Runtime Instrumentation of Python for Serverless Acceleration

DYLAN FOX and ERIKA HUNHOFF, University of Colorado Boulder

Serverless computing offers many benefits for programmers, but frequently, slower interpreted or JIT compiled languages like Python or JavaScript are used. In their 2019 work, Emily Herbert and Arjun Guha developed a serverless function accelerator (Containerless) that traces and compiles JavaScript to much more performant Rust code. JavaScript, however, is not the only language used in serverless computing. Requiring developers to write in JavaScript will limit the usefulness of Containerless to JavaScript code bases. Here, we provide tracing support for Python, the first step towards adding Python to Containerless. Python is a leading language in Serverless computing and adding support to Containerless for Python will allow many more developers to utilize this technology.

ACM Reference Format:
Dylan Fox and Erika Hunhoff. 2020. Runtime Instrumentation of Python for Serverless Acceleration. *ACM Forthcoming* CSCI 5535, Spring 2020 (May 2020), 8 pages.

1 INTRODUCTION

In their 2019 work, Emily Herbert and Arjun Guha developed a serverless function accelerator framework called Containerless. This accelerator traces JavaScript serverless functions at run-time, and produces an intermediate representation (IR) of the program. The IR is then compiled to Rust code which can be executed in place of the JavaScript function. The overarching motivation of Containerless is to accelerate JavaScript serverless functions because small latencies in serverless execution time can cause discernible lag in serverless applications. A secondary goal is to achieve this without putting additional burden on the serverless developer to either learn a more performant language (i.e. learn Rust) or annotate their code in any way. In fact, whether the function is run as JavaScript or run as compiled Rust is hidden from view by Containerless [2]. Here, we extend their work by providing a tracing mechanism for Python code. Many choose Python for reasons similar to JavaScript: it is simple to write, and many programmers are familiar with it already. While more work needs to be done to integrate Python execution into Containerless, this is the first step towards that goal.

What follows is an overview of the path a Python serverless function takes in our Python-to-IR framework. First, source code must be transformed to A-Normal Form. Transforming the Python code to A-Normal Form resolves differences in scoping between Python and Rust, and simplifies the tracing and compilation to Rust.

Our Python A-Normal Form code follows three specifications:

(1) All variables defined in functions or modules are initialized to None at the top of the function or module they are initially defined in.
(2) All loops are while loops.
(3) All function applications are named.

Authors' address: Dylan Fox, dylan.fox@colorado.edu; Erika Hunhoff, erika.hunhoff@colorado.edu, University of Colorado Boulder.

After A-Normalization, we insert tracing statements into the resulting A-Normal Form Python code. These statements are method calls to our tracing library that are added directly above the corresponding statement in the source code. At run-time, as the Python executes, the tracer library constructs an IR of the code run, and when the function is finished executing the IR is output to a JSON file. After a number of invocations of the Python serverless function, the JSON file with the IR could be compiled to Rust.

2 OVERVIEW

In Containerless, a JavaScript function goes through several distinct phases [2]:

(1) Normalization
(2) Addition of Tracing Statements
(3) Tracing to Produce IR (during executions of the function)
(4) Compilation from IR to Rust
(5) Serverless function invoker dynamically runs either JavaScript or Rust executable

The goal of this work is to create normalization and tracing implementations that can produce compatible IR JSON files from serverless functions written in Python. Since our goal is to avoid any changes to the IR, we are limited in what Python features we can support by what is already supported in the IR in Containerless. Details on supported Python are found in Section 3.

The normalization phase in Containerless transforms the source code for a serverless function to A-Normal Form. This phase is also used to address some of the syntactic differences between JavaScript and Python, so that our tracing can mimic the tracing used by Containerless. In JavaScript variables can be declared (use of the 'var' keyword) and are forcibly declared during normalization which determine variable scope during tracing. Since no analogous statement in Python exists, our normalization framework initializes every variable to None at the beginning of each level of scope. More details on the normalization procedure is described in detail in Section 4.

