Thomas Simko
CSC 431

Compiler Writeup

Parsing:

My compiler uses the given parser for handling input files. This given parser creates a token stream using a lexer, parses that token stream, and creates an Abstract Syntax Tree. My compiler then uses the Abstract Syntax Tree to perform the compilation.

Static Semantics:

In order to type check the program, the compiler traverses the Abstract Syntax Tree (AST). Upon reaching an expression, the expression is evaluated for what type it is, such as a struct or integer. If the expression has a subexpression, like a binary expression, those subexpressions are also evaluated for their types. To ensure that they types match, there are assertions in the compiler that will throw an exception if any of the assertions fail.

While expressions return their type, statements return a boolean on whether or not they return among all paths. This is helpful for preventing functions that do not return along all paths. This means that if a conditional statement were to return true, it would have to return along both the if and the else branches. If it did in one but not the other, then it would be false. Block statements are special because they only need one statement in them to return for them to return among all paths; however, instructions after that return will not be executed. I made the design decision to not allow this in my compiler since there is no reason for those instructions to exist if they are never executed. Finally, while statements were the other major area to debate. Returning from inside of a while statement would lead one to believe that it returns among all paths; however, we cannot ensure that the while statement will be run, so it defaults to false. This does however come into question if there is a while statement where the condition in always true; but, I opted not to check for that specific condition.

Passed into each of these check type expressions and statements is a function table, struct table and the current function's name. This allows the expressions to perform lookups to gather the return types of these items. The function table uses the function's name as its key and its return type as its value. This allows us to ensure that any function calls return the same value that the variable is expecting. This is also used to match the return type with the actual return statement to ensure those types match as well. Moving on to the struct table, this dictionary uses the name of the struct as the key and another map of the variables and their types as the value. This allows us to store the fields contained in the struct, so that references into the struct can be validated.

Intermediate Representation:

After the program is type checked, it is transformed into a control flow graph (CFG) with LLVM instructions. By building the code into this representation, it allowed us to visualize all of the possible routes of our program and allowed us to insert labels in our LLVM code. The program can have at least one CFG, one for each function. Each of these CFG's begin with a start block which has all of the setup code for each function. Each start block has one successor, a basic block. This is the most generic block because it does nothing special. Basic blocks however can have more than one successor if certain instructions are used like a conditional statement. With regard to conditional and while statements, one or more new blocks are created with the current block being their predecessor. The condition of these statements is place within the current block before moving on to the body of the statement. For a while block, the condition is also added to the very last block of the loop so that the condition is evaluated again. Finally, at the end of each program is an end block.

Using LLVM as an intermediate allowed us to extract some of the details of arm and create an easier translation. This step also allowed us to test our code early to ensure that everything was working before moving on to some of the more difficult operations. With LLVM, there is no real need to worry about the registers that the program is using. And, for our first iteration we utilized the stack to store all of our variables. By placing variables that might change on the stack, we did not have to concern ourselves with using the changed values later in the program. Further into the quarter; however, we returned to change our LLVM into SSA form. By placing our values into SSA, we could keep track of all of a register's uses and definitions which allows us to perform many optimizations on the code. Also, it allowed us to utilize registers to their fullest potential to enhance the speed of our program. Now instead of needed to load a variable every time from the stack, many of them were just stored in registers that we could then simply reference.

Optimizations:

1. Function Inlining

The first optimization that is performed on the code is function inlining. By inlining a function we can take away all of the stack based operations with setting up and tearing down a function. My compiler requires a few conditions to be met before attempting to inline the function. First, it must only have one call to it. This is to reduce the amount of duplicated code within the program. And secondly, it must be a simple function with only one basic block. This means that the function has to be relatively simple and cannot have any if or while statements in it. If it were to have any of these statements, I would have to break up the CFG which might eliminate some of the optimizations that come from instruction ordering. If these two conditions are met,

however, the function will be inlined. The compiler first pulls all of the code from the basic block and replace the function call instruction. Then it replaces all uses of the returned value with the register that is being returned from the function. Finally, it replaces all of the parameter uses with their respective argument values.

2. Simple Static Constant Propagation (SSCP)

By implementing Simple Static Constant Propagation, I was able to replace references to registers with constant values. To accomplish this, I used a lattice. This lattice has three different levels: top, integer, and bottom. Initially all of the registers are set to top and the parameters are set to bottom. The parameters are then added to a working list that iterates until it is empty. When an item is selected in the working list, it is first removed from the list. Then all of the registers that rely on that item for their definition are evaluated via the lattice. For example, if an addition instruction was evaluated and both of the registers were integers, then the save register would be marked also as an integer and the compiler would add the two values. However, if the values were top and an integer then they would move down the lattice to a common value. This removes the reliance of other instructions on that particular instruction. The save register for that instruction is added to the list if it was not already determined to be bottom. After the working set is empty, then the compiler replaces all of the values with their determined lattice values.

