LLVM Specializer for Python and LLVM Back End Optimizations

Megan Greening
University of Colorado Boulder
Boulder, Colorado
megang@colorado.edu

Kathy Grimes
University of Colorado Boulder
Boulder, Colorado
kathy.grimes@colorado.edu

Aniq Shahid
University of Colorado Boulder
Austin, Texas
aniq.shahid@colorado.edu

Colton Williams
University of Colorado Boulder
Boulder, Colorado
colton.williams@colorado.edu

ABSTRACT

This paper provides an overview of the use of LLVM to optimize a compiler for a subset of the Python language. The output IR from the original compiler was translated into LLVM IR. This IR was then run through several existing LLVM optimizations to confirm that everything was working properly. One analysis optimization was successfully implemented and progress was made toward a transform pass. Based on the preexisting optimizations, the transform pass would have led to a more efficient compiler and shorter machine code as output.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; *Compilers*; Source code generation;

KEYWORDS

LLVM, Compiler optimizations, Python, LLVMLite, LLVM passes, Analysis Passes, Transform Passes, LLVM front-end development

ACM Reference Format:
Megan Greening, Kathy Grimes, Aniq Shahid, and Colton Williams. 2017. LLVM Specializer for Python and LLVM Back End Optimizations. In *Proceedings of (CSCI 5525/ECEN 5523 2017).* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.475/123_4

1 INTRODUCTION AND BACKGROUND

Previous work involved designing a basic compiler for a subset of the Python programming language, called P1. P1 includes print statements, variable assignment, integers, addition, unary subtraction, input calls, comparisons (such as not equal), Or, And, Not, lists, dictionaries, subscripts, and if expressions. This is a very small subset of the Python language and was intended to serve as a basis for exploring compilers and optimizations [6].

LLVM is a compiler infrastructure project. It allows for multi-stage optimizations and it is modular. What this essentially means is that LLVM has a variety of different optimizations. These optimizations (also called passes) can be applied in a variety of orders and combinations. Another beneficial feature of LLVM is the fact that it can work on a variety of input languages. This allows programmers to apply the same optimizations to programs written in different languages[4, 5].

The original compiler for P1, while effective, had a variety of inefficiencies. The only true optimization to the compiler was register allocation, using a fairly straightforward graph coloring algorithm. Since LLVM is a standard way to optimize compilers in industry, it seemed a natural way to try to improve upon the existing work. The goal of this paper was to experiment with the use of LLVM on P1. To this end, preexisting passes were applied to the original compiler. Then efforts were made to build custom passes that could be applied to the previous work [6].

LLVM custom passes, like all passes in LLVM, are subclasses of the Pass class. All custom passes are actually subclasses of of other types of Pass subclasses. For example, the FunctionPass subclasses execute on each function in a program and they can only operate on a single function at a time. Other classes can do things like run executions of loops, look at basic blocks of code, or provide information about the current state of the compiler [5].

In the current work, the output intermediate representation (IR) from the original P1 compiler was modified into an LLVM IR. The IR underwent several built-in passes in order to explore the kinds of improvements LLVM could provide. A custom analysis pass was written that was applied to the generated LLVM IR. This analysis pass, discussed in further detail later in the paper, was intended to be applied to custom transform passes. Due to a variety of difficulties, no new custom transform passes were successfully implemented.

2 IMPLEMENTATION

2.1 Original Compiler

Previous work involved the creation of a simple compiler for the P1 language (a subset of Python described herein earlier)(Figure 1). This compiler was fairly simple. The only optimization passes involved register allocation. Register allocation was accomplished by running liveness analysis on the program variables and then building an interference graph based on that liveness analysis. Finally, the interference graph was colored using a saturation-based, greedy graph coloring algorithm [6].

This compiler was not intended to be used commercially or for any other industry applications, so it was unnecessary to focus

```
key_datum ::= expression ":" expression
subscription ::= expression "[" expression "]"
expression ::= "True" | "False"
             | "not" expression
             | expression "and" expression
             | expression "or" expression
             | expression "==" expression
             | expression "!=" expression
             | expression "if" expression "else" expression
             | "[" expr_list "]"
             | "{" key_datum_list "}"
             | subscription
             | expression "is" expression
expr_list ::= ϵ
            | expression
            | expression "," expr_list
key_datum_list ::= ϵ
                 | key_datum
                 | key_datum "," key_datum_list
target ::= identifier
         | subscription
simple_statement ::= target "=" expression
```

Figure 1: The P1 grammar that the compiler was built to handle [6].

on efficiency in design. Rather it was meant to be used to become familiar with the idiosyncrasies of P1, as well as to explore basic compiler construction [6].