Next, tracing statements are inserted. These “tracing statements" are method calls to our tracing library. These statements are inserted into the source code so that when the function is run, it incrementally constructs a trace structure. These statements generally are inserted directly above a line of Python source code. During this phase a wrapper function is also placed around the user's serverless function. This wrapper passes the tracing library and the arguments to the user's function, and enables us to run the function outside of a serverless invoker. This allows testing our contributions without modifying the invoker in Containerless. After tracing statements are inserted, the function can be executed. When the function has finished executing, the trace is complete and can be converted to JSON and output to a trace file. Details on tracing are found in Section 5. An example of an IR trace follows in figure 2. Note that during execution of this program, the if branch of the if/else was not executed, so it is left as undefined (as expected).

There are two main categories of evaluation for our work: performance and correctness. Since we focus not on the function invoker or system as a whole but on syntactic definitions, transformations, and tracing, we focus our evaluation on correctness. Because the invoker was not modified, it was impossible to actually compile these traces to Rust. Instead, a number of JavaScript programs were ran in Containerless, and the IR output was compared to similar Python programs. Details on evaluation are found in Section 6.

```python
def double_absolute_value(x):

    double_x = 2 * x

    if x < 0:
        return_value = -double_x
    else:
        return_value = double_x

return return_value
```

Fig. 1. An Example Serverless Function

```
{ "kind":"set",
  "name":"double_x",
  "named":{
     "kind":"binop",
     "op":"*",
     "e1":{
        "kind":"number",
        "value":2
     },
     "e2":{
        "kind":"identifier",
        "name":"x"
     }
  }
},
{
  "kind":"if",
  "condition":{
     "kind":"binop",
     "op":"<",
     "e1":{
        "kind":"identifier",
        "name":"x"
     },
     "e2":{
        "kind":"number",
        "value":0
     }
  },
  "true_part":[
     {
        "kind":"unknown"
     }
  ],
  "false_part":[
     {
        "kind":"set",
        "name":"return_value",
        "named":{
           "kind":"identifier",
           "name":"double_x"  }
```

Fig. 2. An An Abbreviated IR Trace Of The Program In Figure 1

3 SUPPORTED PYTHON SYNTAX

Our goals is to fit within the the framework of Containerless so the subset of Python we support maps directly to a strict subset of the JavaScript supported by Containerless. The Python syntax we support is detailed below.

BinaryOp	op2	::=	+	Addition
			-	Subtraction
			*	Multiplication
			/	Division
			==	Equals
			!=	Not equals
			<	Less than
			<=	Less than or equal to
			>	Greater than
			>=	Greater than or equal to
UnaryOp	op1	::=	-	Numeric negation
Exp	exp	::=	Name	Variable
			Exp(<Exp>*)	Function call
			Exp BinOp Exp	Binary operation
			UnaryOp Exp	Unary operation
			Int	Integer
			Bool	Boolean
			Name = Exp	Variable assignment
			def Name(<Name>*) : Block	Function definition
			return Exp	Return
			if Exp: Block else: Block	If-then-else

4 A-NORMAL FORM TRANSFORMATION

A-Normal Form is program transformation that simplifies compilation of source code. It is more commonly used in functional languages, but can be used over imperative languages too. Generally, A-Normal Form simply refers to having all function applications named [1]. This simplifies compilation because intermediate results are computed separately and named. Our A-Normal Form normalizer preforms this transformation, but it also re-initializes variables and coverts for loops to while loops. It should be noted that due to time constraints, tracing was not implemented for loops. A loop can be normalized to ANF using our normalizer, but it is not possible to trace it yet.

