Overview of the next little while

Code Generation

Assembly with Temps

Flow Graph: directed graph
Nodes = Assembly instructions
Edges = Transfer of control
Liveness: for each node in flowgraph:
- Compute which Temps are “live”
  - just before that instruction
  - just after that instruction
Interference Graph: undirected graph
Nodes: Temps
Edge: $t_1 \leftrightarrow t_2$ : $t_1$ and $t_2$ can not be allocated to same register

Register allocation (by Graph Colouring)

Chapter 9

Chapter 10

Chapter 11
Example:

```
movq $0, t1
L1:   movq t1, t2
addq $1, t2
addq t2, t3
imulq $2, t2, t1
cmpq $N, t1
jl   L1
movq t3, %rax
```

Control Flow Graph

1: `movq $0, t1`
2: `movq t1, t2`
3: `addq $1, t2`
4: `addq t2, t3`
5: `imulq $2, t2, t1`
6: `cmpq $N, t1`
7: `jl L1`
8: `movq t3, %rax`
Interference Graph

- Liveness information can be used for many optimizations.
- One important use is register allocation.
- To drive register allocation, we compute an alternative representation of the liveness information, the “interference graph”.
Interference Graph

- **Nodes:**
  - Temps

- **Edges:**
  - An edge $t_1 \rightarrow t_2$ means that $t_1$ and $t_2$ are live at the same time, and therefore can’t be allocated to the same register.
Computing the Interference Graph

• When you have liveness information for each instruction in the program, you can compute the interference graph as follows:
  – For each Temp in the program, create a node
  – If any instruction defines a Temp, that Temp interferes with every other Temp that is live out.
  – For each instruction, instr, in the program:
    for each d in def[instr]
    for each live in out[instr]
      if (live != d)
        add edge d <-> live
An optimization

• We can do slightly better if we treat pure register-to-register moves as a special case.

• For example:

  movq t1, t2
  ...
  t1
  ...
  t2

• By the normal rules, we would add an interference edge from t1 to t2 because the mov instruction defines t2 and t1 is live out. We don’t need that edge.
Computing the Interference Graph – Take 2

- For each Temp in the program, create a node
- For each non-move instruction, \texttt{instr}, in the program:
  
  for each \( d \) in \texttt{def}[\texttt{instr}]
  
  for each live in \texttt{out}[\texttt{instr}]
    
    if (live \(!=\) \( d \))
      
      add edge \( d \leftrightarrow\) live

- For each move instruction, \texttt{movq s, d}, in the program:
  
  for each live in \texttt{out}[\texttt{instr}]
    
    if (live \(!=\) d \&\& live \(!=\) s)
      
      add edge \( d \leftrightarrow\) live
Register Allocation

- The analysis we have done so far produces an interference graph
- How do we colour it with K 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 $< K$
   - 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)
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 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?
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.

```assembly
/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, and RV 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.
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?”