Caroline Cullen
December 22, 2018
CSC 431

Mini Compiler Reflection

Architecture

- **Parsing**

  My mini compiler utilizes the given parser. The compiler takes the given file and begins by tokenizing the mini input file and then feeding the tokenized mini code to the parser, which then takes the tokenized input and creates the abstract syntax tree, which is the first initial representation of the program within the compiler as a data structure. The structure of the ast is that each token is essentially represented as a class within the compiler and depending on the token and where it is, there are other token classes within the origin class. For example, the whole program resides within the program class and within the program class are type declaration, declarations and functions which were all originally represented as tokens. Within each of these classes are other class, such as the block statement class within the function class. You can see the tree structure that is formed.

- **Static Semantics**

  For static type checking the strategy I employed was to keep the original structure in place and work with and around the parser. The type checking is done in a separate pass from the actual parsing, this is also the pass where the global, local, and struct symbol tables are created and used throughout the program. The symbol table can be printed by compiling with flag -S. I then created a method called TypeCheck in the statement and expression interfaces, which take, as a parameter, the local symbol table. This method evaluates all expressions and statements within the ast by working its way down to the leaves of the tree, getting their types, and using this information to work back up the tree while checking for the expected types.

  For return checking I employed the same technique as static type checking. Return checking is also done in a separate pass and the statement interface has a method called returnCheck, which takes the return from any statement that can return and checks it for what is expected in the same manner as type checking.

- **Intermediate Representation**

Caroline Cullen
December 22, 2018
CSC 431

To begin creating the LLVM intermediate representation, I had to create a control flow graph for every function contained within the mini program. Every CFG has an entry block from which all of the other blocks in the CFG stem from and an exit block where the return instructions of the function are placed. From the entry block there will always be another block, from here we begin adding LLVM instructions. If a loop is encountered, then a new block is created where the guard condition instructions are placed and then from this block two more blocks are added to the graph, one for if the guard is true and one for if the guard is false. At the end of the true block is a branch back to the guard block; in the false block is the rest of the code for the function. If a conditional statement is encountered the guard is added to the current block and then three blocks are added, one block for if the guard of the conditional is true, one block for if its false, and finally a join node where the two “else” and “then” blocks are joined back together and the rest of the LLVM code is placed. The exception to both of the above CFG procedures is if there is a return statement contained in any of the above block, in which case the exit block is returned from the current block. This is where the return instructions are then placed. The purpose of this whole structure is to guarantee one entry and exit point from a block which are guaranteed not to have any jumps in or out of the blocks; this is the structure that LLVM and ARM both require. The structure further guarantees that if the first instruction in a block executes then the rest will follow.

After creating the CFG it is then necessary to convert the ast and its instructions into the LLVM intermediate representation while within the confines of our CFG. Changing the code from mini language to LLVM IR is helpful in making optimizations and analysis on this code much easier than if we were to do a direct conversion to ARM instructions. Retargeting the compiler can also be done in a more straightforward manner. In order to change the code to LLVM I went through all the statements while creating the CFG and placed those instructions within the graph to do a direct conversion. When encountering an expression during the analysis of a statement, I analyzed the expressions while also analyzing the statements; this allowed me to place the LLVM instructions in the correct basic block. The expression interface contains a method called getLLVM that is called whenever an expression is encountered. The method receives the current working block and the LLVM instructions (retrieved from the analysis within the expression class) are placed directly into that block that was passed into the getLLVM function. For SSA, my approach was to implement the algorithm based on the pseudocode that was given. After the control flow graph is fully implemented the blocks get sealed at the very end. Throughout the SSA analysis I keep a running total of all of the phi instructions in each block and print them out first when printing all of the

instructions to the ".ll" file. The benefit of implementing this algorithm is that it simplifies your code, implementation of certain optimizations, and improves compiler speed.