In Python, variables are scoped differently than in Rust. Rust utilizes block scoping, meaning variables are scoped to the block of code they are declared in. If a variable is declared inside a loop, conditional, or other block of code, the variable cannot be accessed outside the block. On the other hand, Python only supports scoping at the function, class, and module levels. So, in Python, all variables defined in while loops, for loops, if/else statements, etc... are available at the nearest class, function, or module level. If directly translated to Rust, this scoping would result in errors if the programmer defined a variable inside a loop, conditional, or other sub-block and then tried to access it outside the block it was defined in. For an illustration, see figures 3 and 4. To preserve this functionality of Python, our ANF transformer creates a variable declaration for every variable at the top of the nearest module or function. The variable declaration simply sets the variable to None so that the variables scope is preserved. If variables are declared at a higher scope (eg. in a closure), we do not reinitialize the variable to None at the beginning of the lower scope, we only change the higher scope. We do not support the use of object oriented programming here, so we don't normalize variables in classes, focusing on functions instead.

```
def function():
    if True:
        x = 1
    x += 1
```

Fig. 3. Python Code With Lifted Variable Definition – Runs Properly

```
fn function():
    if (true){
        let x = 1;
    }
    x += 1;
```

Fig. 4. Rust Code With Lifted Variable Definition – Produces Compilation Error Because x Is Not Defined

Our ANF-Transformer also converts all for loops to while loops. In Rust, for loops are actually quite similar to Python for loops. Both languages use for loops that iterate over an iterator and do not support direct C style for loops. Both Rust and Python also have iterators that can be infinite. To avoid the possibility of infinite looping, for loops are converted to while loops, and we only support iterating over index-able finite collections.

```
def double_absolute_value(x):

    double_x = 2 * x

    if x < 0:
        return_value = -double_x
    else:
        return_value = double_x

    print(return_value)

    return return_value
```

Fig. 5. Source Code For An Example Serverless Function Before ANF Normalization

```
def double_absolute_value(x):

    double_x = None
    return_value = None
    app0 = None

    double_x = 2 * x

    if x < 0:
        return_value = -double_x
    else:
        return_value = double_x

    app0 = print(return_value)

    return return_value
```

Fig. 6. Source Code For An Example Serverless Function After ANF Normalization

5 INSERTING TRACING STATEMENTS

To insert the tracing statements into a Python function, we first insert an "import" statement to import the expression objects utilized by our tracing library. Next, for every statement, we import the correct tracing function. For example, a call to the trace_set function must proceed any python code that sets the value of a variable. When tracing statements are inserted into functions, the body of the function must also have tracing statements inserted into it. To capture the arguments, calls to the trace_let function (which defines a variable) must be inserted at the top of the function for every parameter. When if statements are traced, separate statements are inserted into the if block and the else block. During execution, only the block that is actually executed is traced.

Since a new serverless invoker was not built, we insert a wrapper function around the user function. This enables us to pass a tracing module (to allow the use of different tracing code if desired) to the function being executed. This wrapper is directly called by our runner, similar to how a serverless invoker would invoke a serverless function. After this wrapper function is inserted, the function can be executed to produce a trace. See figure 7 for a program after ANF normalization and insertion of tracing statements.

One challenge to highlight is that JavaScript functions in Containerless are run by Node.js which encapsulates a lot of functionality. Creating a realistic Python run-time (even outside of the invoker) that fully mimics the JavaScript run-time then becomes complex: we would have to emulate a portion of the functionality of Node.js in Python in order to create truly identical traces. The overhead associated with creating this would have detracted from our focus on translation from Python to IR, and it's also unclear if - from the invoker's perspective - having a Node-like Python run-time would even be desirable. While we opted to use the simple function wrapper described above, creating a Node.js-like run-time for Python functions may be a necessary step in future work to allow for more rigorous comparison between JavaScript and Python traces.

```
import tracing.exp as exp
def serverless_function(trace, args):
    def double_absolute_value(x):
        x000 = trace.trace_function_body('ret00')
        trace.trace_let('x', x000)
        trace.trace_let('double_x', exp.UndefinedExp())
        double_x = None
        trace.trace_let('return_value', exp.UndefinedExp())
        return_value = None
        trace.trace_let('app0', exp.UndefinedExp())
        app0 = None

        trace.trace_set('double_x',
            exp.BinaryOpExp('*', exp.NumberExp(2),
            exp.IdExp('x')))
        double_x = 2 * x

        test =  exp.BinaryOpExp('<',
            exp.IdExp('x'), exp.NumberExp(0))
        if x < 0:
            trace.trace_if_true(test)
            trace.trace_set('return_value', exp.UnaryOpExp('-',
                exp.IdExp('double_x')))
            return_value = -double_x
        else:
            trace.trace_if_false(test)
            trace.trace_set('return_value', exp.IdExp('double_x'))
            return_value = double_x

        trace.trace_function_call('app0',
            ['print', exp.IdExp('return_value')])
        app0 = print(return_value)

        trace.trace_break('ret00', exp.IdExp('return_value'))
        trace.exit_block()
        return return_value

    trace.trace_function_call('result',
        [exp.IdExp('double_absolute_value'),
        exp.IndexExp(exp.IdExp('args'), exp.NumberExp(0))])
    result = double_absolute_value(args[0])
    return result
```

