Variable Isolation and Interpreter Reuse in Python

Alex Curtis, Stephen Sturdevant, Riccardo Mutschlechner

{acurtis, sturdeva, riccardo}@cs.wisc.edu

July 9, 2017

1 Motivation

Our project stemmed from some prior research ongoing at Wisconsin, on the OpenLambda project. OpenLambda, or more generally, serverless computing, is a new paradigm for running applications in the cloud without the rigidity of a monolithic server or VM. Compared to a monolithic approach (i.e. spinning up a new VM, waiting a few minutes, and then using it just to run one binary), we have quite a few things to be gained in the serverless model. Mainly, we gain elasticity - one can easily scale up or down without needing to worry about actual servers or VMs. Latency, however, is not great on a cold start, which we will discuss later. Thus, our research question is: by sharing a Python interpreter, can we reduce cold start latency and still provide isolation? Ultimately, we were able to implement a checkpoint and restore mechanism to provide isolation while dramatically (roughly 6000x) reducing the cold start cost, while still providing moderate isolation. In the following sections we will discuss some background knowledge, followed by our implementation, and lastly our results and conclusion.

2 Background

To start off, we would like to expand on some background knowledge of serverless computing. When we say ”serverless” computation, we do not mean the lack of a server, but rather the fact that it is abstracted away from a developer. In serverless computing (i.e. AWS Lambda or OpenLambda), a function is your basic unit of computation. You are charged per function invocation, billed in 100ms increments, rounded up. This is truly a ”pay as you go” model, much more finely-grained than other options, such as AWS EC2’s hourly billing. Another advantage here is a smaller binary, which leads to less time to ship, and therefore faster to start new ”instances”. The server is shared across lambda functions, which is what gives such great elasticity for scaling upwards and downwards - one does not have to wait for
a server to spin up. Again, the main advantages are the lower cost (as it is more finely-grained), as well as the better elasticity.

Next, we will provide background on the cold start problem. It takes approximately 20-40ms to start a Python interpreter, as well as roughly 20ms per module import for the larger modules such as *numpy*. Short computation on a cold start leads to much of our time spent being pure overhead and not so much actual computation. Latency-sensitive applications will surely suffer from this. Our idea from these observations is to enable sharing to reduce cold start time. However, we still need to provide moderate isolation upon doing so.

When designing isolation, we have a few specific goals in mind. Our primary goal is to avoid any drastic changes to the Python language itself - increasing the critical path for development will surely set bad precedent and cause developers not to want to adopt our work. We would like performance to remain close to optimal, and will show our performance in comparison to the optimal (i.e. simply running lambda functions back to back with zero isolation) case. Each function should have (almost) no way to tell it is sharing anything with other functions. Lastly, although our threat model is on the conservative side of things, we would like to discuss security more in-depth later on as a consideration that should not be taken lightly in such work.

We evaluated three main ideas, plus the optimal case, and compared and contrasted their isolation versus speed. The first idea is a fork-on-invocation model shown in Figure 1; for each function call, fork off a new process and simply run the lambda function there. Having a separate process provides great isolation. Having copy-on-write pages also leads to more sharing than simply doing a fork and exec (i.e. running `python` from bash), which is also positive. Lastly, this is very simple to implement and requires no changes to the Python language itself. Secondly, we evaluated a reload-on-invocation model shown in Figure 1; for each function call, we reload all shared modules from that function prior to running it. This uses the same process, so the isolation is not fantastic, but it is acceptable in some cases. Also from using the same process, we are sharing much more than both the fork-and-exec model as well as the fork-on-invocation model. Again, this requires no changes to Python at all and is simple to implement. Lastly, the approach which we implemented ourselves: a checkpoint-and-restore model, shown in Figure 2. Here, prior to calling a lambda function, we call a `Checkpoint()` function that we built into the Python interpreter. This notifies the interpreter that we are about to make some changes that should be able to be discarded. Thus, after the lambda is complete, we call a `Restore()` function, which undoes all of the changes - including to object fields, rather than just modules. This approach has more sharing than any of the three past models we mentioned, which leads to improved performance. This model has less isolation than the fork or fork-and-exec models, but more isolation than the reload model. Lastly, the changes to the Python language were minimal - one simply needs to use those two function calls from above. This
model is very simple to use, although it was nontrivial to implement.