- **Optimizations**

  The optimization that I implemented in my compiler is function in-lining. During the construction of the CFG, every function is evaluated based on their ability to be in-lined and is marked as such. If the function is then called later and was marked as able to be in-lined then I in-line the function. This heuristic is based off of research I conducted by Googling this issue. There are many differing opinions on this subject and there seemed no "correct" way to do it; the implementation depended on the context. I tried to "average" all of the ideas that I came across. It seemed that for the most part not only do you want to inline only simplistic CFGs but also, you want an extremely short instruction count for this optimization to be worth it. A small instruction count is important because the more instructions, the more potential register pressure, which can cause spills to the stack. This result can actually lose all of the benefit from in-lining the function in the first-place, due to the amount of time it takes to store and retrieve values from the stack. Considering everything previously state, I in-lined functions that were less than 5 instructions long and only had one block, excluding the entry and exit blocks.

- **Code Generation and Register Allocation**

  For my translation to ARM I went through my CFG and did a one to one mapping of an LLVM instruction to an ARM instruction. For every LLVM instruction an ARM instruction class is created and return as a string into the node of the CFG that it originated from. The beginnings of my register allocation (get set, kill set, and live out set calculations), are done as a separate pass over the CFG before the ARM codegen phase. Register allocation is done on the LLVM instructions because the ARM instructions were directly translated into a string. The phi instructions must be eliminated because of the fact that arm does not have phi instructions. The way that I implemented the translation of the phi instructions from LLVM to ARM is by first numbering all of my phi instructions. Then whenever a phi instruction gets a value from another basic block, we will call this %v0, I load %v0 into a register titled phi plus the phi number from the first step, in this case %phi0. Then, in the phi block I load %phi0 from before into the result register of the original phi. Whatever is loaded from %phi0 will be different depending on what path the code takes to load the value into %phi0.

Caroline Cullen
December 22, 2018
CSC 431

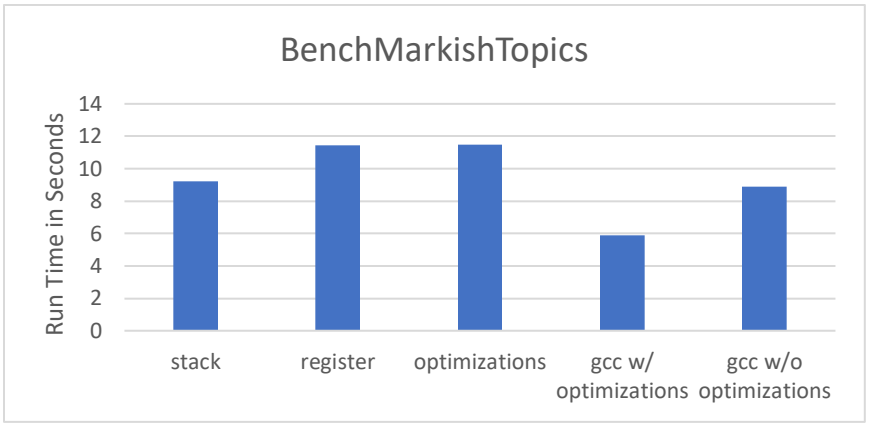
- **Other**

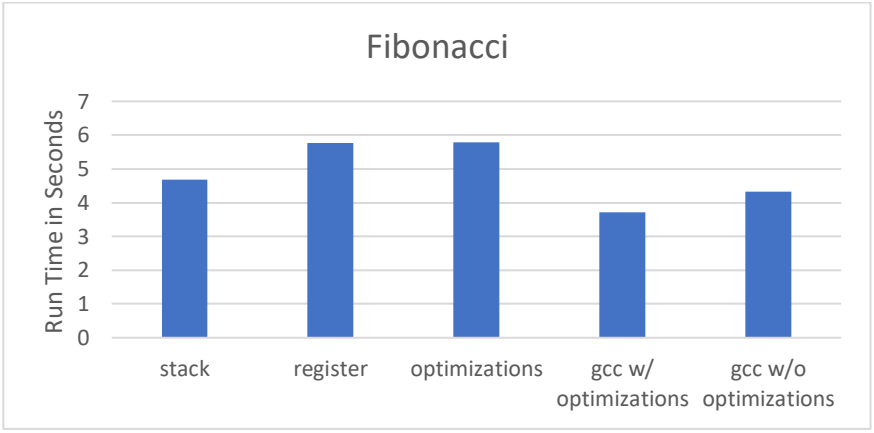
  I am proud that I wrote a compiler by myself in 10 weeks.

