CSC 431 - Final Paper

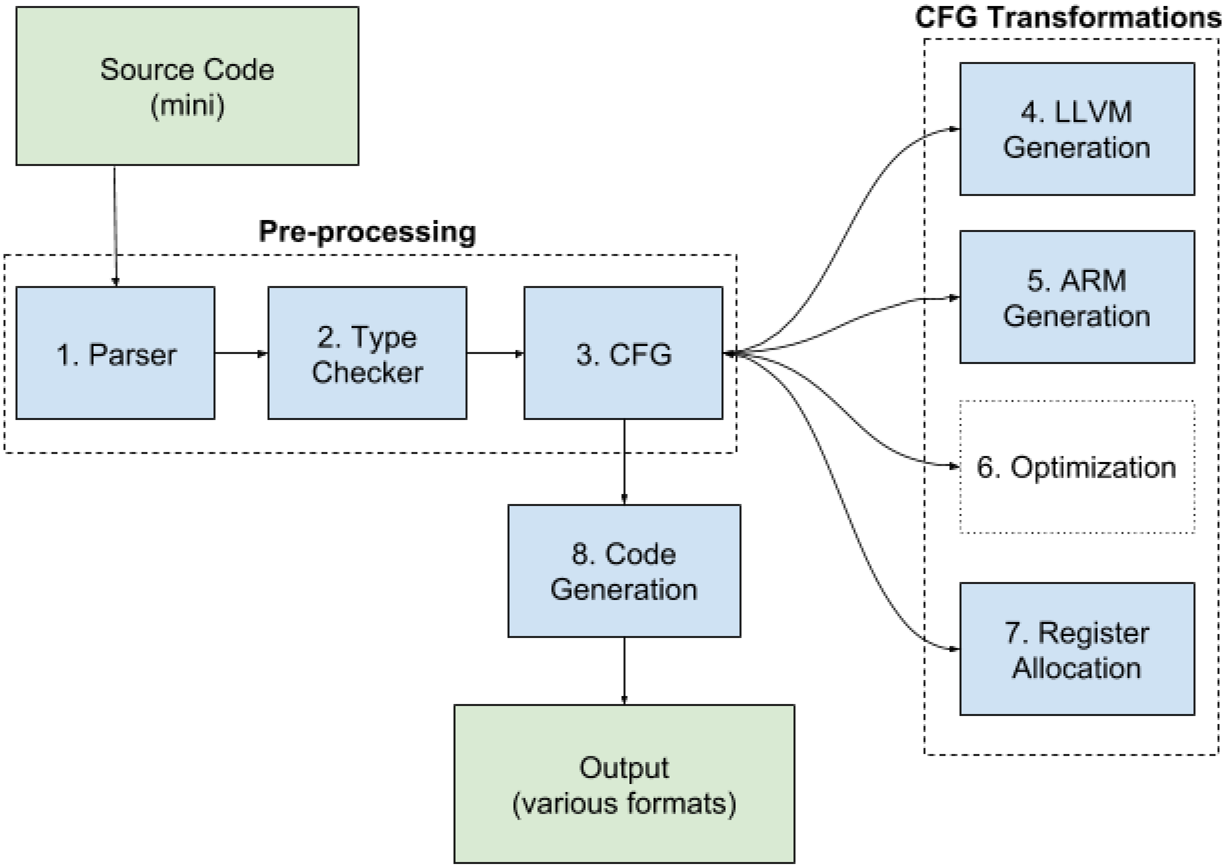
Vittorio Dinovi

The construction of this compiler was a long and arduous process. In hindsight, there are many things I would have done differently. But I'm still pleased with how it turned out. I will now briefly describe key details of the design and implementation followed by a benchmark analysis.

Design & Implementation

I've chosen to implement by compiler in Ruby because it is flexible and high-level lending itself to ad-hoc construction. The Object-Oriented aspect makes the design intuitive and the functional aspect provides simplistic data transformations. And lastly, Ruby has great support for hashmaps with its built-in *Hash* data structure which is used extensively in this project.

The process diagram for compilation is shown below



The input to this process must be valid *mini* source code, and the output may consist of various types of products depending on the flags supplied. The two primary phases of compilation are the *Preprocessing* and *Transformation* phases.