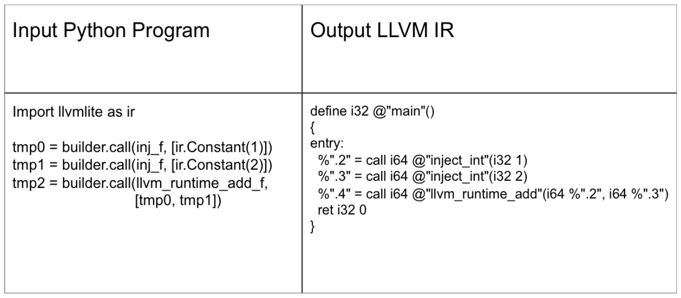
2.2 llvmlite and LLVM Intermediate Representation

Since the original P1 compiler was written in Python, llvmlite was used to update the IR from the original compiler into a format suitable for use in LLVM. llvmlite is a library created by the developers of Numba. llvmlite uses a lot of the LLVM C API. However, llvmlite generates the IR in a new way, using Python. Essentially, the output IR from the original compiler was converted, using llvmlite, into LLVM IR[5].

In order to fully take advantage of LLVM optimizations, the intermediate representation (IR) from the prior P1 compiler needed to be transformed into LLVM IR. This new IR could then be subjected to a variety of optimization passes, both preexisting and custom.

The LLVM IR is strongly typed. This allows for optimizations to act on the IR without requiring any analysis related to type checking. The IR builder itself has many interesting features. The top level building blocks of the IR are functions. Inside of functions are basic blocks and inside of the basic blocks are instructions. The passes can act on these different levels of the IR.

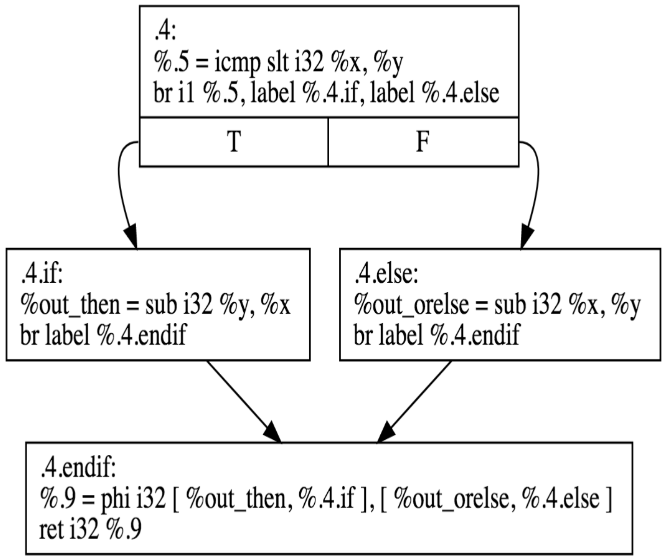
Also of note is the fact that a single variable cannot be defined twice, known as static single assignment form. Due to this restriction LLVM has a number of mechanisms that keep track of variables and arguments across the entirety of a program. This also means that if, for example, a piece of code is tagged for removal it is very easy to check to make sure the variables in that code are not



Input Python Program	Output LLVM IR

Import llvmlite as ir
 	
tmp0 = builder.call(inj_f, [ir.Constant(1)])
tmp1 = builder.call(inj_f, [ir.Constant(2)])
tmp2 = builder.call(llvm_runtime_add_f,
[tmp0, tmp1])
	define i32 @"main"()
	{
	entry:
	%".2" = call i64 @"inject_int"(i32 1)
	%".3" = call i64 @"inject_int"(i32 2)
	%".4" = call i64 @"llvm_runtime_add"(i64 %".2", i64 %".3")
	ret i32 0
	}

Figure 2: A diagram outlining the general organizations of a compiler using LLVM passes.



```
.4:
%.5 = icmp slt i32 %x, %y
br i1 %.5, label %.4.if, label %.4.else
```

```
.4.if:
%out_then = sub i32 %y, %x
br label %.4.endif
```

```
.4.else:
%out_orelse = sub i32 %x, %y
br label %.4.endif
```

```
.4.endif:
%.9 = phi i32 [ %out_then, %.4.if ], [ %out_orelse, %.4.else ]
ret i32 %.9
```

Figure 3: CFG illustrating and example of the way a phi function would be used to handle if expressions in LLVM [2].

used later on in the program. One implication that provided some challenge in the reworking of the original P1 compiler, is that implementation for if expressions had to be modified. In the original implementation, a single variable was used to represent the two possible outcomes of the test. In the LLVM implementation each output was assigned to a separate variable and then only a single was used after the test, determined by use of a Phi function[5] (Figure 3).

