Global Register Allocation

(Slides from Andrew Myers)
Main idea

• Want to replace temporary variables with some fixed set of registers

• **First**: need to know which variables are live after each instruction
  – Two simultaneously live variables cannot be allocated to the same register
Register allocation

• For every node $n$ in CFG, we have $\text{out}[n]$
  – Set of temporaries live out of $n$
• Two variables *interfere* if
  – both initially live (i.e., function args), or
  – both appear in $\text{out}[n]$ for any $n$
• How to assign registers to variables?
Interference graph

- **Nodes** of the graph = variables
- **Edges** connect variables that interfere with one another
- Nodes will be assigned a **color** corresponding to the register assigned to the variable
- Two colors can’t be next to one another in the graph
## Interference graph

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Live vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = a + 2</td>
<td></td>
</tr>
<tr>
<td>c = b * b</td>
<td></td>
</tr>
<tr>
<td>b = c + 1</td>
<td></td>
</tr>
<tr>
<td>return b * a</td>
<td></td>
</tr>
</tbody>
</table>
Interference graph

Instructions	Live vars

b = a + 2

c = b * b

b = c + 1

return b * a
Interference graph

Instructions  Live vars
b = a + 2
b = c + 1
return b * a
### Interference graph

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Live vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b = a + 2 )</td>
<td>( b, a )</td>
</tr>
<tr>
<td>( c = b \times b )</td>
<td>( a, c )</td>
</tr>
<tr>
<td>( b = c + 1 )</td>
<td>( b, a )</td>
</tr>
<tr>
<td>return ( b \times a )</td>
<td></td>
</tr>
</tbody>
</table>
Interference graph

Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Live vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>b = a + 2</td>
<td>a</td>
</tr>
<tr>
<td>c = b * b</td>
<td>b, a</td>
</tr>
<tr>
<td>b = c + 1</td>
<td>a, c</td>
</tr>
<tr>
<td>return b * a</td>
<td>b, a</td>
</tr>
</tbody>
</table>
Interference graph

Instructions

- \( b = a + 2 \)
- \( c = b \times b \)
- \( b = c + 1 \)
- return \( b \times a \)

Live vars

- \( a \)
- \( a, b \)
- \( a, c \)
- \( a, b \)

Register allocation:

- eax
- ebx

Color

- None

Graph:

- Node a
- Node b
- Node c

Edges:

- a to b
- a to c
- b to c
Interference graph

Instructions

\[ b = a + 2 \]

\[ c = b \times b \]

\[ b = c + 1 \]

\[ \text{return } b \times a \]

Live vars

a

a,b

a,c

a,b

Live vars

\[ a \]

\[ a,b \]

\[ a,c \]

\[ a,b \]

Color register

\[ \text{eax} \]

\[ \text{ebx} \]
Graph coloring

• Questions:
  – Can we efficiently find a coloring of the graph whenever possible?
  – Can we efficiently find the optimum coloring of the graph?
  – How do we choose registers to avoid move instructions?
  – What do we do when there aren’t enough colors (registers) to color the graph?
Coloring a graph

• Kempe’s algorithm [1879] for finding a K-coloring of a graph

• Assume K=3

• Step 1 (simplify): find a node with at most $K-1$ edges and cut it out of the graph. (Remember this node on a stack for later stages.)
Coloring a graph

• Once a coloring is found for the simpler graph, we can always color the node we saved on the stack

• Step 2 (color): when the simplified subgraph has been colored, add back the node on the top of the stack and assign it a color not taken by one of the adjacent nodes
Coloring

stack:

```
  a
 / 
b   c
 / 
 d  e
```

color register

- eax
- ebx
Coloring

color register

eax
ebx

stack:
c

Monday, October 17, 2011
Coloring

Color: eax, ebx

Register: eax, ebx

Stack:
e

c
Coloring

color register

eax
ebx

color

stack:

a
e
c

d
b

a

b

c

e

d

Monday, October 17, 2011
Coloring

- eax (color: blue, register: eax)
- ebx (color: yellow, register: ebx)