Pre-processing

This phase intakes the source code and performs the following operations in order to produce a proper CFG containing the original AST statements in their corresponding blocks.

1. Parser

Using the provided java parser, it converts the provided source code into its AST representation. Structural checks are performed. The resulting AST is then loaded as a hash into the main program.

2. Type Checker

The parsed AST is then fed into the type checker which does two things: (1) enforces the type rules prescribed in the language grammar and (2) assemble the symbol and structure tables. It will additionally check that all code-paths contain a return statement for functions that expect a return. My implementation of the symbol table includes two separate tables for globals and functions.

It uses the recursive descent algorithm to traverse the AST raising an exception when a type-violation is found. Since Ruby doesn't support outright pattern matching, its implemented as a series of switch-statements which are called into on the way down and return the type and the return status when coming back up. Upon success, this outputs the AST it was provided and the generated tables as hashes.

3. CFG

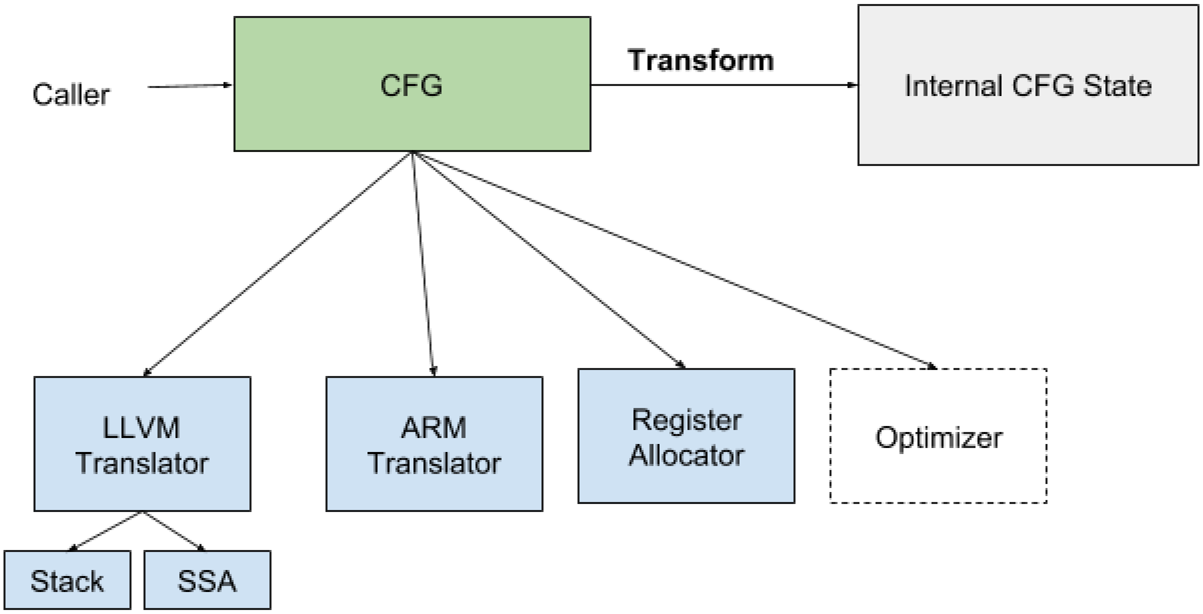
The assemblance of the CFG works by partitioning each functions of the AST into distinct graphs each being composed of blocks containing the corresponding statements directly from the AST. It also houses the symbol and structure tables for reference.

Note that during the construction of the CFG, while and if statements are translated into equivalent branching statements. This simplifies construction later on.

The CFG is the core data structure used throughout the remainder of the construction. This serves the purpose of storing and organizing the blocks. The remaining portion of the compiler is primary composed of applying transformations to each block in the CFG.

CFG Transformations

This set of transformations is best described as *Builder Pattern* that runs through several phases. This process is depicted blow.



The CFG object acts as the director which makes calls to different Builders to transform its internal state.

4. LLVM Translator

The first step in this transformation is translate each block into LLVM, the primary IR for this compiler. The LLVM Translator serves as an interface for two possible translators: Stack and SSA. The choice of which one to use is determined by inclusion of the `--stack` compiler flag. These produce LLVM utilizing the stack or using SSA accordingly.