2.3 New Compiler Pipeline

Most of the original P1 compiler was left intact. However, the original passes to generate the x86 assembly and do register allocation were removed. These were replaced by a pass through llvmlite that generated LLVM IR. This LLVM IR could then go through one or more passes before finally being translated into machine code (Figure 4).

Also of note is that the original P1 compiler was itself written in Python. However, in order to use LLVM much of the new code had

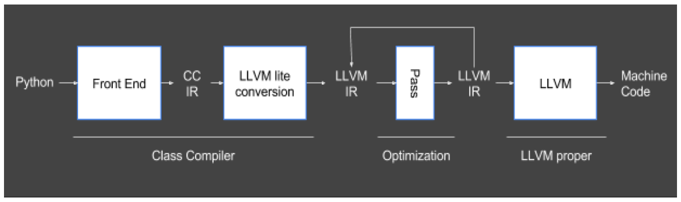


Figure 4: Example of an input Python program and the corresponding output into LLVM IR.

to be written using C++, since that is what LLVM uses. The custom passes that will be described in section 2.5 were written in C++.

2.4 LLVM and Existing Optimization Passes

LLVM has an extensive library of optimizations that can be used in concert with one another or separately. Each optimization that is used is run over the LLVM IR one or more times. Passes can be chained together so that the output of the first pass immediately goes through another pass. Some passes are dependent on one another and must be chained together in a specific order, while others can act independently and be placed almost anywhere in a chain of passes.

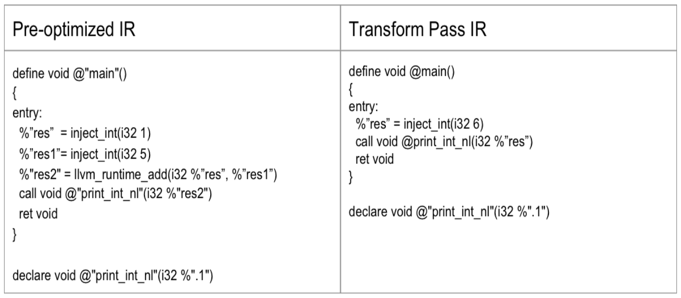
There are two main types of passes available for use in LLVM. The first is an analysis pass. In this type of operation, the LLVM IR is analyzed and useful information is gathered that can be shared with subsequent passes. The LLVM IR is unchanged by an analysis pass. The second type of pass is a transform pass. This type of pass modifies the IR, sometimes with the aid of information from an analysis pass.

In order to explore LLVM and learn about its capabilities, several existing optimizations were studied. Of particular note are InstCombine, ConstMerge, and Dce. InstCombine will combine redundant instructions so as to simply the output code. ConstMerge merges duplicate constants together. Dce is a dead code eliminator that simplifies the output code by removing instructions and other code that is not used in the program. When tested on sample input programs that had features appropriate to the given optimization, all of these passes resulted in shorter and cleaner output code.

Another capability of LLVM is to be either statically compiled or optimized at runtme via just-in-time compilation (JIT)[3]. JIT will preprocess runtime values to optimize out unused branches. This feature leaves cleaner code and fewer assembly instructions. For instance, if runtime values cause an if-else branch of code to always resolve to True, the compare and branching instruction for the else code can be removed.

2.5 LLVM and Custom Optimization Passes

In addition to built-in passes, LLVM also allows for the building of custom passes, which is one of its main features. After becoming familiar with the existing optimizations and learning more about LLVM, several custom passes were attempted. The simplest and most successful of these was an analysis pass, InstCount. InstCount simply counts the number of different instructions in a program. It was hoped that InstCount could be used or modified to help in some custom transform passes.


Pre-optimized IR

```
define void @"main"()
{
entry:
  %"res"  = inject_int(i32 1)
  %"res1"= inject_int(i32 5)
  %"res2" = llvm_runtime_add(i32 %"res", %"res1")
  call void @"print_int_nl"(i32 %"res2")
  ret void
}

declare void @"print_int_nl"(i32 %".1")
```

Transform Pass IR

```
define void @main()
{
entry:
  %"res" = inject_int(i32 6)
  call void @print_int_nl(i32 %"res")
  ret void
}

declare void @"print_int_nl"(i32 %".1")
```

Figure 5: Example of an algebraic transform pass on a simple input program of "print 1 + 5".

