Neutrino: Lightweight Sandbox for Serverless Java with AOT Compilation and VM Isolates

Kangcheng Xu
k57xu@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Yixin Yang
y39yang@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Yandong Zhu
y268zhu@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

ABSTRACT

Serverless computing has shifted the paradigm of building cloud-native applications. By encapsulating on-demand computation as forms of stateless functions executed on underutilized hardware, serverless computing offers application developers a cost-efficient and highly elastic platform for both development and deployment. With automatic resource provisioning and scheduling, developers could quickly develop applications without worrying about scalability and saving money by not paying for idle hardware. One major challenge with serverless computing is to reduce request latencies, especially the cold start latency. To safely evaluate untrusted user functions on shared hardware, function code must execute in controlled and isolated sandboxes. However, any form of isolation adds overhead to processing requests, and typically, stronger isolation is associated with greater latency. Reducing serverless function latency could be achieved by tweaking the existing or designing new sandboxes, especially for serverless workloads.

Traditional hypervisor-based and container-based sandbox systems for generics systems are intrinsically not suitable for the serverless workload. The application-agnostic nature of these systems comes at cost of more runtime bloats which further degrades overall system performance. This paper introduces *Neutrino*, a novel sandbox runtime designed especially for serverless Java applications. Neutrino largely reduces serverless overhead by adopting *ahead-of-time* (AOT) optimization and executing user functions in independent *isolates* within one shared process-VM to reduce further latency caused by application initialization. We also perform AOT optimization to compile Java functions directly into machine code, producing an initialized application image, to skip initialization and textitjust-in-time (JIT) compilation warm-ups. Instead of allocating and initializing system resources to provide a generic operating system-like environment independently for every sandbox, Neutrino shares the runtime among all sandboxes. Isolation is implemented with lightweight process-VM primitives called isolates. The evaluation shows that Neutrino offers faster cold start latency and leaves less memory footprint compared to some state-of-the-art sandboxes.

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

serverless computing, cold start, Java, ahead-of-time optimization, operating system;

ACM Reference Format:
Kangcheng Xu, Yixin Yang, and Yandong Zhu. 2022. Neutrino: Lightweight Sandbox for Serverless Java with AOT Compilation and VM Isolates. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Serverless computing has been widely adopted by cloud-native software development to speed up development and lower operational costs. Many cloud computing providers have offered their serverless platforms, including most noticeably, Amazon Lambda [8], Google Cloud Function [3], and Azure Functions [2]. With the provided serverless runtime and other off-the-shelf automatically provisioned infrastructures, developers can build scalable applications with fractions of time and costs compared to the traditional Infrastructure-as-a-Service approach. With serverless computing, core application logic is encapsulated into stateless functions for ephemeral execution on shared hardware upon an event (most commonly, a user request) is triggered. Because no dedicated hardware is sitting idle while not serving requests, serverless computing allows high hardware utilization and developers only pay for resources used. The stateless nature of serverless function also implies high scalability as multiple copies of the same function could be deployed if demand increases.

Since functions are not provisioned with dedicated hardware, one function must not alert the global system state, either a software state or a hardware state, which could potentially affect the execution of another function. Most importantly, each function should not have access to data belonging to other functions. Therefore, untrusted user functions must be executed in isolated environments called sandboxes. Most existing efforts implement sandboxes by emulating a system with functions that will execute. Hypervisor-based and container-based sandboxes offer good security and performance isolation by emulating physical hardware or software systems which will be used for executing functions, but they have greater performance overhead. This overhead does not only result in wasted computing power but also increase the request latency, especially cold start latency in the case of the first request. Initializing a sandbox takes time, and oftentimes the initialization takes longer than processing the actual request. Cold start latency is a major disadvantage serverless computing has over the traditional architectures. Of its negative impact on user experience, efforts are spent tweaking the existing or designing new sandboxes, especially for serverless workloads.

The goal of this project is to reduce the system overhead, mainly focusing on improving cold start latency, by designing a novel serverless sandbox for Java applications. This paper presents Neutrino, a novel sandbox runtime designed especially for serverless Java applications. On top of the latency introduced from initializing the sandbox, serverless Java applications have an even longer cold start latency due to the cost of initializing the Java Virtual Machine (JVM) language runtime. Neutrino improves cold start latency with two strategies. First, we adopt ahead-of-time (AOT) optimization that turns Java bytecode directly into executable native machine code. We use GraalVM Native Image technology to produce already-initialized and ready-to-execute binary function images as shared libraries. Second, we implement sandboxes with lightweight VM isolates instead of containers or hypervisors. With Graal SubstrateVM, the cost of creating a VM isolate, and therefore initializing a sandbox, could be comparable to spawning a new thread. Since Neutrino only runs one process with an arbitrary number of isolates at any time, together with AOT optimization, we also reduce memory pressure which is a common problem for Java applications.

We implemented Neutrino based on GraalVM SubstrateVM and Native Image and modified *OpenLambda* [7] to support this novel sandbox. We measure the performance improvement with microbenchmarks from both latency and a memory footprint perspective. The result shows Neutrino can achieve less than 10 ms startup latency consistently at load on a simple Java workload, and we also observe a much smaller memory footprint even compared to state-of-the-art serverless sandbox like SOCK [9].