3. Useless Code Removal Using Critical Instructions

After performing constant propagation, there may be instructions that are no longer necessary. These instructions would simply take up CPU cycles and would not effect the output of the program. To deal with this, I implemented useless code removal using critical instructions. This form of code removal is based on the idea that there are certain critical instructions that must stay for the program to work properly, such as function calls and returns. Therefore, these instructions must remain and so must all of their sources, their sources' sources, and so forth. At the end, if any instruction is not marked as critical, then it is removed. This allows us to greatly diminish the program length and decrease execution time.

Code Generation and Register Allocation:

After the LLVM instructions have been created and optimized, the code is then transformed into assembly. LLVM has a fairly straightforward translation to ARM; however, some instructions need to be changed. For instance, a function call in arm can only accept up to four parameters in registers; the rest must be added to the stack.

In order to allocate registers for ARM, I had to perform graph coloring. To do this, the compiler generates an interference graph based on the live out sets of a particular block. By creating this graph, we can then identify which values have to be

in different registers. The nodes were then removed from the graph, unconstrained first, then constrained, and finally forced registers (i.e., r0), and added to a stack. Removing the nodes one-by-one from the stack, each is assigned a register value that doesn't conflict with any of it's interfering nodes. If there are no more available registers for the value to go into, the program spills. This value then have to be saved onto the stack and loaded and stored upon each use.

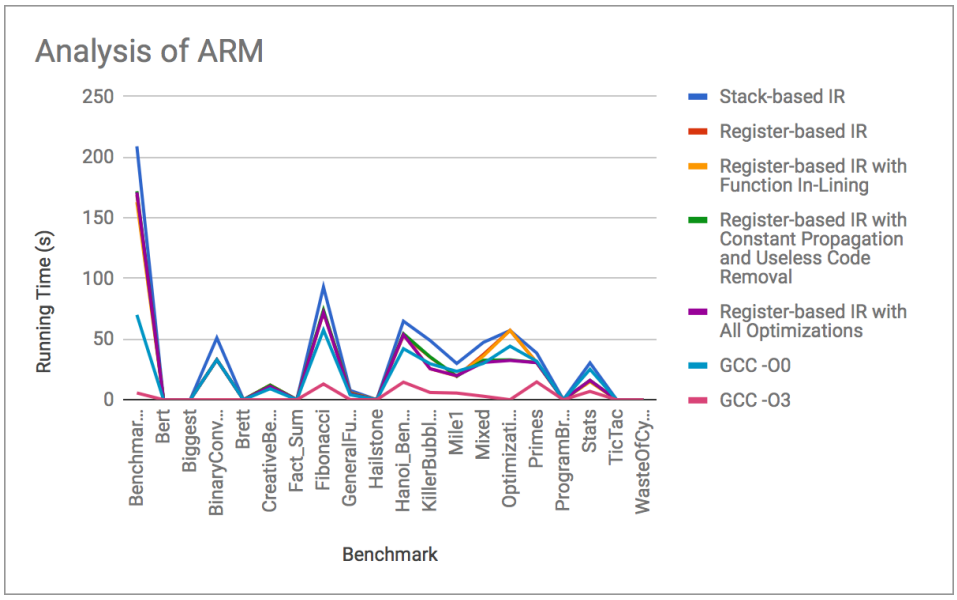
If we are coming from a SSA-based LLVM, then the phi instructions have to be dealt with. This can be done using moves because ARM does not need to conform to SSA. Therefore, for each operand in the phi, a move is placed at the end of the block where that operand is defined. This way we can update the value in the phi appropriately based on which path was taken.

Other:

