Lecture 15

Register Allocation & Spilling

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

Reading: ALSU 8.8.4

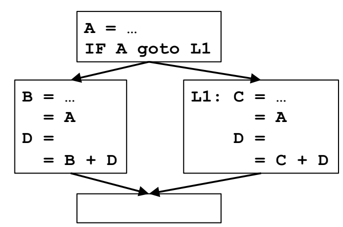
I. Motivation

- **Problem**
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- **A very important optimization!**
  - Directly reduces running time
    - (memory access → register access)
  - Useful for other optimizations
    - e.g. CSE assumes old values are kept in registers.

Goals

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

Example



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

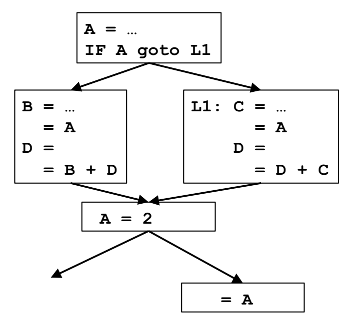
Step 1. Build an interference graph

a. refining notion of a node
b. finding the edges

Step 2. Coloring

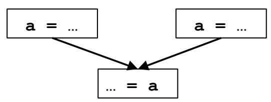
- use heuristics to try to find an n-coloring
  - Success:
    - colorable and we have an assignment
  - Failure:
    - graph not colorable, or
    - graph is colorable, but it is too expensive to color

Step 1a. Nodes in an Interference Graph

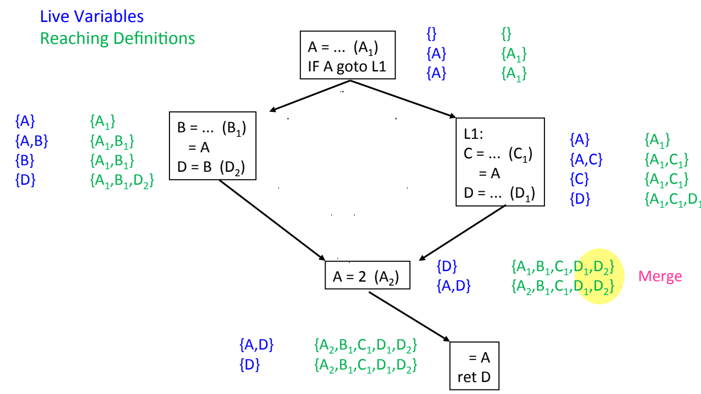


Live Ranges and Merged Live Ranges

- **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)



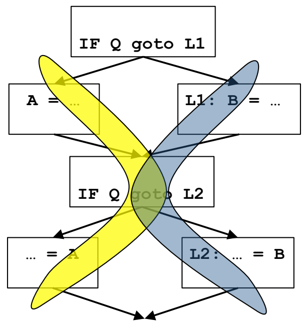
Merging Live Ranges

- **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**
  – merged live ranges are also known as "webs"

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

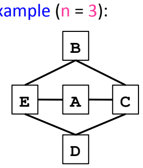


Step 2. Coloring

- **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

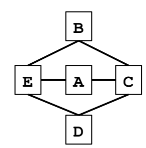
- Algorithm:
  - Iterate until stuck or done
    - Pick any node with degree < n
    - Remove the node and its edges from the graph
  - If done (no nodes left)
    - reverse process and add colors
- Example (n = 3):



- 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?



