Lecture 6
Register Allocation

I. Introduction
II. Abstraction and the Problem
III. Algorithm

Reading: Chapter 8.8.4
Before next class: Chapter 10.1 - 10.2
• **Moving from machine independent (mostly) to machine dependent**
Pseudo Registers

• Code Selection
  • Machine instructions using pseudo registers

• Pseudo registers
  • Like machine registers, but infinite number
  • No aliasing

```c
int g;
int foo(int a, int b)
{
    if (a+b) {
        g++;
        bar();
        g--;
    }
    return a+b+g;
}
```

```c
preg_a = reg2;
preg_b = reg3;
preg_0 = &g;
preg_1 = preg_a + preg_b;
preg_g = *preg_0;
if(preg_1 == 0) goto Lab;
preg_2 = preg_g + 1;
*preg_0 = preg_2;
bar();
preg_3 = *preg_0;
preg_g = preg_3 - 1;
*preg_0 = preg_g;
Lab :;
reg2 = preg_1 + preg_g;
return ;
```
I. Motivation

• Problem
  – Allocation of variables (pseudo-registers) to hardware registers in a procedure

• Perhaps the most important optimization
  – Directly reduces running time
    • (memory access $\rightarrow$ register access)
  – Other optimizations heavily interact
    • PRE
    • Scheduling
    • Loop optimizations
Goal

• Find an assignment for all pseudo-registers, if possible.
• If there are not enough registers in the machine, choose registers to spill to memory
Example

A = ...
IF A goto L1

L1:
... = A
C = ...
D = ...
... = C + D

B = ...
... = A
D = ...
... = B + D

... = D
II. An Abstraction for Allocation & Assignment

• Intuitively
  – Two pseudo-registers interfere if at some point in the program they cannot both occupy the same register.

• Interference graph: an undirected graph, where
  – nodes = pseudo-registers
  – there is an edge between two nodes if their corresponding pseudo-registers interfere

• What is not represented
  – Extent of the interference between uses of different variables
  – Where in the program is the interference
Register Allocation and Coloring

• A graph is **n-colorable** if:
  – every node in the graph can be colored with one of the n colors such that two adjacent nodes do not have the same color.

• **Assigning n register (without spilling) = Coloring with n colors**
  – assign a node to a register (color) such that no two adjacent nodes are assigned same registers(colors)

• Is spilling necessary? = Is the graph n-colorable?

• **To determine if a graph is n-colorable is NP-complete, for n>2**
  • Too expensive
  • Heuristics
III. Algorithm

Step 1. Build an interference graph
   a. refining notion of a node
   b. finding the edges

Step 2. Coloring
   – use heuristics to try to find an $n$-coloring
     • Success:
       – colorable and we have an assignment
     • Failure:
       – graph not colorable, or
       – graph is colorable, but it is too expensive to color
Step 1a. Nodes in an Interference Graph

A = ...
IF A goto L1

B = ...
  = A
D =
  = B + D

L1: C = ...
  = A
D =
  = D + C

A = 2

= A
Live Ranges and Merged Live Ranges

• Motivation: to create an interference graph that is easier to color
  – Eliminate interference in a variable’s “dead” zones.
  – Increase flexibility in allocation:
    • can allocate same variable to different registers

• A live range consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.
  – How to compute a live range?

• Two overlapping live ranges for the same variable must be merged
Example (Revisited)

\[
\begin{align*}
A &= \ldots \quad (A_1) \\
\text{IF } A \text{ goto } L1
\end{align*}
\]

\[
\begin{align*}
B &= \ldots \\
  &= A \\
D &= (D_2) \\
  &= B + D
\end{align*}
\]

\[
\begin{align*}
L1: C &= \ldots \\
  &= A \\
D &= (D_1) \\
  &= D + C
\end{align*}
\]

\[
\begin{align*}
A &= (A_2) \\
  &= D
\end{align*}
\]

\[
\begin{align*}
{} & {} \\
\{A\} & \{A_1\} \\
\{A,B\} & \{A_1,B\} \\
\{B\} & \{A_1,B\} \\
\{B,D\} & \{A_1,B,D_2\} \\
\{D\} & \{A_1,B,D_2\} \\
{} & \{A_2,B,C,D_1,D_2\}
\end{align*}
\]

\[
\begin{align*}
\text{liveness} & \quad \text{reaching-def} \\
\{\} & \{\} \\
\{A\} & \{A_1\} \\
\{A\} & \{A_1\} \\
\{A\} & \{A_1\} \\
\{A,C\} & \{A_1,C\} \\
\{C\} & \{A_1,C\} \\
\{C,D\} & \{A_1,C,D_1\} \\
\{D\} & \{A_1,C,D_1\} \\
\{D\} & \{A_1,B,C,D_1,D_2\} \\
\{A,D\} & \{A_2,B,C,D_1,D_2\} \\
\{A\} & \{A_2,B,C,D_1,D_2\} \\
\{A\} & \{A_2,B,C,D_1,D_2\}
\end{align*}
\]