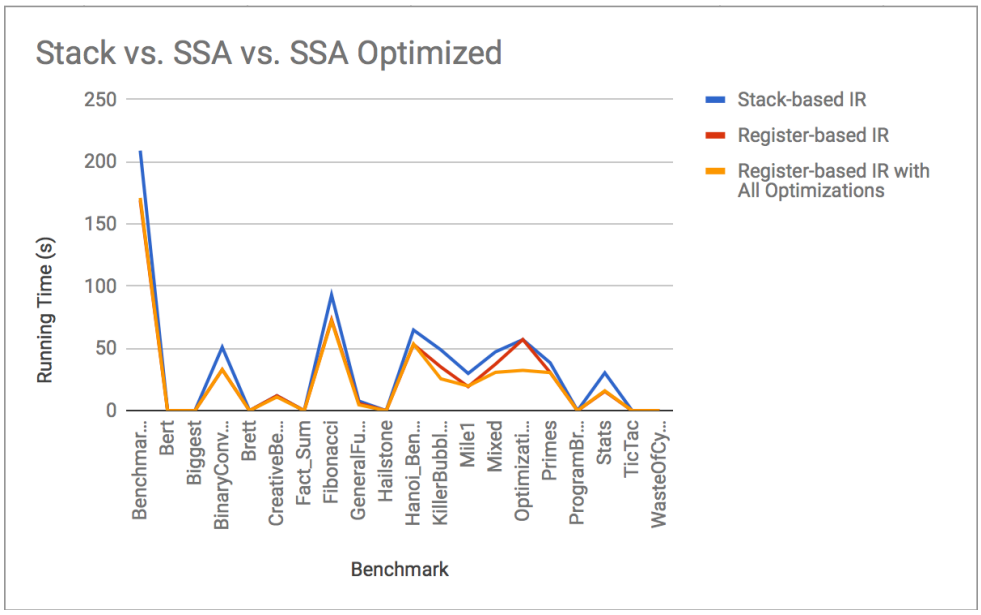
One aspect that I am particularly proud of is coming from SSA LLVM to ARM. Using SSA was very interesting and helpful in storing the details about a register's uses and definition. But transforming that into assembly was challenging due to the phi instructions. Once I was able to grasp my head around how to circumvent the issue, it was fairly simple.

In addition to this, the graph coloring is also something that I found very interesting. I had never implemented any sort of graph coloring before and it was very intriguing to perform. I particularly liked the ordering of how we pulled the nodes out of the graph in order to minimize the chances of spills.

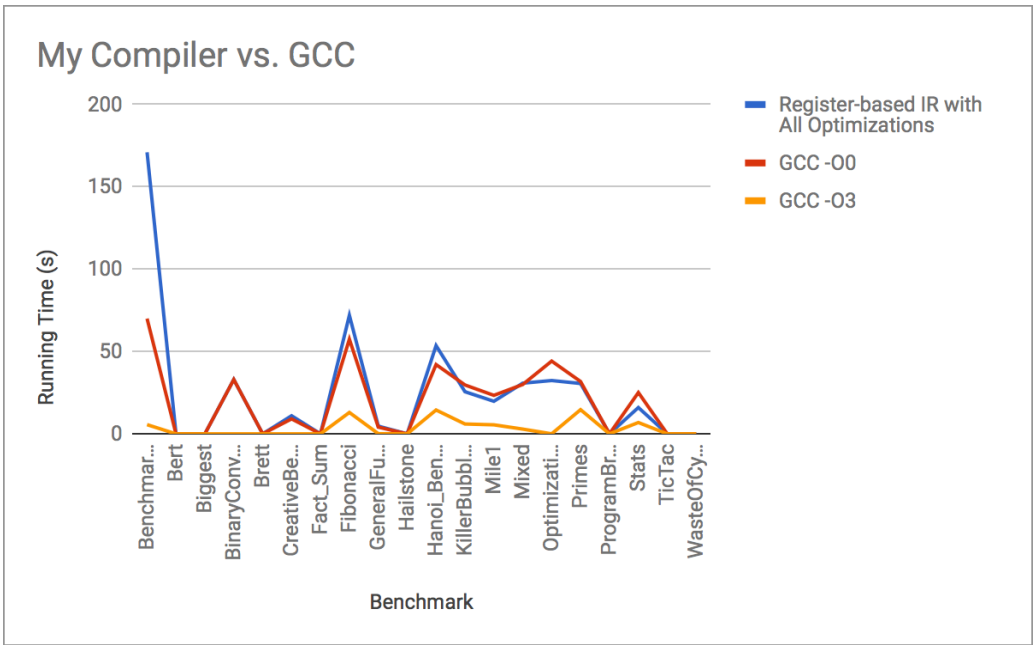
Analysis:



This chart shows how my compiler did across common variations of it. Overall, we can see that stack-based IR is the worst and optimized GCC is the best.



From this line graph, we can see how switching to a register-based IR provided a large benefit over the stack-based IR. Also, we can see that the optimizations did not do much on many of the benchmarks, but on some it provided a serious reduction in running time.



From this graph, we can see how my compiler fares to GCC unoptimized and optimized. Overall, my compiler traded which one was better with GCC unoptimized. But, the years of research on GCC's optimizations showed their value because it was much better than the others.