Class 42: Register Allocation

Held: Friday, 9 December 2011

Summary: Today we consider how to assign registers to temporaries.

Related Pages:

- EBoard.
- Reading: Aho et al. 8.9-8.10.

Overview:

- The Problem of Register Allocation
- A Basic Algorithm
- Detour: NP Completeness
- Graph Coloring
- Coalescing
- The Algorithm, Revisited

Register Allocation

- We are finally ready to worry about a problem we knew we would need to handle: converting the many temporaries we created to a fixed number of registers (along with some accompanying stack space).
- There are two related goals in register allocation:
  - Use only the registers available in the machine
  - Whenever possible, store both sides of a MOVE in the same register, so that the MOVE can be eliminated.
- There are programs in which it is not possible to convert all of the temporaries to registers, let alone eliminate MOVE instructions. In these cases, the values in the registers may need to spill to memory (requiring a change in the program so that we write them to the stack whenever we change them and read them from the stack whenever we need them).
  - How does spilling help, given that you need registers for writing and reading? It turns out that we can make the lifetime of a temporary much smaller.
  - Consider a program in which \( t_{10} \) is assigned a value in the first statement and that value is not used again until the hundredth statement. \( t_{10} \) is live for all 100 statements.
  - However, if we write it to the stack immediately after assigning it, and read it from the stack immediately before using it, then it won’t be alive in between (since the read overwrites the register). This frees it to be used for other purposes in the mean time.
A Basic Algorithm

- We will use an *interference graph* to do register allocation. This is a graph in which there is one node for each temporary and an edge between two nodes if the corresponding temporaries are live at the same time.
- The algorithm for allocating registers seems relatively simple

```plaintext
compute g, an interference graph for the program
while g cannot be colored with k colors
    remove a node from g, spilling it to memory
    rewrite the program and recompute g
end while
 color g with k colors
replace each color by a register
end allocateRegisters
```

- Note that this requires three key subroutines:
  - Determining whether a graph is k-colorable
  - Finding a k-coloring for a graph
  - Picking a node to remove
- We’ll discuss each in turn.

**Detour: NP-completeness**

- It turns out that k-coloring is an NP-complete problem
  - I’m not as sure about k-colorability. For example, we know that every planar graph is four-colorable.
  - It is likely that many of you do not know what NP-complete means, so we’ll look at it for a little while.
- What is an NP complete problem? Many computer scientists read that as “very difficult” or “exponential time”
  - However, NP complete has a more formal definition.
  - I’ll give an informal, but more precise definition
- A problem is in NP if it is possible to determine whether a solution to the problem is correct in polynomial time (big-O of some polynomial).
- One solution to any problem in NP is “repeatedly guess a solution and test whether it is correct”
  - This gives the N in the NP. A problem is in NP if one can nondeterministically (N) guess and verify an answer in polynomial (P) time.
  - Note that if we the solution must be deterministic, the running time must incorporate the number of guesses, which may be exponential.
- Of course, some problems that can be verified in polynomial time also have better solutions than “guess and test”.
  - For example, you can verify that a list is sorted in polynomial time, but we typically find better ways to sort than “try all permutations until you find a sorted one”.
- There are some problems for which no better algorithm has been found.
  - In many of these, there are exponentially many potential solutions to verify (e.g., all
permutations).

- Some of these problems have the characteristic that if we found a polynomial-time solution to one problem, then we can find a polynomial time solution to any problem in NP. How? By converting the other problem to the one that we know how to solve, and then converting the solution back again.
- Take CS301 and CS341 for more information.

Optimistic Graph Coloring

- Since graph coloring is NP-complete, we instead look to an approximate algorithm that often colors well, but sometimes fails to color optimally.
  ○ In particular, this algorithm may fail to K-color some graphs that are K-colorable.
- The steps are relatively simple.
  ○ **Build** the interference graph.
  ○ **Simplify** the graph using a simple heuristic: whenever a node has less than K neighbors, drop it from the graph (since we know we can color it using some color). We put it on a stack so that we remember to color it later.
  ○ If necessary, **Spill** some node with degree of K or greater. In fact, we do not yet spill it to memory. Rather, we remove it from the graph and will spill it in a later step. After spilling, we again try to simplify.
  ○ Once the graph is empty, we **Select** colors to assign to the nodes by popping them off the stack and assigning colors as we go.
- If during selection, we find that a spilled node cannot be colored (sometimes they can, and sometimes they can’t) we note that it really does have to be spilled. We give it no color and continue.
- If some temporaries really had to be spilled, we need to rewrite our program to do that spilling and then start again.

An Example

Here’s a graph (a list of edges) to try the algorithm on. We’ll try to four-color it. Note that this is a modified version of a graph that I took from Appel’s book (a book we no longer use for reasons that the first set of CS362 students will be happy to tell you).

\[(A,B), (A,F), (A,G), (B,C), (B,D), (B,K), (B,M), (C,M), (D,J), (D,K), (D,M), (E,F), (E,J), (E,M), (F,J), (F,M), (G,H), (G,I), (G,J), (G,K), (H,I), (H,J), (J,K), (J,L), (L,M)\]

(I haven’t given the underlying program. I don’t think we’ll need to rewrite the program, but it may depend on choices we make.)

Coalescing

- Note that the algorithm doesn’t have any emphasis towards helping eliminate moves.
  ○ It can eliminate moves if the two temporaries in a move are assigned the same color, but no particular effort is made to guarantee this.
- Hence, we’d like to modify the algorithm so that it *coalesces* the two parts of a MOVE when it is “safe” to do so.
○ That is, we join two nodes into one, and note that the two temporaries will be assigned the same register (or memory location).
○ When is it safe to do so?
○ Abstractly, when doing so doesn’t affect the K-colorability of the graph.
○ But this is difficult to determine, so we use heuristics.
● The Briggs heuristic: Coalesce a and b if no neighbor of the combined node will have k or more edges.
○ This does seem safe, doesn’t it? After coalescing, we can still remove all neighbors, so we haven’t changed the colorability of the graph.
● The George heuristic: Coalesce a and b if, for all neighbors n of a, either n already interferes with b or n has degree less than k.
○ Hmm ... analysis of this is harder. We need to consider the graphs that result after removing insignificant degree neighbors.

The Algorithm

function allocReg(program p, regcount k)
  build the interference graph for p
  build the list of move-pairs
  create an empty stack
  while p is nonempty
    repeat
      simplify:
      remove nodes of arity < k not in move pairs,
      pushing them on the stack
      coalesce:
      join nodes in any move pairs if safe to do so
      if (we neither simplified nor coalesced)
        freeze:
        pick some move-pair node with arity < k, and
        remove all move-pairs with that node
        until no changes can be made
      if there are nodes of arity >= k
        spill:
        remove some node of arity >= k,
        pushing it on the stack
      end if
    end while
    select:
    while there are nodes on the stack
      if some color can be assigned to the top node on
      the stack then assign that color
      otherwise, note that the top node must be spilled
    recover:
    if some node was marked as spilled during the select phase,
    then update the program to do the spilling and call allocReg again.
  end allocReg

● The algorithm, as given, is not quite right. For example, sometimes we have additional restrictions on when registers can be used that need to be reflected in the graph.