(Does not use \(A, B, C, \) or \(D\).)
Merging Live Ranges

- **Merging definitions into equivalence classes**
  - Start by putting each definition in a different equivalence class
  - For each point in a program:
    - if (i) variable is live, and (ii) there are multiple reaching definitions for the variable, then:
      - merge the equivalence classes of all such definitions into one equivalence class

- From now on, refer to merged live ranges simply as live ranges
Step 1b. Edges of Interference Graph

- **Intuitively:**
  - Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.
  - **Algorithm:**
    - At each point in the program:
      - enter an edge for every pair of live ranges at that point.

- **An optimized definition & algorithm for edges:**
  - **Algorithm:**
    - check for interference only at the start of each live range
    - Faster
    - Better quality
Example 2

IF Q goto L1

A = ...

L1: B = ...

IF Q goto L2

A = ...

L2: ... = B
Step 2. Coloring

• Reminder: coloring for $n > 2$ is NP-complete

• Observations:
  – a node with degree $< n$ ⇒
    • can always color it successfully, given its neighbors’ colors

  – a node with degree $= n$ ⇒

  – a node with degree $> n$ ⇒
Coloring Algorithm

- **Algorithm:**
  - Iterate until stuck or done
    - Pick any node with degree < n
    - Remove the node and its edges from the graph
  - If done (no nodes left)
    - reverse process and add colors
- **Example (n = 3):**

```
  B
 / \  
E  A   C
 /     \\  
D      D
```

- Note: degree of a node may drop in iteration
- Avoids making arbitrary decisions that make coloring fail
What Does Coloring Accomplish?

• **Done:**
  – colorable, also obtained an assignment

• **Stuck:**
  – colorable or not?

![Diagram showing nodes A, B, C, D, E connected in a network]
What if Coloring Fails?

- Use heuristics to improve its chance of success and to spill code

Build interference graph

Iterative until there are no nodes left
   If there exists a node $v$ with less than $n$ neighbors
      place $v$ on stack to register allocate
   else
      $v$ = node chosen by heuristics
      (least frequently executed, has many neighbors)
      place $v$ on stack to register allocate (mark as spilled)
      remove $v$ and its edges from graph

While stack is not empty
   Remove $v$ from stack
   Reinsert $v$ and its edges into the graph
   Assign $v$ a color that differs from all its neighbors
   (guaranteed to be possible for nodes not marked as spilled)
Summary

• Problems:
  – Given n registers in a machine, is spilling avoided?
  – Find an assignment for all pseudo-registers, whenever possible.

• Solution:
  – Abstraction: an interference graph
    • nodes: live ranges
    • edges: presence of live range at time of definition
  – Register Allocation and Assignment problems
    • equivalent to n-colorability of interference graph
      ➔ NP-complete
  – Heuristics to find an assignment for n colors
    • successful: colorable, and finds assignment
    • not successful: colorability unknown & no assignment
Extra

• **Scope of Problem:**
  - Example Showed Whole Function
  - Separate Inner Loops
    - Interaction with scheduling
    - Don’t want to spill in the inner loop
  - Inter-procedural

• Preferencing
  - Avoid copies on function calls, entries and returns
  - Glue together inner loop allocation with global

• Homing
• Rematerialization
**Homing**

P1 = load A

P1 = ...

... = P1

Store P1 into A
Rematerialization

\[ a = \ldots \]
\[ b = \ldots \]
\[ t = a + b \]
\[ \ldots \]
\[ \ldots = t \]
Interaction With Scheduling

\[
A = \frac{C}{3} \\
E = A + 1 \\
B = \frac{D}{3} \\
F = B + 1
\]

Vs

\[
A = \frac{C}{3} \\
B = \frac{D}{3} \\
E = A + 1 \\
F = B + 1
\]
Interaction with Loop Optimization

for $i = 0$ to $n$
  for $j = 0$ to $n$
    $a[i][j] += b[i][j] \times (c[i] + c[i+1])$

Versus

for $i = 0$ to $n$ by 2
  for $j = 0$ to $n$
    $a[i][j] += b[i][j] \times (c[i] + c[i+1])$
    $a[i+1][j] += b[i+1][j] \times (c[i+1] + c[i+2])$

In the first loop we have two loads of “c” per multiply. In the second we have 1.5 per multiply. However, in the second loop we need three registers to hold “c” instead of two.