Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Register Allocation
Register Allocation

• Main Idea: We 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
Live variable analysis (or simply liveness analysis) is a classic data-flow analysis to calculate the variables that are live at each point in the program.

A variable is live at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to.

Analysis is performed starting from the end of the function working towards the beginning → backwards analysis.

Compute def(inition) – use(age) regions: A variable is live between its (most recent) definition and (last) use.
Liveness Analysis

Instructions    Live vars

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]
Instructions | Live vars
---|---
b = a + 2
\n\nc = b * b
\n\nb = c + 1
\n\nreturn b * a
Instructions

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]

Live vars

\[ a, c \]
\[ b, a \]

return \( b \times a \)
Liveness Analysis

Instructions | Live vars
--- | ---
b = a + 2 | b, a

--- | ---
c = b * b | a, c

--- | ---
b = c + 1 | b, a

--- | ---
return b * a |
## Liveness Analysis

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

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]
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 cannot be next to one another in the graph
Instructions

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]

Live vars

\[ a \]
\[ a, b \]
\[ a, c \]
\[ a, b \]
Instructions

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]

Live vars

\[ a \]
\[ a, b \]
\[ a, c \]
\[ a, b \]
Instructions | Live vars
---|---
b = a + 2 | a

c = b * b | a,b

b = c + 1 | a,c

return b * a | a,b
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

• **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

- **color**
  - eax
- **register**
  - ebx

**stack:**

```
(a, b, d, e, c)
```
Coloring

color register

eax

ebx

stack:
c

d
b

a

c

e
Coloring

- color
- register
  - eax
  - ebx

- stack:
  - e
  - c
Coloring

- Color:
  - eax
  - ebx

- Register:
  - Color
  - Register

- Stack:
  - a
  - e
  - c
Coloring

stack:
b
a
e
c

color register

eax
ebx
Coloring

color register

eax

ebx

stack:
d
b
a
e
c

UC Santa Barbara
Coloring

color register

eax

ebx

stack:

b

a
e
c

d

b

a
e
c
Coloring

- **color register**
  - eax
  - ebx

- **stack:**
  - a
  - e
  - c
Coloring

stack:

e

c

color register

eax

ebx
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:

c

d e b a c
Coloring

color register

eax

 ebx

stack:

a

b

d
c

e

uc santa barbara
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

Stack:

color     register

eax

ebx
All nodes have (at least) 2 neighbors!
Coloring

stack:

b
d
Coloring

- color: eax
- register: ebx

Diagram:

- Stack: c e a b d
Coloring

stack:

e
a
b
d

color register

eax

ebx

a

b
d

e

c


Coloring

- **color**
- **register**
  - eax
  - ebx

**stack:**
- a
- b
- d
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
d
Coloring

color register

eax

ebx

stack:

d

d
Coloring

We got lucky!

stack:

d

b

c

We got lucky!
Some graphs cannot be colored in K colors
Some graphs cannot be colored in $K$ colors
Some graphs cannot be colored in $K$ colors

stack:

- $e$
- $a$
- $d$
Some graphs cannot be colored in K colors

No colors left for e!

stack:

e
a
d
• **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
• We need to generate extra instructions to load variables from stack and store them

• These instructions use registers themselves. What to do?
  – Naive approach: always keep extra registers handy for shuffling data in and out: What a waste!
  – Better approach: ?
Spilling Code

- We need to generate extra instructions to load variables from stack and store them.

- These instructions use registers themselves. What to do?
  - **Naive 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.
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 (overwrites) caller-save registers `eax`, `ecx`, `edx`

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

• Cannot 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
Summary

- Register allocation has three major parts
  - Liveness analysis
  - Graph coloring
  - Program transformation (spilling and optimizations)