Diagram:
- a connects to b and c
- b connects to d and e
- c connects to nothing

Stack:
- b
- a
- e
- c
Coloring

- ** eax **
- ** ebx **

```
stack:
  d
  b
  a
e
```
Coloring

- eax
- ebx

stack:
- b
- a
- e
- c
Coloring

color register

eax
ebx

stack:
e
c
Coloring

color register

eax

stack:

ebx
Failure

• If the graph cannot be colored, it will eventually be simplified to a graph in which every node has at least $K$ neighbors.
• Sometimes, the graph is still $K$-colorable!
• Finding a $K$-coloring in all situations is an NP-complete problem.
  – We will have to approximate to make register allocators fast enough.
Coloring

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
</tr>
</tbody>
</table>

a

b
d
e
c

stack:
Coloring

all nodes have 2 neighbours!
Coloring

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
</tr>
</tbody>
</table>

stack:

- b
- d
Coloring

color     register

eax

ebx

a

b

c

d

e

stack:

c
e
a
b

d
Coloring

color register

eax
ebx

stack:

e
a
b
d
Coloring

color    register

eax

ebx

stack:

a
b
d

da b
c
Coloring

color     register

eax

ebx

stack:

b
d
Coloring

color | register
---|---

eax

ebx

stack:
d

Monday, October 17, 2011
Coloring

We got lucky!

stack:

We got lucky!
Some graphs can’t be colored in \( K \) colors:
Some graphs can’t be colored in K colors:

**Coloring**

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td></td>
</tr>
<tr>
<td>ebx</td>
<td></td>
</tr>
</tbody>
</table>

Stack:

- b
- e
- a
- d
Some graphs can’t be colored in K colors:

The diagram shows a graph with nodes labeled A, B, C, D, and E. Nodes A and C are colored yellow, indicating they are in the same color set. Node B is colored blue, and nodes D and E are colored gray.

The stack contains elements in the order E, A, D.
Some graphs can’t be colored in K colors:

no colors left for e!
Spilling

- **Step 3 (spilling):** once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack
- There are many heuristics that can be used to pick a node
  - not in an inner loop
Spilling code

• We need to generate extra instructions to load variables from stack and store them
• These instructions use registers themselves. What to do?
  – Stupid approach: always keep extra registers handy for shuffling data in and out: what a waste!
  – Better approach: rewrite code introducing a new temporary; rerun liveness analysis and register allocation
    • Intuition: you were not able to assign a single register to the variable that was spilled but there may be a free register available at each spot where you need to use the value of that variable
Rewriting code

• Consider: add t1 t2
  – Suppose t2 is selected for spilling and assigned to stack location [ebp-24]
  – Invent new temporary t35 for just this instruction and rewrite:
    • mov t35, [ebp – 24];
    • add t1, t35
  – Advantage: t35 has a very short live range and is much less likely to interfere.
  – Rerun the algorithm; fewer variables will spill
Precolored Nodes

• Some variables are pre-assigned to registers
  – Eg: mul on x86/pentium
    • uses eax; defines eax, edx
  – Eg: call on x86/pentium
    • Defines (trashes) caller-save registers eax, ecx, edx

• Treat these registers as special temporaries; before beginning, add them to the graph with their colors
Precolored Nodes

• Can’t simplify a graph by removing a precolored node
• Precolored nodes are the starting point of the coloring process
• Once simplified down to colored nodes start adding back the other nodes as before
Optimizing Moves

- Code generation produces a lot of extra move instructions
  - mov t1, t2
  - If we can assign t1 and t2 to the same register, we do not have to execute the mov
  - Idea: if t1 and t2 are not connected in the interference graph, we coalesce into a single variable
Coalescing

- Problem: coalescing can increase the number of interference edges and make a graph uncolorable

- Solution 1 (Briggs): avoid creation of high-degree (>= K) nodes
- Solution 2 (George): a can be coalesced with b if every neighbour t of a:
  - already interferes with b, or
  - has low-degree (< K)
Simplify & Coalesce

- **Step 1 (simplify):** simplify as much as possible without removing nodes that are the source or destination of a move (*move-related nodes*)
- **Step 2 (coalesce):** coalesce move-related nodes provided low-degree node results
- **Step 3 (freeze):** if neither steps 1 or 2 apply, freeze a move instruction: registers involved are marked *not move-related* and try step 1 again
Overall Algorithm

1. Simplify, freeze and coalesce
2. Mark possible spills
3. Color & detect actual spills
4. Liveness
5. Rewrite code to implement actual spills