Using Registers

CS 4447 / CS 9545

Stephen M. Watt

Register Allocation and Assignment

- Regiser *Allocation*: deciding which variables will use registers at which points in the program.

- Register *Assignment*: decide which registers will be used for which variables.

Complications

- Typically some registers are used by each architecture for specific purposes.
  Frame ptr, Stack ptr.

- Registers might be of different classes.
  Address registers, Floating point registers,
  General purpose registers

- Some instructions might use certain specific registers.

- Some instructions operate on register pairs.
  E.g. Register *n* and register *n*+1.

Some Observations

- Good use of registers (or of memory slots for temporaries) requires knowing when locations have “live” values.
  I.e. values that are current and later used.

- When a value is “moved” from one location to another, it is not really moved. It is copied.
  The value at both the old location and the new location are available.
  The old location may no longer contain the value of a variable if
  (1) the location has a new value placed in it, or
  (2) the variable’s value is modified in its new location.

Register and Address Descriptors

- Register descriptor:
  keeps track of what is currently in each register.

  E.g. Register 0 holds value of x, Register 1 of y and (copy) z.

- Address descriptor:
  keeps track of all locations where the current value of a variable (or temporary) can be found.

  E.g. The value of x is located currently at this locations a1 and a2 in memory, in register R7 and on the stack.

Next Use Information

- **If** statement *i* assigns x and statement *j* uses x and control can flow from statement *i* to statement *j* along a path that does not assign to x, **then** statement *j* **uses** the value of x computed at statement *i.*

- Within a basic block, we can easily determine the *next* use of a variable at any given point.

  We can also determine if it there is no next use.

A Simple *GetReg* Function for BBs

Find a location L for x in the statement x = y op z.

1. If y is in a register that holds the value of no other names, and
   y has no next use after this statement, then
   return the register of y as L.
   Update address descriptor of y to no longer include L.
2. Else: If there is an empty register, return that as L.
3. Else: If x has a next use or if “op” requires a register, then
   - Find an occupied register R.
   - Store the value of R into a memory location M if it is not already there.
   - Update the address descriptor for M and return R.
   - If R holds the value of may variables do this for each variable.

   A good choice of R would be one whose value
   does not need to be stored.
4. If x is not used in the block, or no register can be found,
   select the memory location of x as L.

Assumptions

- Can leave values in registers as long as possible, storing only if the register is needed for some other operation.
- Values are stored before exit of basic block.

- For debugging we may want to store user-level variables (but not generated temporaries) immediately after they are computed.

Cost Estimation

- We can assume a cost model where accessing memory costs *m* times as much as accessing a register. Say *m* = 2.

- Then consider alternative to find minimum cost.

Code Generation for Expressions

- Label each node with the number of registers needed to evaluate it without storing any values to memory.
  - Label each leaf 1.
  - Label for unary operators is the same as label for its operand.
  - Label for binary operator with operand labels L1 and L2 is max(L1,L2) if different, L1+1 if same.

Code Generation from Labelled Tree

- Input: Labelled tree with no common sub-expressions.
- Output: Optimal sequence to evaluate expr into a register.
  Use registers $R_1,...,R_N$.

0. Start with b = 1.
1. For interior node with label k and equal children (therefore with k-1)
   a) Recursively generate code for right child, starting with base b+1. Result is in R_{b+k}.
   b) Recursively generate code for left child, starting with base b. Result is in R_{b+k-1}.
   c) Generate $R_{b+k} := R_{b+k-1}$ OP R_{b+k}.
2. For an interior node with label k and unequal children, a big one with label k and a small one with label m, m < k.
   a) Recursively generate code for the big child, using base b. Result is in R_{b+k-1}.
   b) Recursively generate code for the small child, using base b. Result is in R_{b+m-1}.
   c) Generate $R_{b+k-1} := R_{b+k-1}$ OP R_{b+m-1} **or** $R_{b+k-1} := R_{b+m-1}$ OP R_{b+k-1} as appropriate.
3. For a leaf representing operand x, with base b, generate R_b := x.

If There are Not Enough Registers

- Input: Labelled tree. No common sub-expres. No *r* of registers.
- Output: Optimal sequence of machine instructions for *r* regsiters.

As before, but for interior nodes with label > r need to spill.

1. Interior node with one or more children with label > *r.*
2. Recursively generate code for the big child using *b = 1.*
   Result will be in R_r.
3. Generate $Temp_k := R_r$. ($Temp_k$ is used to help eval nodes with label k)
4. Generate code for the little child.
   a) If it has label *r* or greater, pick base b = 1 else if it has label j< r pick base b = r-j.
   b) Recursively apply algorithm to little child. Result is in R_r.
5. Generate $R_{r-1} := Temp_k$.
6. Generate $R_r := R_r$ OP R_{r-1} **or** $R_r := R_{r-1}$ OP R_r as appropriate

Global Register Allocation

- Want to avoid storing all variables at the end of each block.

- Approach of Fortran H:

  Reserve some number of registers for the most active variables used in an inner loop.

  Move them into registers in the loop pre-header.

- To choose which variables to store, use the cost model.

Register Allocation by Graph Coloring

- Used to manage spills. Two passes.
- Pass 1. Assume an infinite number of registers.
- Pass 2. For each procedure construct a "register interference graph"
  - An edge connects two nodes if one is live at the time the other is defined.
  - Attempt to color the graph with *r* colors. Each color represents a register. Then no two interfering"symbolic" registers are ever assigned the same real register.

Static Single Assignment

(something completely different)

- Make all LHS unique.

  Combine with φ functions.

```
if (cond) x = 10; else x = 20;
```

  becomes

```
if (cond) x1 = 10; else x2 = 20;
x3 = φ(x1, x2)
```