Compiler Final Project

Kyle Pfromer, Michael Lucky, and Kaden Ostendorf

May 2, 2022

1 Abstract

Over the course of this class, we built a compiler for a subset of Python to x86 assembly. Our compiler had several issues, foremost among them the inability to run our code on non-x86 architecture. It also struggled with bloated code size resulting from naive implementations of various algorithms. In this project we tweaked our compiler to utilize LLVM IR instead of our own custom x86 IR implementation, to resolve those two issues.

2 Introduction

The problem with our current implementation of Python is that it is heavily dependent on the host system architecture. It requires that the host run an X86 machine that supports 32-bit operations. Smartphones, embedded systems, and even the newer M1 Macs move away from X86 to ARM-based CPUs. We want a portable compiler to produce binaries for various machines and architectures. Additionally, we want our final binary to be optimized. Redundant stores and useless moves should be optimized for a fast, small binary.
In order to compile universal binaries and improve performance, we implemented the Python P0 compiler for LLVM IR. With LLVM, we can apply many optimizations such as dead store and instruction elimination, global value numbering, etc. LLVM is the frontend for many languages like C and Rust that compiles its intermediate representation (IR) into assembly language for a vast set of architectures. It can compile programs to X86 or ARM for 32bit or 64bit machines.

3 Notations

Our Python P0 format has changed slightly. We require that prints have parentheses around the expression. This is required for the Python 3 AST module to parse correctly.
That means our P0 grammar is as follows in EBNF:

⟨*identifier*⟩ ::= [a-zA-Z_][a-zA-Z0-9_]*

⟨*constant*⟩ ::= [0-9]+

⟨*expression*⟩ ::= ⟨*identifier*⟩
  | ⟨*constant*⟩

| ⟨*expression*⟩ '+' ⟨*expression*⟩
| '-' ⟨*expression*⟩
| 'input' '(' ')'
| '(' ⟨*expression*⟩ ')'

⟨*statement*⟩ ::= ⟨*identifier*⟩ '=' ⟨*expression*⟩
| 'print' '(' ⟨*expression*⟩ ')'
| ⟨*expression*⟩
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ '\n' ⟨*stat-list*⟩ | ⟨*statement*⟩

⟨*module*⟩ ::= ⟨*stat-list*⟩

4 Architecture

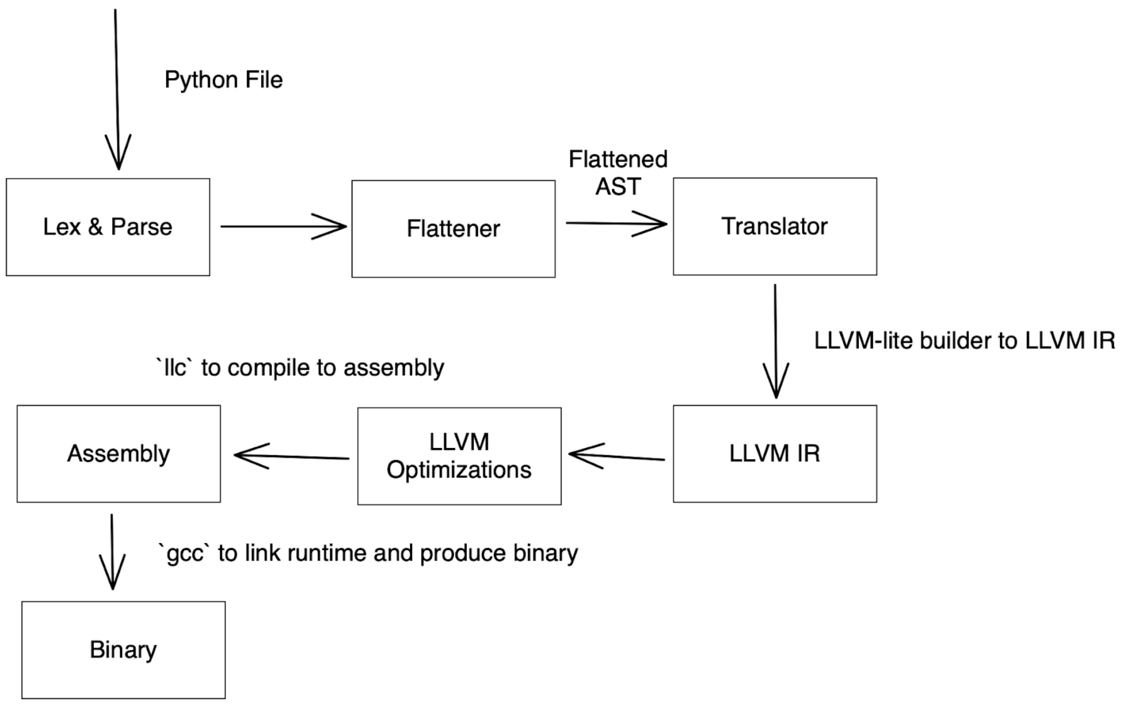
Python File
Lex & Parse
Flattener
Flattened AST
Translator
LLVM-lite builder to LLVM IR
`llc` to compile to assembly
Assembly
LLVM Optimizations
LLVM IR
`gcc` to link runtime and produce binary
Binary

Figure 4.1: Architecture for the compiler.