Each block in each graph is then translated by this translator until all blocks have been filled with appropriate LLVM instructions. These LLVM instructions are represented by LLVM instruction objects.

There are two primary reasons for using LLVM as the intermediate representation. The first is that it is a well-defined abstraction from actual assembly allowing us to work outside of the constraints of any particular machine. In addition, a compiler like clang supports compiling this abstract form into an executable which provides a nice intermediary sanity-check. The second is that it is defined in SSA form which simplifies the def-use chains leading to easier implementation of optimizations.

5. ARM Translator

In the same manner as the LLVM translation, each block is translated into ARM assembly. Note that this ARM representation still includes virtual registers which are then allocated in the subsequent step. These ARM instructions are represented by ARM instruction objects.

6. Optimizations

Unfortunately, for various reasons I was not able to implement any optimizations for my compiler. However, I was still able to learn about them through lecture and appreciate the payoff that they provide

7. Register Allocation

In the case that SSA was used, resolution of the phi instructions must first be done. This simply involves inserting move instructions for each source in the phi instruction into the end of the corresponding predecessor block. These move all of the source registers into the same target register taken from the original phi. This of course breaks the SSA form by inserting multiple definitions for the same register along different code-paths.

Register allocation is done using the live-range analysis and graph-coloring approach discussed in lecture. The empirical allocation technique of pushing registers to a stack in order of likeliness of spill was used as well.

I decided to reserve registers r9 and r10 as dedicated spill registers because it *seemed* simpler than the alternative.

8. Code Generation

To produce the output, all instructions are converted into their string representation and written to the output file.

Analysis

LLVM Comparison

The following table shows the number of LLVM instructions generated. As expected, the instruction count decreases with the use of SSA due to a reduced number of loads/stores even though it introduces additional phi instructions.

Benchmark Name	Stack LLVM Instructions	SSA LLVM Instructions
Fibonacci	61	47
wasteOfCycles	78	55
hailstone	89	66
mile1	111	79
fact_sum	113	77
programBreaker	129	86
biggest	139	94
primes	151	103
BenchMarkishTopics	185	134
binaryConverter	191	117
GeneralFunctAndOptimize	194	145
killerBubbles	256	176
mixed	270	193
hanoi_benchmark	279	215
creativeBenchMarkName	302	203
stats	331	221
TicTac	705	570
bert	917	637
brett	967	813
OptimizationBenchmark	1217	588

ARM Comparison

The graph on the following page shows the runtime comparison between GCC -O0, GCC -O3, stack-based ARM, and ssa-based ARM. The results are summarized in the table below. These were all run remotely on the CSL lab's *Raspberry Pis*. As stated earlier, no optimizations were implemented and so it will not be included in this analysis.

Configuration	Avg. runtime efficiency improvement over Stack-based ARM
SSA ARM	1.2184
GCC -O0	5.0897
GCC -O3	29.7506