Analysis

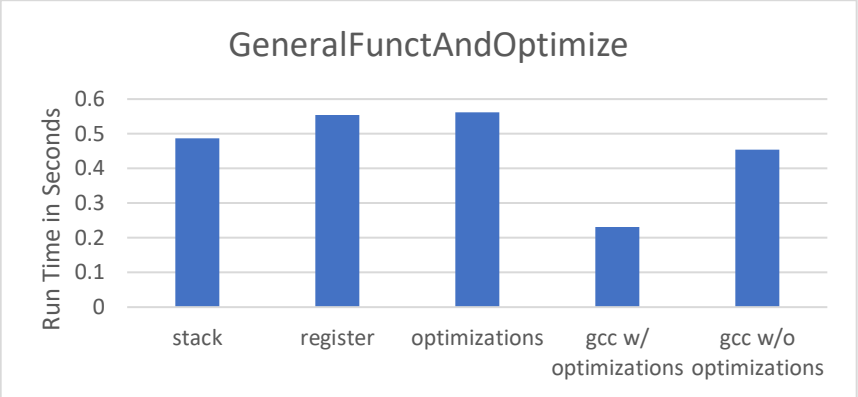
Since register allocation was not fully implemented in my compiler, the configurations I used were gcc's generated llvm (using the command gcc -w -S -O0 -emit-llvm "file".c) and clanging that llvm that gcc generated. I compared the run time of this to my clanged llvm output with various configurations.

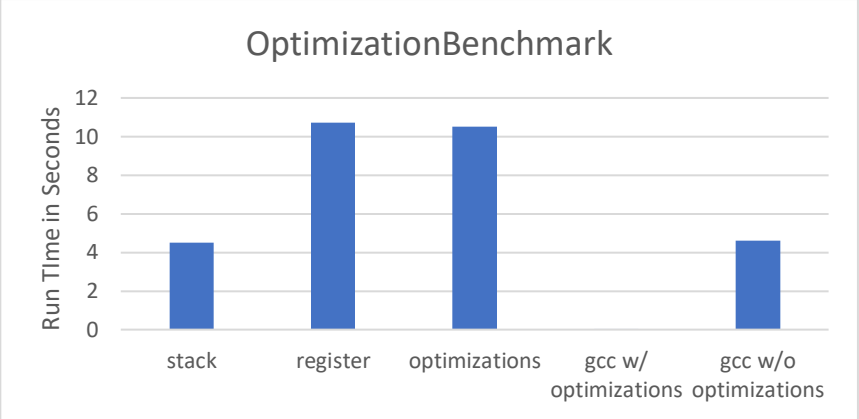
Below are bar graphs for every benchmark, they include the run times for register based LLVM, stack based LLVM, optimizations, gcc with optimizations, and gcc without optimizations with input.longer:

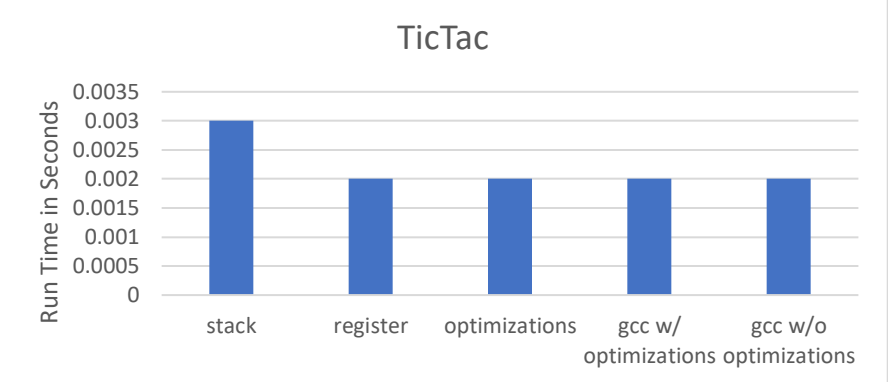


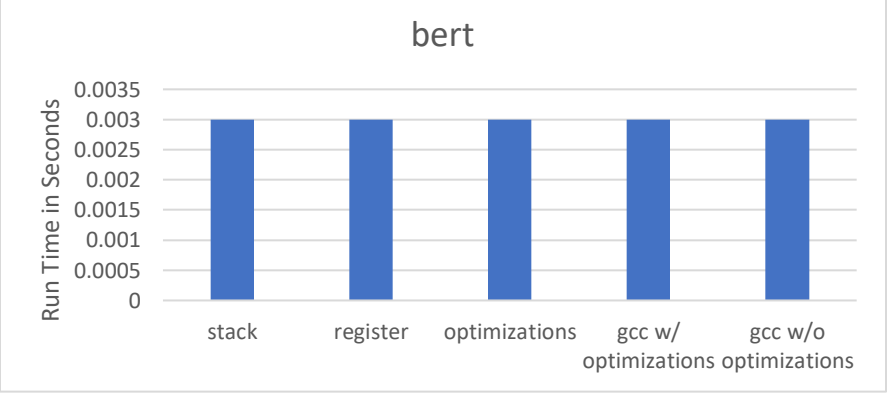


Caroline Cullen
December 22, 2018
CSC 431

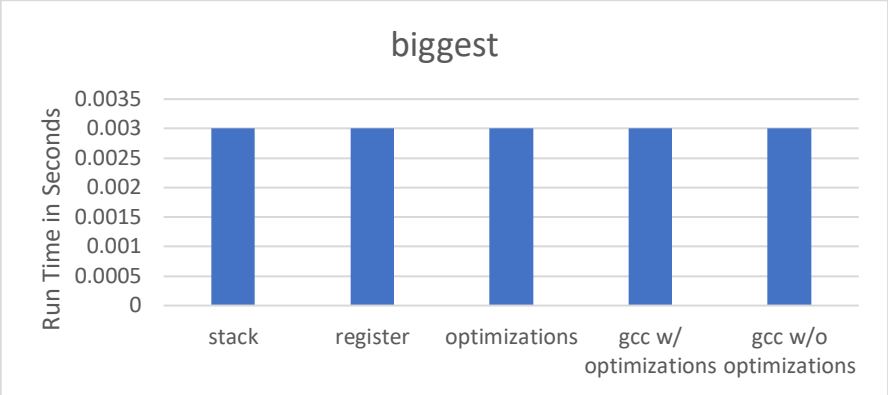


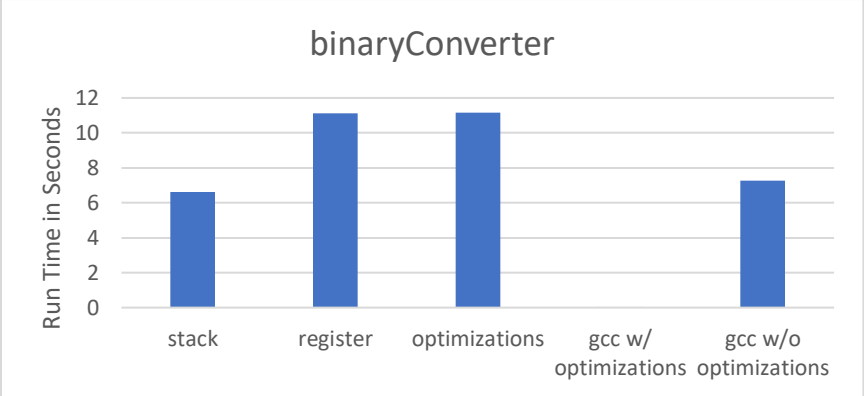


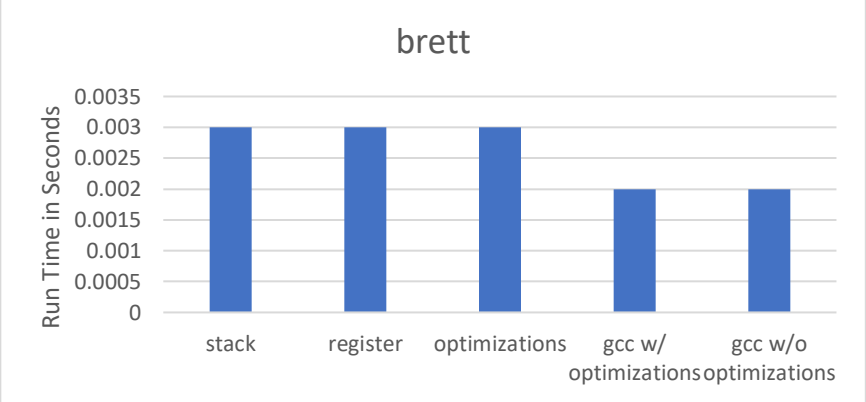


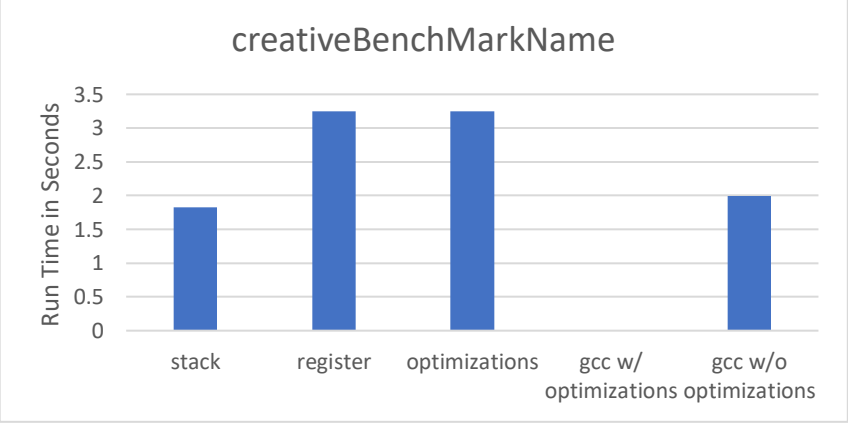


