JFass: A New Serverless Platform with Customized JVM Runtime

ABSTRACT

Faas (Function as a service) is one form of serverless computing where users only need to deploy function on the cloud and the cloud service vendor will handle all the hardware resources the function needs. The vendor will provide runtimes for these functions which are driven by events. However, Faas is tailored for short-lived functions which makes traditional runtime optimizations such as JIT compilation fail to enhance performance. We want to design a new framework to bridge the gap between modern language runtime and serverless platforms.

In this paper, we aim at building a new serverless platform tailored for JVM runtime with profile information sharing and native code sharing across nodes. We also hope to use hardware tracing technology to help reduce profiling overhead from interpreter stage.

CCS CONCEPTS

• Software and its engineering -> Compiler • Software performance

KEYWORDS

serverless computing, JVM, tracing

1 Introduction

Serverless Computing is gaining popularity for its low cost which provides a “pay as you go” code execution platform. Faas (Function-as-a-Service) is one form of serverless computing. Faas abstracts away the complexity of resource management which means programmers do not need to worry about deployment, but only need to upload code to the Faas platform and the service vendor will do everything else necessary such as underlying resources provisioning, process management, etc. Programmers divide an application into small isolated pieces of code and each piece is called a function (or a logic unit), function will be uploaded to the service provider which are triggered by events. Faas make this possible by offering computing runtimes such as JVM and functions run under isolation boundaries such as containers. However, this simplification may lead to performance slowdown when compared to traditional platforms. This arises from the fact that high-level language runtimes will dynamically collect profile information and generate optimized code for hot blocks which are executed many times. Bad news is that Faas functions are usually small units of code. To make it worse, Faas functions are limited in how long each invocation is allowed to run by the vendor and most profile information is discard after the function is finished. [6]

As is known, Java can be considered both an interpreted and a compiled language because Java source code will first be translated to bytecodes and Java Virtual Machine (JVM) will interpret them and then compile them into native code when JVM figures out they are worth compiling from statistics collected during runtime. Take HotSpot for example, there are some types of counters such as invocation counters, back-edge counters to describe the runtime information of a method. They can start from the interpreter stage. HotSpot kicks off two different Just In Time (JIT) compilers, the client (known as C1) and the server (known as C2) to get faster, efficient machine code. The C1 compiler has a low compilation pre-defined threshold to reduce startup time and C2 compiler has a higher threshold and can generate more optimized native code to enhance performance by adapting more aggressive compiling methods to those may be in the critical execution path of the program based on the profile information collected during runtime. By doing so, Java can get better performance as native code runs faster than interpreting. But this is based on the assumption that the compiled part will continue executing many times and the saved time can compensate for the compilation phase and interpreter phase because the overhead introduced by compilation cannot be ignored. The framework of Faas may contradict to this assumption because most functions on Faas are short-lived and most critical blocks are not executed on optimized code or the compilation overhead is bigger than its income. This arises from the fact that profile information is distributed on different nodes which will make it hard to exceed the pre-defined threshold on a single node or cause repeated compilation on every node. [4]

Intel Processor Tracing (PT) is a new feature that will expose an accurate and detailed trace of activities of Intel Processor. It mainly traces the execution of branch instructions by recording them in the data packets, then these packets will be stored in the disk. PT has been used in various areas such as testing, performance analysis and debugging. We can use PT or other hardware tracing technologies to assist profiling collecting in JVM runtime to slowdown overhead by transforming the overhead to offline.[5]

Based on the facts above, we plan to build a new serverless platform tailored for JVM runtime with profile information sharing and native code sharing across nodes assisted by hardware tracing technology.

