CS316: Compilers Lab

Programming Assignment 7: Optimization - register allocation Due: 17/4/2022

1 Introduction

This step asks you to perform register allocation. Up until now, you have been using a version of Tiny that allows you to use 200 registers. Now, you will use a version of Tiny that uses 4 registers (r0, r1, r2, r3). Register allocation is the process by which variables (global variables, local variables, and temporaries) are assigned to registers to allow values that you need to reside in registers as much as possible, rather than having to read and write them from the stack. This can be challenging when you have a limited number of registers, as in that case, some times you do not have enough registers to hold all the values you care about. Week 10 slides provide more details about how register allocation works.

2 Liveness

The first step in performing register allocation is performing a liveness analysis. We are asking you to perform liveness analysis across an entire function at once. See Week 11 (Slide 34 onwards) for more details.

2.1 Control flow graphs

The first step in computing liveness is to build a control flow graph for each function in your program. To represent your control flow graph, each IR Node should know its successors (IR instructions that could possibly execute immediately after it) and predecessors (IR instructions that could possible execute immediately before it). Conditional jumps have two successors: the explicit target of the jump, and the implicit (fall-through) target of the jump. Unconditional jumps only have one successor. Function calls should be treated as straight-line IR nodes (i.e., they are not treated as branches; their successor is the instruction immediately after the call). Return nodes do not have any successors.

2.2 Computing Liveness

For each IR node in a function, you should define two sets: GEN and KILL. GEN represents all the temporaries and variables that are used in an instruction, and KILL represents all the temporaries and variables that are defined in an instruction. For most instructions, this should be pretty straightforward. A few tricky cases:

- PUSH instructions use the variable/temporary being pushed
- POP instructions define the variable/temporary being popped
- WRITE instructions use their variables.
- READ instructions define their variables.
- CALL instructions require special care. Because we do not analyze liveness across functions, we must make conservative assumptions about what happens function calls. In particular, we GEN any variables that may be used, and KILL any variables that must be used. The GEN set for any CALL instruction therefore contains all global variables, while the KILL set is empty.

Once you know the GEN and KILL sets for each IR node, you can compute liveness. To do this, define IN (live-in) and OUT (live-out) sets for each IR Node. Initialize the OUT sets for RETURN IR nodes to all global variables (because global variables may be used after the function returns), and initialize all other sets to empty. Then compute the live-in and live-out sets for each IR node as follows:

- The set of variables that are live out of a node is the union of all the variables that are live in to the node's successors.

- The set of variables that are live in to a node is the set of variables that are live out for the node, minus any variables that are killed by the node, plus any variables that are gen-ed by the node.

Note that in these definitions are recursive: the live-out set of a node is defined in terms of the live-in sets of its successors, which are in turn defined in terms of the live-in sets of their successors, and so on. If there is a loop in the code, then the definition seems circular.

The trick to computing liveness is to compute a fixpoint: assignments to each of the live-in and live-out sets so that if you try to compute any node's live-in or live-out set again, you'll get the same result you already have. To do this, we will use a worklist algorithm:

1. Put all of the IR nodes on the worklist
2. Pull an IR node off the worklist, and compute its live-out and live-in sets according to the definitions above.
3. If the live-in set of the node gets updated by the previous step, put all of the node's predecessors on the worklist (because they may need to update their live-out sets).
4. Repeat steps 2 and 3 until the worklist is empty.

(Note: you can write a slower version of this code that ignores identifying nodes' predecessors. This still works. However, we suggest you to put all the IR nodes on the worklist and process all of them. If *any* live-in or live-out set has changed, put all the IR nodes on the work list and repeat the process.)

3 Register Allocation Algorithm

Use the bottom-up register allocation algorithm or the graph-coloring method discussed in class. For each statement, you must ensure that the source operands are in registers, and that there is a register for the destination operand. Use the liveness information you computed (i.e., the live-out set for the instruction) to determine when it is safe to free registers, and when a dirty register needs to be stored back to memory (only when the variable in the register is live).

Bottom-up register allocation works at the basic-block level: any register allocation decisions you make apply for the current basic block only. This means that when you get to the end of a basic block, you must reset your register allocation. Any register that (a) holds local/global variables and (b) is dirty should be written back to the stack/global variable.