Caroline Cullen
December 22, 2018
CSC 431

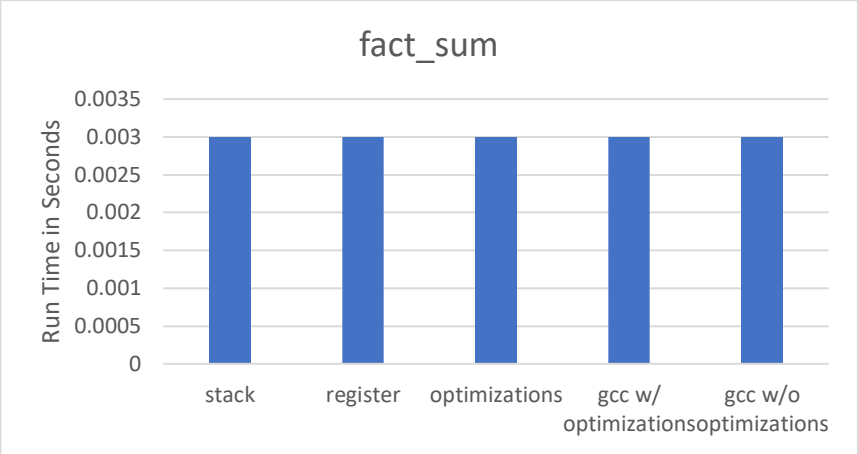


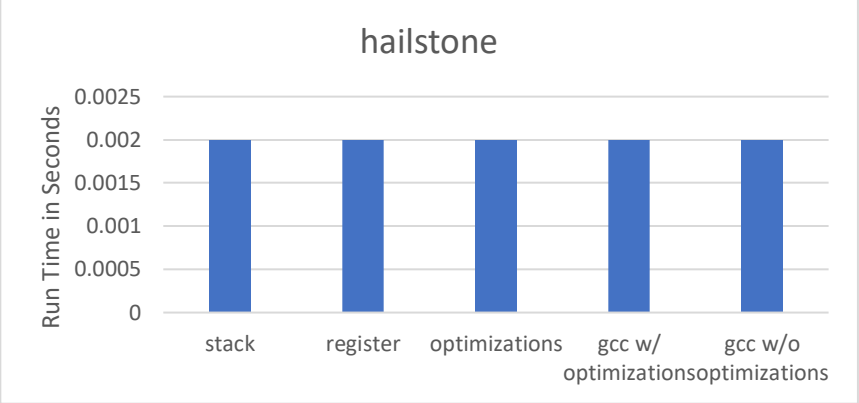


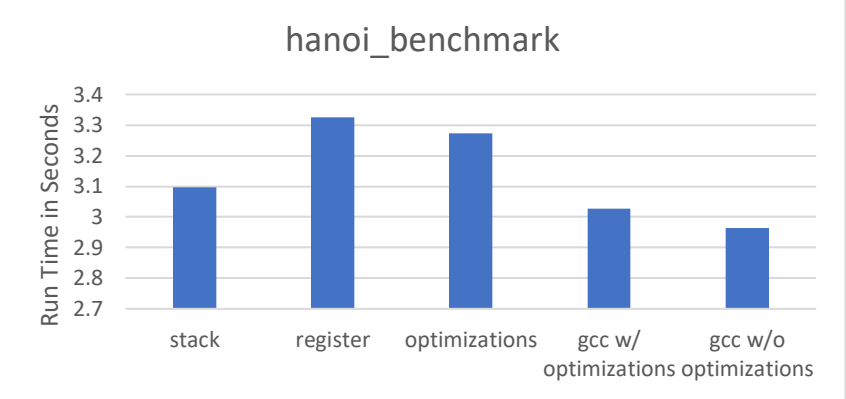


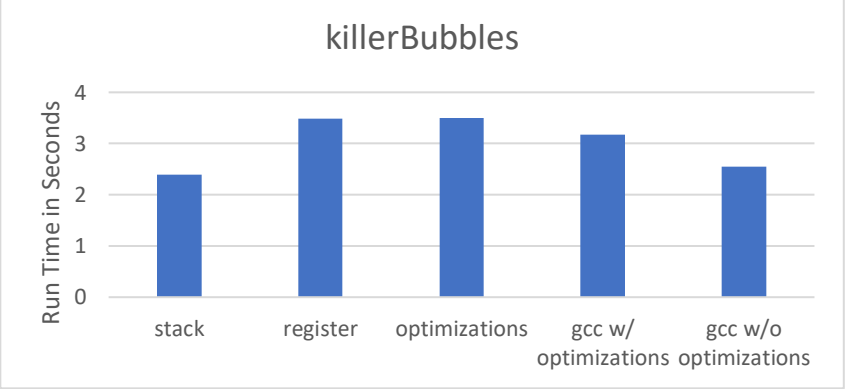


Caroline Cullen
December 22, 2018
CSC 431

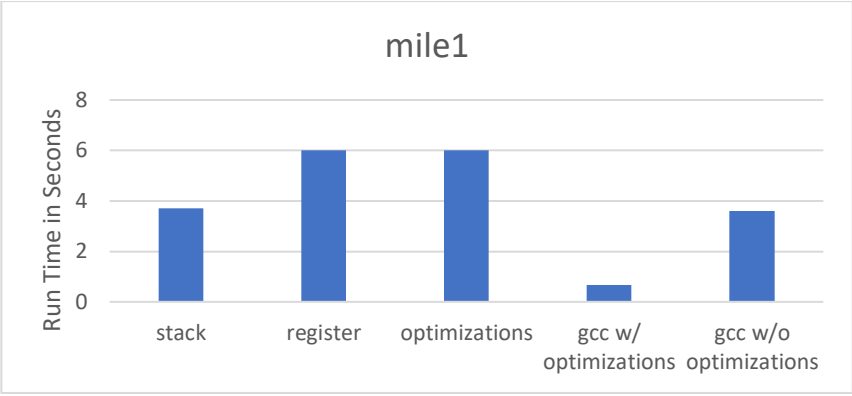