Fig. 7. ANF Transformation and Tracing Statement Insertion of Code From Figure 5

6 EVALUATION

Since this work did not encompass modifications to the invoker (Containerless does not run Python functions in any capacity) it was not possible to run performance tests using the Containerless infrastructure or to accurately test Rust compiled from our generated IR. Thus, our evaluation focuses on correctness of the produced traces.A series of test JavaScript and Python functions were written. The JavaScript functions were traced in Containerless, and the Python functions were traced in our system. The resulting IR traces were inspected to determine if they were similar.

A tangible result of our choice to write a runner as a temporary stopgap until future work can be done to integrate Python into the invoker is seen in the difference in IR in the initial call and in the return (response). IR generated from JavaScript calls function via a callback, whereas IR generated from our Python runner is invoked by a direct function call. Similarly, IR generated from JavaScript terminates via the respond function, whereas our Python functions use a typical return statement.

Ignoring the above (expected differences) in the outermost and innermost layers of the traces, we can still observe that the bodies of serverless functions produce near identical IR. This manual process leaves something to be desired, but more rigorous testing would require coupling with components of Containerless that were beyond the scope of this project. The github repo for this project (https://github.com/csci5535-s20/project-containerless-py) contains a collection of examples written in both Python and JavaScript with corresponding transformed code and traces.

7 RELATED WORK

Our work builds on the concepts and code presented in Herbert and Guha [2], which was built upon the principles presented in [4]. [2] focuses on the entire process of creating a JavaScript to IR framework, an IR to Rust compilation process, and a serverless function invoker compatible with OpenWhisk [3]. Our work is more narrow, and seeks to modify the JavaScript-to-IR phase into a Python-to-IR process.

Work has been done towards formalizing a semantics for Python [5]. However, our work is specifically geared towards the subset of Python needed to write serverless functions. This is in contrast with existing work which focuses on either creation of general purpose tools or solutions for a particular case (other than serverless).

8 CONCLUSION

In this work, a system for executing python at run-time to produce traces for Containerless is presented. Python code was normalized and had tracing statements inserted into it as in Containerless. In a simplified run-time environment, the transformed code was executed and produced JSON traces similar to analogous JavaScript functions. This tracing mechanism is the first step towards making Containerless accessible to Python developers. Next steps would include creating a more rigorous Python run-time, creating a Python invoker, and more rigorous testing.

REFERENCES

[1] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. *SIGPLAN Not.* 28, 6 (June 1993), 237–247. https://doi.org/10.1145/173262.155113

[2] Emily Herbert and Arjun Guha. 2019. A Language-based Serverless Function Accelerator. arXiv:cs.DC/1911.02178

[3] Baldini Ioana, Castro Paul, Cheng Perry, Fink Stephen, Ishakian Vatche, Mitchell Nick, Muthusamy Vinod, Rabbah Rodric, and Suter Philippe. 2016. Cloud-Native, Event-Based Programming for Mobile Applications.

[4] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 149 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360575

[5] Gideon Joachim Smeding. [n.d.]. An executable operational semantics for Python. http://gideon.smdng.nl/2009/01/an-executable-operational-semantics-for-python/. Accessed: 2020-03-05.