by marking it with a special SpillColour

```java
public class SpillColor extends Color {
    private IRExp location;
    private Access access;

    public SpillColor(Frame frame) {
        access = frame.allocLocal(true);
        location = access.exp(frame.FP());
    }
    ...
}
```
Handling “actual” spills – Implementation Notes

- Re-write the assembly code
- The book is a bit unclear about just how to do this
- It is hard to do this in an architecture-independent way
- The sample solution does this by rerunning the translate phase on the “painted” IR code
  - Needs special patterns to match spilled Temps, i.e., Temps that are painted with a SpillColor in the IR
More complex register allocation

- Simplification isn’t the only way to simplify the interference graph
- Another way is to coalesce two move-related nodes, which has the side-effect of eliminating the move instruction
- In principle we can coalesce any two move-related nodes
- In practice, this increases the degree of the nodes in the interference graph, which makes it harder to colour it.
Adding a Coalesce Step

1. Build the interference graph

2. Simplify:
   - Find a non-move related node \( n \) that has degree < \( K \)
   - push \( n \) on a stack
   - remove \( n \) from the graph
   - repeat step 2 until the graph is empty

3. Coalesce two move-related nodes
   - Find two move-related nodes that are “safe” to coalesce
   - Merge them into a single node
   - Remove the corresponding move from consideration

4. Colour: ...
Adding a Coalesce Step

3. Coalesce two move-related nodes
   – Find two move-related nodes that are “safe” to coalesce
   – Merge them into a single node
   – Remove the corresponding move from consideration

• What does it mean for a move to be “safe” to coalesce?
• What if there are no such safe coalesceable nodes?
Safe?

• Why wouldn’t a move be “safe” to coalesce?
  – Because the nodes are “constrained”
  – Because if we coalesced the two nodes, a colourable graph might become un-colourable
  – How can we tell?

--- move-related
___ interference
Test for safety

• **Briggs**
  – Nodes a and b can be coalesced if the resulting node $ab$ will have fewer than $K$ neighbours of significant degree

• **George**
  – Nodes a and b can be coalesced if, for every neighbour $t$ of a, either $t$ already interferes with b, or $t$ is of insignificant degree

• **Significant degree**
  – A node has significant degree if it has $\geq K$ edges
Conservative safety

- Both the George and Briggs tests are conservative
- It is possible to coalesce a pair of nodes that fail both tests without jeopardizing colourability of the graph
  - But we can’t guarantee it
Getting stuck (again)

- What if we can’t simplify or coalesce?
- We might be able to make progress by giving up on a move-related node
  - Find a move-related node that can be simplified, but not coalesced.
  - Give up on coalescing it, and simplify it instead
  - The book calls this “Freezing” a node
- What effect does doing this have on the generated code?
Complication: Handling “Special” Temps

Example: Generated X86_64 for a simple method.

```
    _Test_do:
    pushq %rbp
    movq %rsp, %rbp
    _L_3:
        movq %rdi, t005
        movq %rsi, t006
        movq %rdx, t007
        imulq $99, t007, t016
        movq t016, t015
        addq t006, t015
        movq t015, %rax
    _DONE_4:
        #return sink
        leave
        ret
```

Special Temps like FP, SP, RV, and argument temps should always be allocated to some specific register.

Q: How to handle this?
Precoloured nodes

• All registers of the processor are represented by precoloured Temps.

• When used explicitly in Assembly code, their live ranges will cause them to interfere with other (non-precoloured) Temps that are live at the same time.

• This takes care of “quirky” register use by some instructions, e.g., idivq on the X86_86
  – Divides the 128 bit dividend in (%rdx,%rax) by a given register. Quotient in %rax, remainder in %rdx.

• What if a precoloured Temp is not live at the same time as some “normal” Temp?
  – Then that Temp can be allocated in that register.
The final algorithm

1. Build the interference graph

2. Simplify:
   - Find a non-move-related, non-precoloured node \( n \) that has degree \( < K \)
   - Push \( n \) on a stack
   - Remove \( n \) from the graph
   - Repeat step 2 until there are only precoloured nodes

3. Coalesce two move-related nodes
   - Find two move-related nodes that are “safe” to coalesce
   - Merge them into a single node
   - Remove the corresponding move from consideration

4. Colour: ...
Caller vs. callee save regs

• Given a variable, when would you prefer to allocate it to:
  – A callee-save register?
  – A caller-save register?

• Does the register allocator “do the right thing?”