(a) Fork Model

```python
#!/usr/bin/python
import os
import time
import lambda_func

start = time.time()
for i in range(1000):
    rc = os.fork()
    if rc == 0:
        lambda_func.f()
        exit()
    elif rc < 0:
        print "error!"
    else:
        continue
for i in range(1000):
    os.wait()
end = time.time()
print end - start
```

(b) Reload Model

```python
#!/usr/bin/python
import os
import time
import lambda_func

start = time.time()
for i in range(1000):
    lambda_func = reload(lambda_func)
    lambda_func.f()
end = time.time()
print end - start
```

(a) Checkpoint & Restore Model

```python
#!/usr/bin/python
import os
import time
import lambda_func

def Checkpoint():
    lambda_func._CHECKPOINT_ = 1

def Restore():
    lambda_func._RESTORE_ = 1

start = time.time()
for i in range(1000):
    Checkpoint()
    lambda_func.f()
    Restore()
end = time.time()
print end - start
```

3 Implementation

There are a few ways that checkpoint-and-restore could be implemented in Python. However, this discussion will focus on the approach that was actually taken which we believe is optimal. First, we will discuss some necessary Python background information. Then, we will discuss how this was used to actually implement checkpoint-and-restore.
This discussion will focus on Python 2.7.12 and specifically CPython, the default implementation of Python. As the name suggests, it is written in C. Each Python module contains a global dictionary, which contains a mapping from object name to object value for each object at the top level of the module. That object value can be a primitive or a reference to a more complicated structure. These more complicated structures can contain references to other objects. These references are generally called attributes, and they are stored in a dictionary internal to the original object to which they belong. This global dictionary can thus be seen as the root of a tree containing all the objects in the module. This distinction between an object’s attributes and the object itself is important. Python provides some built-in namespace protection for the object itself. If an object \( x \) exists, and a different module tries to assign a value to \( x \) as in \( x = 4 \), a new object \( x \) is actually created in that different module and the \( x \) in the first module is left untouched. This is not true for attributes. If an object \( y \) has an attribute \( z \), and a different module has visibility of \( y \), then the original \( z \) can itself be changed.

With that background information in mind, it becomes noticeable that what really needs additional protection is attributes. How do we protect attributes? Python code is broken down into bytecode which is then executed by the interpreter. There is a bytecode instruction, \( \text{STOREATTR} \), which handles all attribute assignments. When the interpreter is processing a \( \text{STOREATTR} \) instruction (ceval.c), it calls into some other code which handles this at a PyObject level (PyObject_SetAttr in object.c). This actually boils down to another function (PyObject_GenericSetAttrWithDict in object.c). This code actually sets the attribute value. Immediately before an attribute is set, we want to note the attribute name, its current value, and the object the attribute belongs to. With this information, we can later restore the attribute. How do we store this? We keep a special per-interpreter dictionary (truly global, as opposed to global within a module only). At the base level of this dictionary, a key is an object which contains at least 1 attribute that has been changed, and that key’s corresponding value is a dictionary which maps attribute names to their old values. This dictionary is initialized once, and then lazily populated as attributes are changed. It’s important to note also that if an attribute’s value is being changed but it already has an entry in this dictionary, we do not want to modify this per-interpreter dictionary further as we want to keep track of the old value when it is first modified. This describes how we keep track of changes. Restoration is simply a recursive traversal of this dictionary, pulling out the object, and setting its attribute (referred to by name) back to the old value.

There are a few remaining questions. We do not always want to be keeping track of attribute changes. We only want to track it between two set points in the code. How do we do this? As mentioned in the prior sections, we provide a \( \text{Checkpoint}() \) and \( \text{Restore}() \) function that is available in the Python code. These each correspond to an underlying function in C (see object.c) which are invoked from Python itself whenever any attribute is set to a certain value. This decision was primarily made in the interest of time, and is not ideal. A better approach could be to actually add a
new bytecode instruction. Furthermore, any user can invoke these two functions, so we would want to introduce some notion of privilege so that lambdas cannot access them.

4 Results

In comparing our approach with the others mentioned previously, we found that checkpoint and restore is much more efficient while still providing good isolation. See slides for more information on the exact results. While the isolation from checkpoint and restore is not as good as through forking a new process, we believe with a few more modifications we could get to that point.

5 Conclusion

We’ve shown that checkpoint and restore is a viable solution to isolation within a single process running a Python interpreter. This can be used to achieve greatly reduced cold start latency for lambdas. While this outcome is desirable, the approach outlined is only viable when limited to strict Python. Embedded C code is allowed in Python and frequently used for high-performance code; this approach does not provide the necessary isolation under such circumstances. We would need OS support to achieve this isolation. However, prior research into single address space operating systems shows that isolation is possible within a single address space [1], and we believe this topic should be revisited.

6 References