1. The lexer will handle converting our Python P0 in tokens and then into a common, documented AST format.
2. The flattener will handle flattening our complex expressions into unary or binary operations.
3. The translator will handle converting our flattened version of the AST into LLVM IR.
4. LLVM IR will handle optimizing our IR when converting to assembly.
5. With our assembly we can then link with GCC in our runtime functions (for input and print) and produce a valid binary that we can then run.

4.1 Lexer

Lexing and parsing is handled by Python3's 'ast' library. There were modifications made in the AST module from Python 2 to Python 3. As such we had to adjust our project around the new nodes and systems.

4.2 Flattener

The flattener is still a key component in our pipeline. LLVM requires that binary operators are flattened. A big difference, however, is that binary operators in LLVM do not affect previous values. This actually makes our lives easier since we know that previous references to values **can not** change. This is due to LLVM IR being SSA (single static assignment).

4.3 Translator

We chose to use the 'llvmlite' package to interface with LLVM IR. With the library we just needed to create a few classes and call helper functions on them that dynamically generate instructions.

4.3.1 LLVM lite

'llvmlite' was created and still is maintained by the creators of 'Numba'. 'Numba' is an organization that creates accelerated NumPy by compiling a subset of Python and NumPy to LLVM. They created 'llvmlite' for 'Numba' to make generating IR simple. With 'llvmlite' we don't have to manually write IR strings. With 'llvmlite' we can JIT Python code and quickly initialize a runtime. With the 'ctypes' library we can even create FFI bindings from our compile runtime with the generated LLVM code.

4.4 Optimizations

LLVM will perform a large number of optimizations on our IR to produce a fast small binary. Most of the optimizations LLVM performs are not applicable to a single basic block (since we are using P0 Python all programs will be just one basic block). However, the three notable optimizations it will perform are constant folding, copy folding, and dead store elimination.

4.5 Linking

Similar to the previous compiler that we wrote in class, we created a runtime library in C for the 'input' and 'print' functions. Then we could generate our assembly; using 'llc' we could link our shared library with GCC to produce a valid binary. Then we could run that binary to get the program's results!

5 Optimizations

5.1 Areas of optimization

Programs can be optimized in multiple dimensions, with common ones being: run time, code size, memory usage, and power consumption. For this project we will focus our benchmark criteria on run time of the program by looking at pre- and post-optimization performance.

5.2 Optimization types

Optimizations come in many types and are always evolving. They are categorized into three main sections. First, basic block optimization, regional optimizations, and whole program optimizations. Our focus for optimizations is on simpler basic block optimizations. Basic block optimizations are easy to prove safety with, but lack in performance compared to the scope of other optimizations; they are a good starting point to show differences in optimized vs non optimized code. The optimizations we are focusing on in this project are dead store elimination and copy and constant folding. These optimizations eliminate unused stores to registers or memory and eliminate the recalculation of variables respectively.

5.3 Benchmark procedure

To benchmark these programs we aimed to run each program 10,000 times because the programs are so small that they ran extremely fast and had no determinable difference. Since P0 does not have loops we had to copy and paste code using Vim to get the program to run over and over again. Once we had the test code ready, we utilized a Rust-based CLI benchmark program called Hyperfine that accurately times the running of the code to check performance.

6 SSA Considerations

LLVM IR is a single static assignment (SSA) IR language. This has major implications for our compiler. The compiler must generate SSA compliant code for LLVM to work properly.
One implication is that all "variables" are immutable. Thus in order to change a variable you need to create a new one with it's updated values. That means that we, as compiler builders, need to keep track of variables and how many versions there are. This typically just boils down to adding a number suffix to each variable reassignment: "x = 1; x = 2" becomes "x1 = 1; x2 = 2".
The other challenge is control flow. In order to produce valid control flow code one must include phi functions or use the mem2reg LLVM step to do this for us. This step would handle converting stack variables to register variables with phi functions. However with LLVM mem2reg every time we access or modify a stack variable we would need to load/store. This proved to be a challenge and thus we stuck with Python P0 without control flow blocks.