There were several ideas for implementing new, simple transform passes such as addition simplification and simplification of multiple chained additions. The idea with this is that if a pass could identify the addition of two or more integers (Const values), then the output machine code be simplified to the result of the addition (Figure 5). It was also hoped that if the simplification of addition was achieved, that an optimization concerning lists could be created. The idea here was to identify small lists and put them on the register or stack space, to avoid slow access to them on the heap.

In the end, some progress was made on a transform pass. InstCount was able to identify the instructions that could be removed for a simplification of addition. However, attempts to write a pass that could delete the old instructions and add a new, simplified instruction in order to mutate the LLVM IR were unsuccessful. This was mainly due to the difficulties encountered in fully understanding the LLVM methods needed to build complex custom passes. If we had finished a transform pass, a way to analyze the usefulness in terms of compile time would have been a special option call 'time-passes'. This LLVM pass would print to standard error the time each pass would take and would be a good comparison towards the overall time taken to compile versus without the new transform pass.

It is also worth noting that one of the best features of LLVM is the ability to reuse preexisting optimizations. Originally the custom pass was going to include a way to eliminate dead code. It became clear, however, that the transform pass could be written is such a way that the preexisitng Dce pass could eliminate the unused code without a need to create a novel dead code elimination pass[5].

3 RELATED WORK

Chris Lattner's original work laid a strong foundation for LLVM and much of what he accomplished still holds true today. LLVM was designed to have a lot of the benefits that come compact intermediate representations and a wide array of available optimizations passes. Essentially, Lattner saw a an opportunity: as improvements to the hardware side of computers have improved, compilers were not fully utilizing the new hardware capabilities. By taking advantage of deep pipelines and parallel execution capabilities, LLVM increases its ability to do analysis and optimization [4].

LLVM has a multi-stage optimization strategy meaning that it can perform optimizations at link-time and run-time. Some passes are done at link-time, while others are done at run-tie. Furthermore, while some passes may be too resource intensive to be done at run-time, LLVM may instead run an analysis pass at run-time and use that information to do a so-called off-line reoptimzation pass after run-time[4].

It has also been shown that LLVM provides many ways to improve upon simple compilers. Previous work indicated that dead instruction elimination, dead code elimination, constant propagation, constant folding, and function inlining (all passes available in LLVM) were effective in improving the performance of a similar compiler. While all of these were initially considerations for custom passes, it was deemed unnecessary to recreate existing passes. Rather the focus was turned to related optimization ideas, such as the simplification of addition[1].

4 CONCLUSIONS

It took longer than expected to get the original compiler translated into a form that would work with LLVM. Once that was done, further experimentation was needed both to utilize built-in customization as well as to design and utilize custom passes. In the end, a single analysis pass was successfully implemented, but no transform passes were completed.

When the project was started it was assumed that gaining a basic proficiency in LLVM would not be too time consuming and that most of the allotted time would be spent designing custom LLVM passes and running analytics. However, the reality was that LLVM was much more complicated to run and use than expected

One of the original plans was to really limit the P1 subset down to the minimum needed to proceed with LLVM. This would have meant that there was no time spent learning how to use the LLVM IR builder. Between learning a new compiler infrastructure, translating the original compiler IR into LLVM IR, and setting up the appropriate environments to run the code, not much progress was made on original work.

In hindsight, more progress could have been made if the IR builder had not been learned and utilized. At the outset of the project, this option was considered. However, for some of the more complex aspects of P1 it would have been difficult to get get LLVM passes to work.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Bor-Yuh Evan Chang for providing guidance on building the original P1 compiler.

The authors would also like to acknowledge Numba for creating and maintaining llvmlite, which was instrumental in creating an IR for out compiler that would work with LLVM.

REFERENCES

[1] 2009. Compiler Optimization Techniques. (Dec. 2009).
[2] SK Lam. 2017. if else. (Dec. 2017).
[3] Siu Kwan Lam and Antoine Pitrou Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. *J. ACM* (Feb. 2015).
[4] Chris A. Lattner. 2002. *LLVM: AN INFRASTRUCTURE FOR MULTI-STAGE OPTIMIZATION.* Master's thesis. University of Illinois at Urbana-Champaign.
[5] llvm.org. 2017. LLVM language reference manual. (Dec. 2017).
[6] Jeremy Siek and Bor-Yuh Evan Chang. 2017. A problem course in compilation: from Python to x86 assembly. (Aug. 2017).