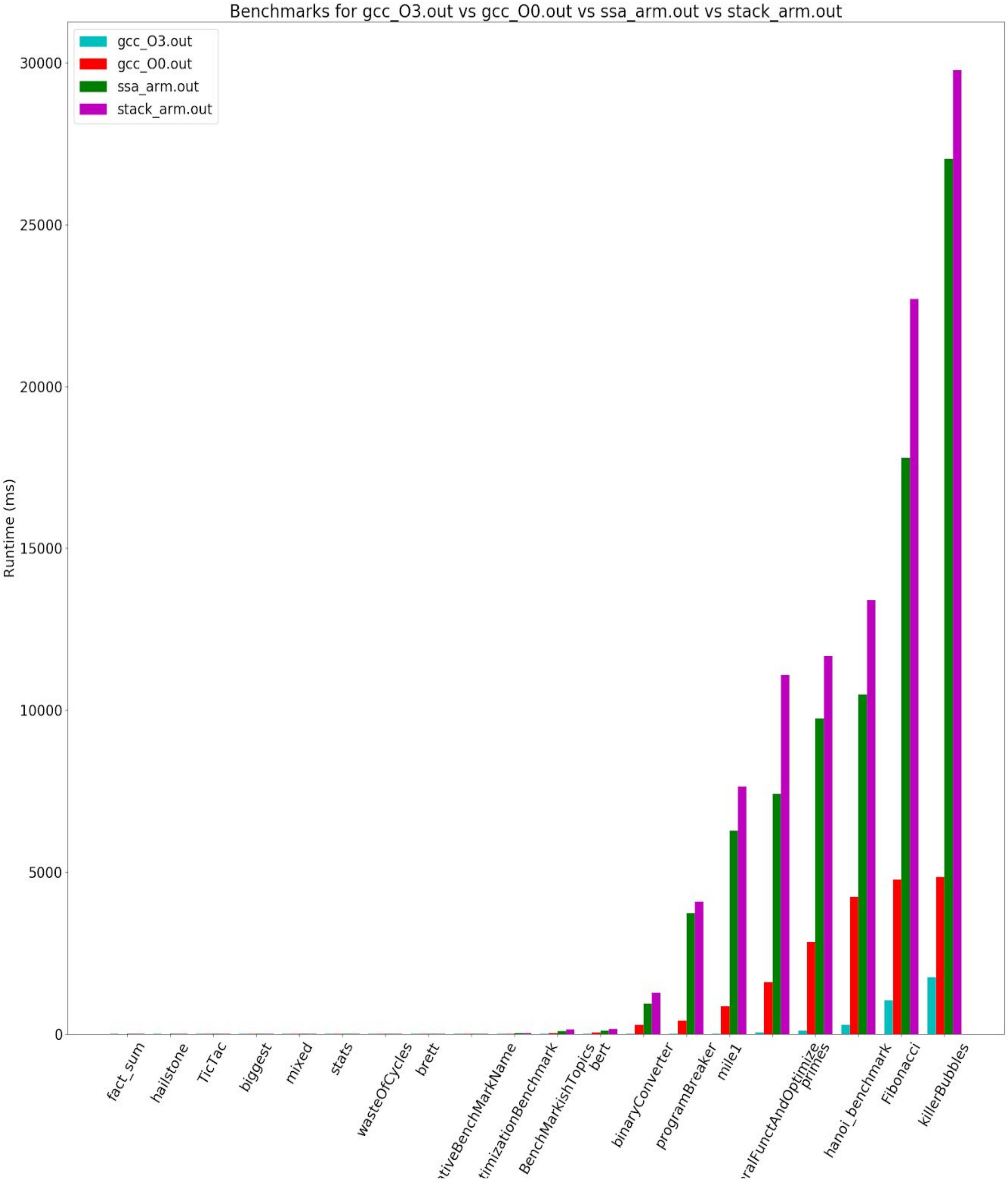
As expected my SSA implementation outperforms the Stack-based implementation. However, GCC without optimizations still outperforms my SSA implementation. This is likely due to inefficient instruction usage on my part (unneeded pushes/pops and others). GCC with all optimizations is wins outright many times over.

A **graphical comparison** of benchmark runtimes is included at the end of this document. The results are as expected, though I'd hoped for my generated SSA ARM to have performed better when compared to GCC without optimizations.

Conclusion

Compiler construction and design is a really challenging. Though we were guided pretty directly through the construction of a very specific compiler, I still found it to be a difficult task. This project has helped me improve my top-down design skills and it also exposed me to some interesting tricks that may be applied in other contexts.

The larger takeaway from this project is that I've become familiar with *some* of the inner-workings of a compiler. I realized this when recently stumbling across a couple blog posts and tech talks regarding language design and optimization that make direct reference to the topics we've discussed and implemented. It's rewarding to be able to recognize these references and understand them.


Benchmarks for gcc_O3.out vs gcc_O0.out vs ssa_arm.out vs stack_arm.out
gcc_O3.out
gcc_O0.out
ssa_arm.out
stack_arm.out
Runtime (ms)
0
5000
10000
15000
20000
25000
30000
fact_sum
hailstone
TicTac
biggest
mixed
stats
wasteOfCycles
brett
tiveBenchMarkName
timizationBenchmark
BenchMarkishTopics
bert
binaryConverter
programBreaker
mile1
ralFunctAndOptimize
primes
hanoi_benchmark
Fibonacci
killerBubbles