7 Results

7.1 Constant Folding

Listing 1: Constant Folding Python.

```python
a = 1
b = 2 + 3
c = 3 + --4
d = 4 + 5 + 6

print (a + b + c + d + 1000 + --5)
```

Listing 2: Constant Folding LLVM X86-64 Assembly.

```
        .text
        .file   "constant-folding.ll"
        .globl  main                            # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        pushq   %rax
        .cfi_def_cfa_offset 16
        movl    $1033, %edi                     # imm = 0x409
        callq   print@PLT
        popq    %rax
        .cfi_def_cfa_offset 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .section        ".note.GNU-stack","",@progbits
```

As you can see in Listing 2 LLVM computed the sum of numbers to be 1033. With the older compiler we generated **76** lines of performed assembly, whereas with our new LLVM compiler we generated **5** lines of assembly.

7.2 Copy Folding

Listing 3: Copy Folding Python.

```
a = input()
b = input()
print(a+b)
print(a+b)
```

Listing 4: Copy Folding LLVM X86-64 Assembly.

```
        .text
        .file   "copy-folding.ll"
        .globl  main                            # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        pushq   %rbp
        .cfi_def_cfa_offset 16
        pushq   %rbx
        .cfi_def_cfa_offset 24
        pushq   %rax
        .cfi_def_cfa_offset 32
        .cfi_offset %rbx, -24
        .cfi_offset %rbp, -16
        callq   input@PLT
        movl    %eax, %ebx
        callq   input@PLT
        movl    %eax, %ebp
        addl    %ebx, %ebp
        movl    %ebp, %edi
        callq   print@PLT
        movl    %ebp, %edi
        callq   print@PLT
        addq    $8, %rsp
        .cfi_def_cfa_offset 24
        popq    %rbx
        .cfi_def_cfa_offset 16
        popq    %rbp
        .cfi_def_cfa_offset 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .section        ".note.GNU-stack","",@progbits
```

Again, as you can see in Listing 4 LLVM figured out using GVN/LVN that the second $a+b$ was redundant and replaced its usage with copy (in this case it just reused the same register $\%ebp$). With the older compiler we generated **25** lines of performed assembly, whereas with our new LLVM compiler we generated **15** lines of assembly.

7.3 Dead Store Elimination

Listing 5: Dead Store Elimination Python.

```
a = 100
a = input()

print(a)
```

Listing 6: Dead Store Elimination LLVM X86-64 Assembly.

```
        .text
        .file   "deadstore.ll"
        .globl  main                    # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        pushq   %rax
        .cfi_def_cfa_offset 16
        callq   input@PLT
        movl    %eax, %edi
        callq   print@PLT
        popq    %rax
        .cfi_def_cfa_offset 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .section        ".note.GNU-stack","",@progbits
```

As you can see dead store elimination works as intended since 100 is not included in the LLVM assembly output at all.

7.4 Benchmark of non optimize custom compiler vs optimized LLVM compiler

The goal of this next procedure was to show a usable metric to quantify the difference between optimized and non-optimized code. We ran the programs we were comparing 10,000 times and got the following result:

P0 Program Type	Time in milliseconds
Non-optimized custom compiler	0.4
Basic block optimized LLVM compiler	0.2

As seen from the results, the LLVM optimizations are 0.2 milliseconds faster per 10,000 iterations.

7.5 Compiling for Other Architectures

Although we only showed the resulting X86-64 assembly for the optimizations above we were able to successfully build and run programs on an M1 Mac (ARM based system). The constant folding, copy folding, and dead store elimination passes still ran successfully and the final assembly was small and fast. The only thing that needed to change was rebuilding the runtime C file.

8 Conclusion

8.1 Discussion

The purpose of the this project was to optimize a previously built Python to x86 assembly compiler in two main ways: first, to make it universal to all types of assembly so that it would be processor independent, and second, to make the resulting assembly optimized. We solved this problem by switching our compiler's intermediate language from x86 IR to LLVM IR. This allowed us to leverage the pre-built optimizations and portability of LLVM to create a compiler that would optimally run on most machines. We were successful in this undertaking but also acknowledge the ability to expand upon this basis.

8.2 Open Problems

The next logical step would be to expand our compiler to handle the entire Python language. We would have to change how we assign variable values. All local values and arguments would initially need to be stored on the stack and then LLVM would convert them to registers via an optimization step all "mem2reg". This would ensure that we don't have to handle creating a valid SSA IR with phi functions. LLVM would handle promoting stack variables to registers with phi functions for us. The big considerations we would need to address before moving forward would be how we would handle big pyobjects, tags, closure conversion and control flow.
A big place where we can use LLVM is in the web! LLVM can compile into web assembly. Theoretically we could create a set of conventions that allows JavaScript to interface with Python

and then compile Python in WASM so that a website can use it. This could make the web much more accessible for those who don't have experience with JavaScript.