2 Related work

Mohammad first characterized the entire production Faas workload of Azure Functions and they found that vast majority of functions have a maximum execution time of 10 seconds and vast majority of applications are invoked infrequently. This implies that Faas are tailored for small functions and traditional server-based runtimes may not be suitable for directly being deployed to Faas. [1]

Joao shows modern serverless platforms do not fully leverage runtime optimizations and a significant number of function invocations running on warm containers are executed with unoptimized code. They said they were implementing a new serverless platform with profiles sharing and code across nodes sharing and the work is still under implementation today. Inspired by the workshop they published on HotOS'21, I plan to extend their work and design a new serverless platform tailored for JVM runtimes. [2]

Zhiqiang presents JPortal, a JVM-based profiling tool that bridge the gap between high-level language applications and low-level hardware traces. By precisely designing algorithms, JPortal can recover the program's control flow graph from the PT traces recorded during runtime. Based on their work, I can design new profiling pattern assisted by PT. [3]

3 Challenges

- Portability

  HotSpot will generate machine code by inserting a lot of runtime information directly to the instruction which will make it difficult to reuse the code on other machines. To make it worse, C1 and C2 will conduct different optimization-level strategies which make portability harder. No previous work has ever tried to solve this, so it is worth time designing some new algorithms for code portability because compilation compose an overhead which cannot be ignored during execution.

- Deoptimization

  To make the machine code smaller and faster, HotSpot will use some very aggressive optimization strategies. For example, for a branch instruction, the compiler will directly assume that fewer taken one will not be executed next time and discard it. So how to properly add logic for native code to return back to interpreter is a little difficult.

- Profiles Integration

  Different stage will outcome different form of profiles. In HotSpot, there are three stages for each function: interpretation, native code generated by C1 compiler and native code generated by C2 compiler. How to integrate profile information produced by different stage is still under consideration.

- Code-transmission Overhead

  Though native code runs faster than interpreter, transmission overhead cannot be ignored because it is likely that compilation is faster than waiting for the code to come. To make it worse, traditional network stack will waste a lot of time repeating copying. Some new technology may assist reducing the overhead such as RDMA.

- Security

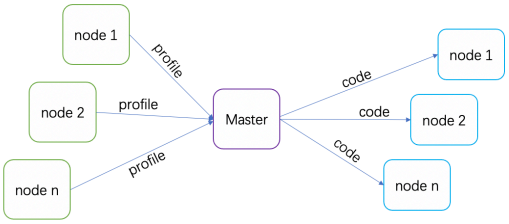
  Sharing profiles and code may impose a threat for attack. For example, attacker can conclude some runtime information from the native code and can make the entire platform crash. This means there must exist a limitation for the range of profile and code sharing.

- Asynchronous Operation

  Sharing code and profiles during runtime asynchronously can add a lot difficulties to the framework design and make it hard to maintain.

4 Overview

Figure 1 shows the components of the new framework. There are two types of nodes, master node and normal nodes. Master node is responsible for integrating profile information from other nodes and generating native codes and handing out them. When an event arrives, a new normal node will be called to handle this event. Every time a normal node finishes, it will send profile information back to the master node. The master node will integrate profile information of different stages and generate native code when it finds some code blocks are 'hot' which means it is on the critical path of a program. Once the master generates native code, it will send it to normal nodes.



5 Discussion

Here I list some threats that may lead to failure of my research.

- Hardware-Tracing Assistance

  Hardware tracing technology is not mature nowadays and it is not as precise and accurate as software tracing such as instrumentation or other tools. To make it worse, decoding the packets generated by hardware costs a lot of time which will make it hard to generate code in time.

- Transmission Overhead

If Transmission overhead is larger than compilation locally, the work will make no sense since just follow the traditional pattern will gain much income.

6 Conclusion

In this paper, we draw up a new serverless platform tailored for JVM runtime with profile information sharing and native code sharing across nodes to bridge the performance gap by co-design between modern language runtime and serverless platform. We also hope to use hardware tracing technology to help reduce profiling overhead from interpreter stage.

REFERENCES

[1] Shahrad, Mohammad, et al. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider." 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20). 2020.

[2] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: leveraging runtimes for the serverless era. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21). Association for Computing Machinery, New York, NY, USA, 58–64. DOI:https://doi.org/10.1145/3458336.3465305

[3]Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: precise and efficient control-flow tracing for JVM programs with Intel processor trace. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 1080–1094. DOI:https://doi.org/10.1145/3453483.3454096

[4] OpenJDK. Compiler. https://wiki.openjdk.java.net/display/HotSpot/Compiler

[5] Intel. Processor Tracing. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html

[6] IBM. FaaS (Function-as-a-Service). https://www.ibm.com/cloud/learn/faas