Administration

• Announcements
  – Final exam is scheduled: Apr 18 2018, 7:00pm

• Minijava project
  – Phase 3 due tomorrow
Dataflow Equations

- What is the relationship between $\text{in}[n]$ and $\text{out}[n]$ for a given instruction.

  \[ \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \]

- What is the relationship between in and out sets of a node and its predecessors and successors?

  \[ \text{out}[n] = \bigcup_{s \text{ in succ}[n]} \text{in}[s] \]
Look at liveness for live.java
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.
How to find which pairs of temps interfere?
Examining all pairs in in[n] and out[n]

• Does this work?
  – Mostly
  – Look at t007, t008 (instructions 14 and 18) in liveness for live.java

• Is there a better way?
  – This re-examines the same pairs of temps lots of times
When do registers (temps) get written to?

- By instructions
  - Each register changed by an instruction is in the def set
- Can we use that to focus our work of building the interference graph?
Computing the Interference Graph – version 2

• A better way of computing the interference graph:
  – 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, \texttt{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:
  
  \[
  \text{movq } t1, t2 \\
  \ldots \quad t1 \\
  \ldots \quad t2
  \]

- By the normal rules, we would add an interference edge from \( t1 \) to \( t2 \) because the \texttt{movq} instruction defines \( t2 \) and \( t1 \) is live out. We don’t need that edge. Why not?
Computing the Interference Graph – version 3

– 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}[instr]
    for each live in \texttt{out}[instr]
      if (live \neq d)
        add edge d <\rightarrow> live
– For each move instruction, \texttt{movq s, d}, in the program:
  for each live in \texttt{out}[instr]
    if (live \neq d \&\& live \neq s)
      add edge d <\rightarrow> live
Look at interference graphs

- run interf live.java
- show Test_do
- run interf BinarySearchCleanedup.java
- show BS_Search
- The analysis we have done so far produces an interference graph
- How do we colour it with K colours?
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