CS412/413

Introduction to Compilers
Radu Rugina

Lecture 30: Register Allocation
9 Apr 03

Variables vs. Registers/Memory

- Difference between IR and assembly code:
  - IR (and abstract assembly) manipulate data in local and temporary variables
  - Assembly code manipulates data in memory/registers
- During code generation, compiler must account for this difference
- Compiler backend must allocate variables to memory or registers in the generated assembly code

Simple Approach

- Straightforward solution:
  - Allocate each variable on stack
  - At each instruction, bring values needed into registers, perform operation, then store result to memory

\[
\begin{align*}
x &= y + z \\
mov 16(\%ebp), \%eax \\
mov 20(\%ebp), \%ebx \\
add \%ebx, \%eax \\
mov \%eax, 24(\%ebx)
\end{align*}
\]

- Problem: program execution very inefficient
  - Moving data back and forth between memory and registers

Register Allocation

- Better approach = register allocation: keep variable values in registers as long as possible
- Best case: keep a variable’s value in a register throughout the lifetime of that variable
  - In that case, we don’t need to store it in memory
  - We say that the variable has been allocated in a register
  - Otherwise allocate variables on stack
  - We say that variable is spilled to memory
- Which variables can we allocate in registers?
  - Depends on the number of registers in the machine
  - Depends on how variable values are being used
- Main idea: cannot allocate two variables to the same register if they are both live at some program point

Register Allocation Algorithm

Hence, basic algorithm for register allocation is:

1. Perform live variable analysis
2. Inspect live variables at each program point
3. If two variables are in same live set, can’t be allocated to the same register – they interfere with each other

How do we determine register assignments next?

Interference Graph

- Nodes = program variables
- Edges = connect variables that interfere with each other

\[
\begin{align*}
b &= a + z \\
c &= b^2 b; \quad (a,b) \\
b &= c + 1; \quad (a,c) \\
\text{return } b^2 a; \quad (a,b)
\end{align*}
\]

- Register allocation = graph coloring

\[-
\begin{align*}
eax \\
ebx \\
\end{align*}
\]
Graph Coloring

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

Coloring a Graph

- Assume K = number of registers (take K=3)
- Try to color graph with K colors
- Key operation = Simplify: find some node with at most K-1 edges and cut it out of the graph

Coloring a Graph

- Idea: once coloring is found for simplified graph, removed node can be colored using free color
- Algorithm: simplify until graph contain no nodes
- unwind adding nodes back & assigning colors

Stack Algorithm

- Phase 1: Simplification
  - Repeatedly simplify graph
  - When remove a variable (i.e., graph node), push it on a stack

- Phase 2: Coloring
  - Unwind stack and reconstruct the graph as follows:
  - Pop variable from the stack
  - Add it to the graph
  - Color the node for that variable

Stack Algorithm

- Example:

- ...how about:

Failure of Heuristic

- If graph cannot be colored, it will reduce to a graph in which every node has at least K neighbors
- May happen even if graph is colorable in K!
- Finding K-coloring is NP-hard problem (requires search)
**Spilling**
- Once all nodes have K or more neighbors, pick a node and mark it for spilling (storage on stack).
- Remove it from graph, push it on stack.
- Try to pick node not used much, not in inner loop.

![Spilling Diagram]

**Optimistic Coloring**
- Spilled node may be K-colorable.
- Try to color it when popping the stack.
- If not colorable, actual spill: assign it a stack location.

![Optimistic Coloring Diagram]

**Accessing Spilled Variables**
- Need to generate additional instructions to get spilled variables out of stack and back in again.
- Naive approach: always keep extra registers handy for shuttling data in and out.
- Better approach: rewrite code introducing a new temporary, rerun liveness analysis and register allocation.

![Accessing Spilled Variables Diagram]

**Rewriting Code**
- Example: add v1, v2.
- Suppose that v2 is selected for spilling and assigned to stack location [ebp-24].
- Add new variable t35 for just this instruction, rewrite:
  - mov –24(%ebp), t35
  - add v1, t35
- Advantage: t35 has short lifetime and doesn’t interfere with other variables as much as v2 did.
- Now rerun algorithm; fewer or no variables will spill.

![Rewriting Code Diagram]

**Putting Pieces Together**
- Simplify
- Potential Spill
- Optimistic coloring
- Actual Spill

\{ Simplification \}
\{ Coloring \}

**Precolored Nodes**
- Some variables are pre-assigned to registers.
- mul instruction has: use[I] = eax, def[I] = { eax, edx }.
- call instruction kills caller-save regs: def[I] = { eax, ecx, edx }.
- To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes.

![Precolored Nodes Diagram]
Precolored Nodes

- Can’t simplify graph by removing a precolored node
- Precolored nodes: starting point of coloring process
- Once simplified graph is all colored nodes, add other nodes back in and color them

Optimizing Move Instructions

- Code generation produces a lot of extra mov instructions
  - `mov t5, t9`
- If we can assign t5 and t9 to same register, we can get rid of the mov
- Idea: if t5 and t9 are not connected in inference graph, coalesce them into a single variable; the move will be redundant.

Coalescing

- When coalescing nodes, take union of edges
- Hence, coalescing results in high-degree nodes
- Problem: coalescing nodes can make a graph uncolorable

Conservative Coalescing

- Conservative = ensure that coalescing doesn’t make the graph non-colorable (if it was colorable before)
- Approach 1: coalesce a and b if resulting node ab has less than K neighbors of significant degree
  - Safe because we can simplify graph by removing neighbors with insignificant degree, then remove coalesced node and get the same graph as before
- Approach 2: coalesce a and b if for every neighbor of t of a: either t already interferes with b; or t has insignificant degree
  - Safe because removing insignificant neighbors with coalescing yields a subgraph of the graph obtained by removing those neighbors without coalescing

Simplification + Coalescing

- Consider M = set of move-related nodes (which appear in the source or destination of a move instruction) and N = all of the other variables
- Start by simplifying as many nodes as possible from N
  - Coalesce some pairs of move-related nodes using conservative coalescing; delete corresponding mov instruction(s)
- Coalescing gives more opportunities for simplification: coalesced nodes may be simplified
- If can neither simplify nor coalesce, take a node in M and freeze all the move instruction involving that variable; go back to simplify.
- If all nodes frozen, no simplify possible, spill a variable

Full Algorithm

- **Simplify**
  - Coalesce
  - Freeze
- **Optimistic Spill**
  - **Actual Spill**
  - Coloring