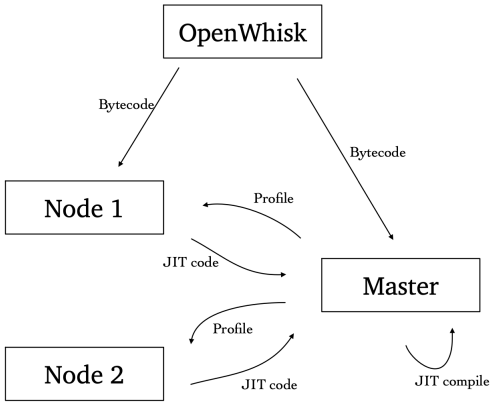
Design & Implementation Draft of JFass

Yuantian Ding

December 11, 2021

Overview

Currently, we design our framework of JFass as follows:



In JFass, we use the master node to manage the profiles, JIT-compiled code, and other data in each data. The master node collects enough profile information and performs JIT-compilation according to a certain compilation policy. Non-master nodes are responsible for handling the function tasks sent from OpenWhisk. All the non-master nodes need to synchronize the profile information with the master node from time to time during the execution. After the compilation is complete, the master node sends the JIT-compiled binary code to all the nodes that need it. Also, both non-master nodes and master nodes need bytecode to execute/compile a function. Therefore, OpenWhisk needs to send the function's bytecode to both non-master and master nodes before executing the function.

In the following sections, we will explain:

- How we manage to send JIT code, profiles across multiple nodes.
- When to send profiles and JIT code information between master and non-master nodes.

1 Code Sharing Between Nodes

Relocating JIT code across different machines is not an easy task, as JIT compilers often directly embed pointers and addresses into jitted code with little relocate information. Therefore, we have to thoroughly research the JVM compiler's source code and check every possibility of address embedding.

According to our research, there are three categories of addresses in the JVM compiler that need to be relocated: Object Pointer, Metadata Pointer, and Runtime Address.

- Object Pointer: a pointer to an object instance in Java runtime. Object pointer are allocated in GC, so they may be changed during the runtime.
- Metadata Pointer: a pointer to class and method data including profile data, bytecodes, and constants. Java metadata is not always static and can be dynamically loaded by Java's ClassLoader.
- Runtime Address: static addresses inside the JVM, includes stub routines(runtime specific subroutines), internal tables, and string messages.

Object pointers are the hardest to relocate. They may change during Java runtime, and it's tough to trace them. However, most object pointers used in jitted code are constants (static final variables). Constants are easy to trace as you can send a static field's name directly. Also, there are object pointers that are hard to trace: trusted non-static final variables. As they are very rare, we disable this feature in JFass.

Metadata pointer can be relocated by disabling the ClassLoader feature in FaaS functions. Since most FaaS functions don't need to load new classes in their execution, we decided to disable the ClassLoader feature in JFass, which guarantees all class data can be found in pre-defined class loaders.

Runtime Addresses are the easiest to relocate. But they are relevant to JVM source code, which means you need to change hundreds of runtime addresses in the source code. Relocating runtime addresses requires a lot of tedious work. Maybe we can leverage program analysis to relocate every runtime address in the source code.

2 Profile Sharing and Merging

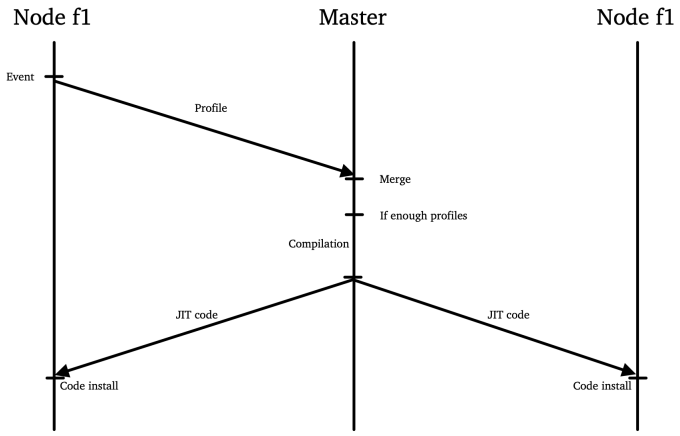
Profile sharing is not a tough task as it does not require more relocation than code sharing. However, unlike code sharing, profile sharing requires the merging of profiles from different nodes. There are three categories of profile data that needs to be merged in execution:

- Function Counter: includes invocation counters (counter for function invocation times), and backedge counters (counter for loop execution times in a certain function).
- Branch Data: includes how many times the branch instruction was taken or not taken.
- Virtual Call Data: how many times and which class called a virtual method.

We can merge all profile data by just adding up all the counters. However, as HotspotVM only records two most frequent classes in virtual call data, we may miss some infomation in virtual call data, which we will explore further.

3 Code and Profile Sharing Control

We design our control flow as follows:



When the non-master node exceeds a certain execution threshold or needs deoptimization, it triggers an event that sends the current profiles to the master node. Master node merges the local profiles with the received profiles. Once it finds enough information in the collected profiles, it will compile them according to the JVM's tiered compilation policy. Considering that the nodes processing the same task will use similar libraries, the master node sends the JIT code to all nodes executing the same FaaS function after the compilation. After receiving the JIT code, the node immediately installs the code into the JVM runtime and runs them when needed.

4 Implementation

We implement the whole framework based on Hotspot JVM and Openwhisk. Networking is implemented on GRPC and Scala Akka (OpenWhisk required).