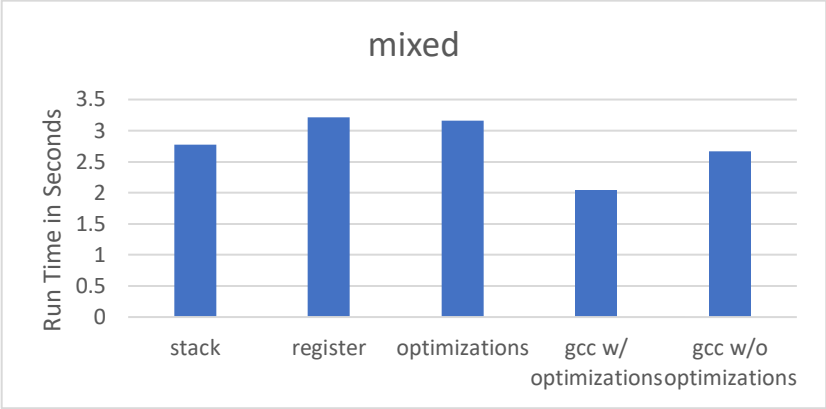


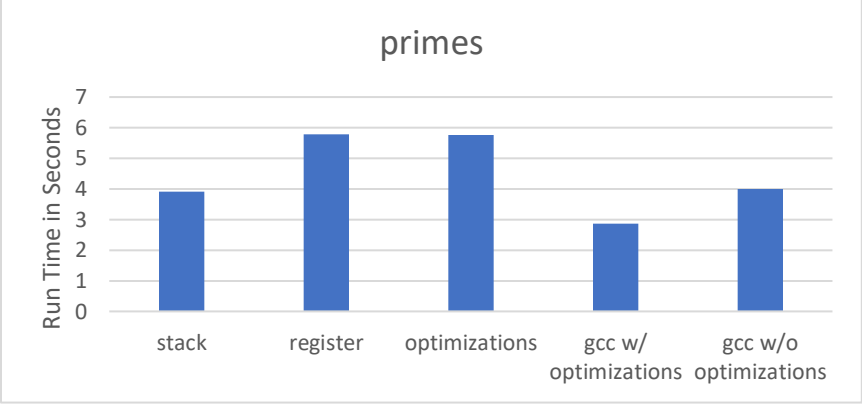




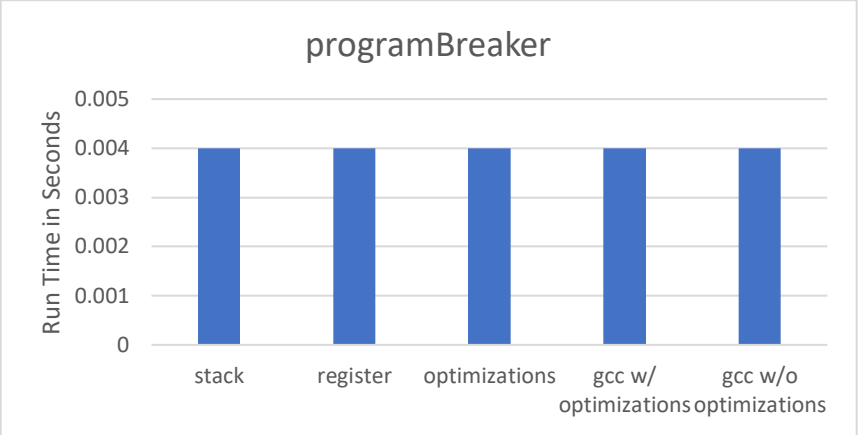
Caroline Cullen
December 22, 2018
CSC 431

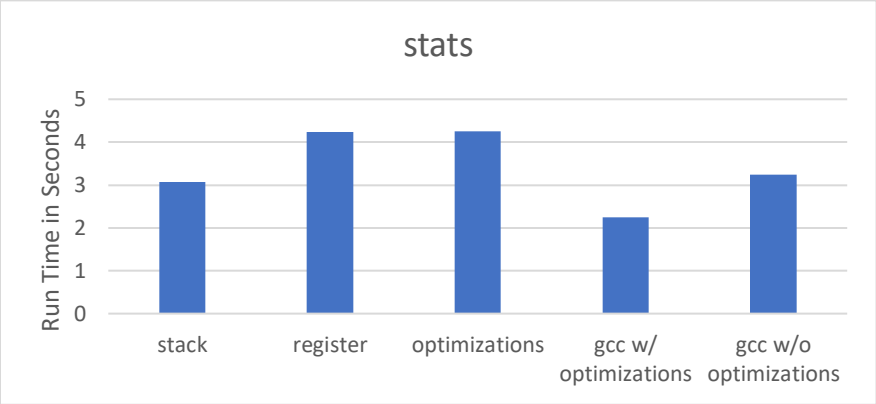


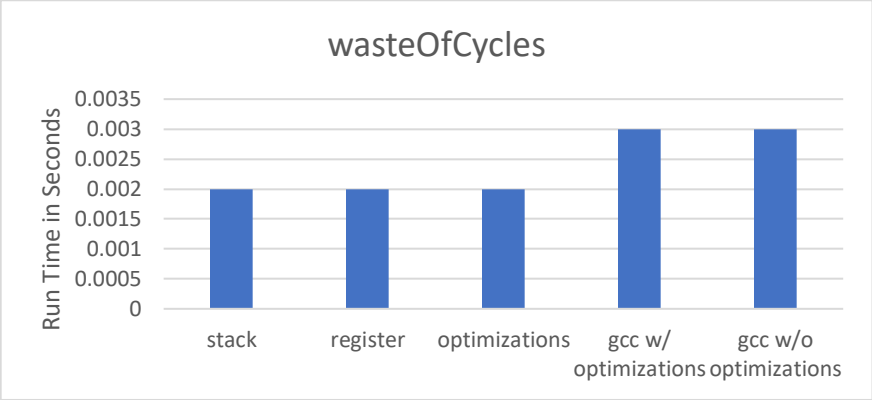




Caroline Cullen
December 22, 2018
CSC 431







Below is a chart of the line count for the register based llvm versus the stack based llvm:

Benchmark Names	llvm lines w/o ssa	llvm lines w/ ssa
BenchMarkishTopics	215	171
Fibonacci	73	60
GeneralFunctAndOptimize	217	176

Caroline Cullen
December 22, 2018
CSC 431

OptimizationBenchmark	1303	723
TicTac	933	878
bert	1047	832
biggest	155	120
binaryConverter	221	164
brett	1151	1029
creativeBenchMarkName	330	238
fact_sum	134	99
hailstone	111	93
hanoi_benchmark	316	270
killerBubbles	278	215
mile1	126	101
mixed	308	235
primes	174	145
programBreaker	151	119
stats	372	295
wasteOfCycles	89	70