Note also that because a CALL instruction jumps into another method, any global variables that are in registers when the CALL is performed should be freed immediately prior to the CALL instruction, ensuring that the correct value for the global is in memory. This is different from saving the registers on the stack prior to a function call. The latter is done so that the caller method doesn't get its registers overwritten; the values of the registers are stored where only the caller can see them. The former is done so that the callee method sees the right values for global variables; the values need to be stored back to globals so that everyone can see them, and freed from the registers so that the caller will reload them after the callee returns.

Testing your Tiny code You can test your Tiny code by using, `tiny4regs.C`, *a version of the simulator that limits you to 4 registers.* `tiny4regs.C` is provided to you along with the starter files. There are no test cases provided along with the starter files for this assignment. You must test your compiler with all the test cases of PA4, PA5, and PA6. In addition, we will test with some hidden test cases.

4 What you need to do

Perform the liveness analysis and register allocation steps as described above, so that your compiler generates code that only uses 4 registers.

Handling errors All the inputs we will give you in this step will be valid programs. We will also ensure that all expressions are type safe: a given expression will operate on either INTs or FLOATs, but not a mix, and all assignment statements will assign INT results to variables that are declared as INTs (and respectively for FLOATs).

Grading In this step, we will only grade your compiler on the correctness of the generated code. We will run your generated code through the Tiny simulator and check to make sure that you produce the same result as our code. When we say result, we mean the outputs of any WRITE statements in the program (not details such as how many cycles the code uses, how many registers, etc.)

We will not check to see if you generate exactly the same code that we do – no need to diff anything. We only care if your generated code *works correctly*. You may generate slightly different code than we did.

5 What you need to submit

- Place all the necessary code for your compiler that you wrote yourself. You do not need to include the ANTLR jar files if you are using ANTLR.
- A Makefile with the following targets:
  1. `compiler`: this target will build your compiler. If you are using ANTLR, this should create a `.jar` file. (*-1 for warnings*)
  2. `clean`: this target will remove any intermediate files that were created to build the compiler. (*-1 for not doing the clean properly*)
  3. `dev`: this target will print the same information that you printed in previous PA.
- A shell script (this must be written in bash) called `runme` that runs your compiler. This script should take in two arguments: first, the input program file to be compiled and second, the filename where you want to put the compiler. You can assume that we will have run `make compiler` before running this script.
- You should tag your programming assignment submission as cs316pa7submission

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation. If you have a folder named `test` in your repo, it will be deleted as part of running our test script (though the deletion won't get pushed) – make sure no code necessary for building/running your compiler is in such a directory.

Hints for Project Step 7

Here are a couple of ideas to help you get started with project step 7. The description of the step on the project page tells you to do liveness on a per-basic-block basis. Here, I will describe an approach that calculates liveness on a per-statement basis. You can choose either approach.

1. **Control flow graph.** In class, we discussed how to build a control flow graph for any piece of code, including code that uses arbitrary jumps and branches. Building a CFG for Micro code is easier, as the only control flow is `if` statements and `for` statements. Your current IR probably represents the three address code as a linked list of IR instructions. You can use this linked list as a basis for a CFG—each statement represents one node in the CFG. The only modifications you need to make to this are to add additional predecessor and successor information to nodes that need it.

    - **Unconditional jumps** (like at the end of for loops) have only one successor: the target of the jump statement. When you see an unconditional jump, add the target of the jump statement as a successor of the jump, and the jump statement as a *predecessor* of the target.
    - **Conditional jumps** have two successors: the "fall-through" target, which is already a successor in the original linked list, and the "taken" target. Add the branch as a predecessor of the taken target, and the taken target as an additional successor of the branch.

2. **Liveness analysis.** Once you've built the CFG, liveness analysis is easy. Compute the GEN and KILL sets for each statement. Create IN and OUT sets for each CFG node (program statement), and initialize them to empty. Then iterate over every node, updating IN and OUT according to the dataflow equations we discussed in class. You need to iterate until convergence (there are many ways to do this, but in general, when a statement's liveness information changes, you need to update its predecessors).

3. **Constructing basic blocks.** A basic block is a set of statements that starts with a "leader" statement and continues until the next leader. In this case, leader statements are any statements that a) have predecessors other than the immediately preceding instruction or b) have successors other than the immediately following instruction.

4. **Register allocation.** You may use either bottom-up register allocation, as discussed in class, or graph coloring-based register allocation.