Extending Coloring: Design Principles

- **A pseudo-register is**
  - Colored successfully: allocated a hardware register
  - Not colored: left in memory
- **Objective function**
  - Cost of an uncolored node:
    - proportional to number of uses/definitions (dynamically)
    - estimate by its loop nesting
  - Objective: minimize sum of cost of uncolored nodes
- **Heuristics**
  - Benefit of spilling a pseudo-register:
    - increases colorability of pseudo-registers it interferes with
    - can approximate by its degree in interference graph
  - Greedy heuristic
    - spill the pseudo-register with lowest cost-to-benefit ratio, whenever spilling is necessary

Spilling to Memory

- CISC architectures
  - can operate on data in memory directly
  - memory operations are slower than register operations
- RISC architectures
  - machine instructions can only apply to registers
  - Use
    - must first load data from memory to a register before use
  - Definition
    - must first compute RHS in a register
    - store to memory afterwards
  - Even if spilled to memory, needs a register at time of use/definition

Review: Coloring Algorithm (Without Spilling)

- **Attempt to Color Graph**

```
Build interference graph
Iterate until there are no nodes left
    If there exists a node v with less than n neighbor
        place v on stack to register allocate
    else
        return (coloring heuristics fail)
    remove v and its edges from graph
```

- **Assign registers**

```
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
```


Chaitin: Coloring and Spilling

- **Identify spilling**

```
Build interference graph
Iterate until there are no nodes left
    If there exists a node v with less than n neighbor
        place v on stack to register allocate
    else
        v = node with highest degree-to-cost ratio
        mark v as spilled
    remove v and its edges from graph
```

- **Spilling may require use of registers; change interference graph**

```
While there is spilling
    rebuild interference graph and perform step above
```

- **Assign registers**

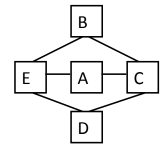
```
While stack is not empty
    Remove v from stack
    Reinsert v and its edges into the graph
    Assign v a color that differs from all its neighbors
```

Spilling

- What should we spill?
  - Something that will eliminate a lot of interference edges
  - Something that is used infrequently
  - Maybe something that is live across a lot of calls?
- One Heuristic:
  - spill cheapest live range (aka "web")
  - Cost = [(# defs & uses)*$10^{\text{loop-nest-depth}}$]/degree

Quality of Chaitin's Algorithm

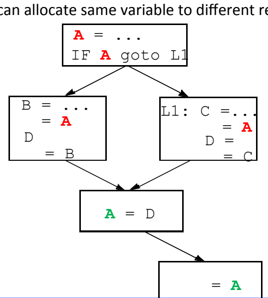
- Giving up too quickly

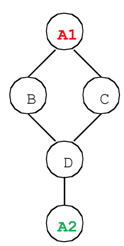


- An optimization: "Prioritize the coloring"
  - Still eliminate a node and its edges from graph
  - Do not commit to "spilling" just yet
  - Try to color again in assignment phase.

Splitting Live Ranges

- Recall: Split pseudo-registers into live ranges 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





Insight

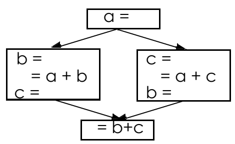
- Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
  - Eliminate interference in a variable's "nearly dead" zones.
    - Cost: Memory loads and stores
      - Load and store at boundaries of regions with no activity
    - # active live ranges at a program point can be > # registers
  - Can allocate same variable to different registers
    - Cost: Register operations
      - a register copy between regions of different assignments
    - # active live ranges cannot be > # registers

Examples

Example 1:

```
FOR i = 0 TO 10
   FOR j = 0 TO 10000
      A = A + ...
      (does not use B)
   FOR j = 0 TO 10000
      B = B + ...
      (does not use A)
```

Example 2:



Live Range Splitting

- When do we apply live range splitting?
- Which live range to split?
- Where should the live range be split?
- How to apply live-range splitting with coloring?
  - Advantage of coloring:
    - defers arbitrary assignment decisions until later
  - When coloring fails to proceed, may not need to split live range
    - degree of a node >= n does not mean that the graph definitely is not colorable
  - Interference graph does not capture positions of a live range

One Algorithm

- Observation: spilling is absolutely necessary if
  - number of live ranges active at a program point > n
- Apply live-range splitting before coloring
  - Identify a point where number of live ranges > n
  - For each live range active around that point:
    - find the outermost "block construct" that does not access the variable
  - Choose a live range with the largest inactive region
  - Split the inactive region from the live range

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