Advanced Compilers
CMPSCI 710
Spring 2003
Register Allocation

Emery Berger
University of Massachusetts, Amherst

The Memory Hierarchy
- Higher = smaller, faster, closer to CPU
  - A real desktop machine (mine)
  - 8 integer, 8 floating-point; 1-cycle latency
  - 8K data & instructions; 2-cycle latency
  - 512K; 7-cycle latency
  - 1GB; 100 cycle latency
  - 40 GB; 38,000,000 cycle latency (!)

Managing the Memory Hierarchy
- Programmer view: only two levels of memory
  - Main memory (stores & loads)
  - Disk (file I/O)
- Two things maintain this abstraction:
  - Hardware
    - Moves data between memory and caches
  - Compiler
    - Moves data between memory and registers

Overview
- Introduction
- Register Allocation
  - Definition
  - History
  - Interference graphs
  - Graph coloring
  - Register spilling

Register Allocation: Definition
- Register allocation assigns registers to values
  - Candidate values:
    - Variables
    - Temporaries
    - Large constants
    - When needed, spill registers to memory
  - Important low-level optimization
    - Registers are 2x – 7x faster than cache
    - Judicious use ⇒ big performance improvements

Register Allocation: Complications
- Explicit names
  - Unlike all other levels of hierarchy
  - Scarce
  - Small register files (set of all registers)
  - Some reserved by operating system
    - e.g., “BP”, “SP”...
- Complicated
  - Weird constraints, esp. on CISC architectures
  - Special registers: zero-load
History
- As old as intermediate code
  - Used in the original FORTRAN compiler (1950's)
  - Very crude algorithms
- No breakthroughs until 1981!
  - Chaitin invented register allocation scheme based on graph coloring
  - Equivalence first noted by Cocke et al., 1971
  - Simple heuristic, works well in practice

Register Allocation Example
- Consider this program with six variables:
  \[ a := c + d \]
  \[ e := a + b \]
  \[ f := e - 1 \]
- with the assumption that \( a \) and \( e \) die after use
- Temporaries can be “reused” after \( e := a + b \)
- Same with temporary \( e \)
- Can allocate \( a \), \( e \), and \( f \) all to one register \( (r_1) \):
  \[
  r_1 := r_2 + r_3 \\
  r_1 := r_1 + r_4 \\
  r_1 := r_1 - 1
  \]

Basic Register Allocation Idea
- Value in dead temporary not needed for rest of the computation
  - Dead temporary can be reused
- Basic rule:
  - Temporaries \( t_1 \) and \( t_2 \) can share same register if at any point in the program at most one of \( t_1 \) or \( t_2 \) is live!

Algorithm: Part I
- Compute live variables for each point:
  \[
  \begin{align*}
  (a, c, f) & \quad b := b + c \\
  (c, d, f) & \quad d := a - c \\
  (c, e) & \quad e := d + f \\
  (c, f) & \quad f := 2 \times e \\
  (b, c, e, f) & \quad b := d + e \\
  (c, e) & \quad e := e - 1 \\
  (c, f) & \quad f := e + c \\
  (b) & \quad (b)
  \end{align*}
  \]

Register Interference Graph
- Two temporaries live simultaneously
  - Cannot be allocated in the same register
- Construct register interference graph
  - Node for each temporary
  - Undirected edge between \( t_1 \) and \( t_2 \)
  - If live simultaneously at some point in the program
- Two temporaries can be allocated to same register if no edge connects them

Register Interference Graph: Example
- For our example:
  \[
  \begin{align*}
  (b, c, f) & \quad (a, c, f) \\
  (c, d, f) & \quad (c, d, e, f) \\
  (c, e) & \quad (b, c, e, f) \\
  (c, f) & \quad (b)
  \end{align*}
  \]
Register Interference Graph: Properties

- Extracts exactly the information needed to characterize legal register assignments
- Gives global picture of register requirements
  - Over the entire flow graph
  - After RIG construction, register allocation is architecture-independent
  - Add additional edges in RIG to encode architectural intricacies
- Now what do we do with this graph?

Graph Coloring

- Graph coloring: assignment of colors to nodes
  - Nodes connected by edge have different colors
  - Equivalently: no adjacent nodes have same color
- Graph k-colorable = can be colored with k colors

Register Allocation Through Graph Coloring

- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (temporaries)
  - Let k = number of machine registers
- If the RIG is k-colorable, there’s a register assignment that uses no more than k registers

Graph Coloring Example

- Consider the example RIG

There is no coloring with fewer than 4 colors
There are 4-colorings of this graph

Graph Coloring Example, Continued

- Under this coloring the code becomes:

Computing Graph Colorings

- How do we compute coloring for interference graph?
  - NP-hard!
  - For given # of registers, coloring may not exist
- Solution
  - Use heuristics (here, Briggs)
Graph Coloring Heuristic

- Observation: “degree < k rule”
  - Reduce graph:
    - Pick node t with < k neighbors in RIG
    - Eliminate t and its edges from RIG
    - If the resulting graph has k-coloring, so does the original graph

- Why?
  - Let c₁,...,cₙ be colors assigned to neighbors of t in reduced graph
  - Since n < k, we can pick some color for t different from those of its neighbors

Graph Coloring Heuristic, Continued

- Heuristic:
  - Pick node t with fewer than k neighbors
  - Put t on a stack and remove it from the RIG
  - Repeat until the graph has one node
  - Start assigning colors to nodes on the stack (starting with the last node added)
  - At each step, pick color different from those assigned to already-colored neighbors

Graph Coloring Example (1)

- Start with the RIG and with k = 4:
  - Stack: {}
  - Remove a and then d

Graph Coloring Example (2)

- Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f
  - Stack: {d, a}

Graph Coloring Example (2)

- Start assigning colors to: f, e, b, c, d, a

What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors?
  - Example: try to find a 3-coloring of the RIG:
What if the Heuristic Fails?

- Remove a and get stuck (as shown below)
- Pick a node as a candidate for spilling
- Assume that f is picked

What if the Heuristic Fails?

- Remove f and continue the simplification
- Simplification now succeeds: b, d, e, c

What if the Heuristic Fails?

- During assignment phase, we get to the point when we have to assign a color to f
- Hope: among the 4 neighbors of f, we use less than 3 colors ⇒ optimistic coloring

Spilling

- Optimistic coloring failed ⇒ must spill temporary f
- Allocate memory location as home of f
- Typically in current stack frame
- Call this address fa
- Before each operation that uses f, insert
  \[ f := \text{load } fa \]
- After each operation that defines f, insert
  \[ \text{store } f, fa \]

Recomputing Liveness Information

- New liveness information after spilling:
Recomputing Liveness Information

- New liveness info almost as before, but:
  - \( f \) is live only
  - Between \( f := \text{load fa} \) and the next instruction
  - Between \( \text{store f, fa} \) and the preceding instruction

- Spilling reduces the live range of \( f \)
  - Reduces its interferences
  - Results in fewer neighbors in RIG for \( f \)

Recompute RIG After Spilling

- Remove some edges of spilled node
- Here, \( f \) still interferes only with \( c \) and \( d \)
  - Resulting RIG is 3-colorable

Spilling, Continued

- Additional spills might be required before coloring is found
- Tricky part: deciding what to spill
  - Possible heuristics:
    - Spill temporaries with most conflicts
    - Spill temporaries with few definitions and uses
    - Avoid spilling in inner loops
    - All are “correct”

Conclusion

- Register allocation: “must have” optimization in most compilers:
  - Intermediate code uses too many temporaries
  - Makes a big difference in performance
- Graph coloring:
  - Powerful register allocation scheme

Next Time

- Scheduling
- Read ACDI Chapter 17