The main contributions of the paper are as follows:

- A detailed survey on GraalVM technologies (§2)
- A general design of VM isolate-based serverless sandbox running AOT optimized serverless functions (§3)
- An implementation of Neutrino with Graal SubstrateVM and Graal Native-image built for the OpenLambda serverless platform (§4)
- An evaluation of Neutrino shows improved cold start latency and reduced memory footprint compared with other sandboxes (§5)

2 BACKGROUND AND MOTIVATION

2.1 AOT Java and Graal Native Image

Java Virtual Machine is a heavy-weight process that comes at a high startup cost, making it unsuitable for high-performance cloud applications where requests must be processed in a timely fashion with limited resources. Traditionally, Java source code is compiled into bytecode and loaded into a JVM for execution. At first, the JVM executes bytecode similarly to an interpreter. Once the number of execution of a Java function is high enough to reach a set threshold (known as "warm-ups"), the JVM will use a just-in-time (JIT) compiler to perform a profile-guided optimization that lowers Java bytecode into a format that allows more efficient execution (i.e., most commonly the native machine code). However, in typical serverless settings, sandboxes are frequently destroyed and recreated, and JIT-optimized code cannot retain through the process, making performance improvement from JIT less significant.

Ahead-of-time compilation advances machine code generation to compile time. Other than skipping waiting for functions to warm up, AOT compilation also performs some application initialization (i.e., static initializers) that would otherwise be performed at the cold start time. AOT compilation also reduces memory footprint since unused code, references, symbols and metadata are stripped during closed-world analysis, yielding a smaller program size and more tightly packed object structures. Wimmer et al. [11] evaluated that AOT optimization techniques lead to faster startup, lower memory usage, and better peak performance, making Java a viable choice for microservices and serverless cloud functions.

Ahead-of-time compilation for Java is already a common practice in the industry, especially for cloud computing. Quarkus [10] is a notable example of a cloud-native, container-first framework for writing Java applications. Quarkus is powered by the same technologies used by Neutrino, namely GraalVM Native Image [5]. However, Quarkus does not differ from traditional Java applications architecturally. One Java process still only executes on application, while security and performance isolation is achieved with generic external isolations (e.g., containers).

Mainly developed by Oracle Labs, GraalVM [4] is a suite of Java runtime-related tools aiming to improve the performance of Java applications mainly by AOT compilation and reducing memory footprint. Native Image builder is a utility that processes all classes of an application and their dependencies, producing a native image. A native image is a Java application AOT compiled targeting a specific architect and OS type. A native image bundles everything it needs at run time into a single executable.

2.2 VM Isolates and Graal SubstrateVM

SubstrateVM [6] is a new lightweight JVM implemented as AOT-compiled Java code. Substrate VM takes the place where HotSpot normally sits. It contains components like an optimizer, memory management, thread scheduling and communication with the host OS. Substrate VM itself is partly AOT-compiled Java runtime classes (written in Java) with native callouts to the JVM library swapped to its own native implementation. Due to the nature of application code being AOT compiled, Substrate VM does not support features that complicate AOT optimization. For example, reflective accesses have to be explicitly configured, and dynamic class loading is not supported with Substrate VM. Additionally, features like class validation and type checks are also stripped since they can be performed equally well at compilation time. By not including runtime bloats typically seen on a traditional JVM, SubstrateVM performs well with limited system resources.

Another feature introduced by SubstrateVM is VM Isolates. Isolates are independent VM contexts that share the same VM process. For most cases, isolates could to considered as multiple VM instances sharing the same underlying system resources and code. Each isolate has its own dedicated heap and creating a new isolate also reverses and assigns a new heap [11]. Because isolate heaps are disjoint, and Java is a managed language where raw pointer arithmetics and differences are not possible, user functions from one isolate cannot access memory belonging to another isolate, making isolates the perfect building blocks for serverless sandboxes.

3 ARCHITECTURAL DESIGN

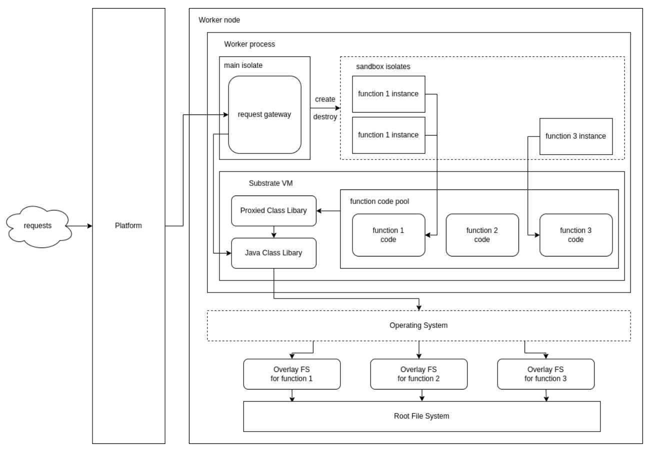


Figure 1: Architecture Overview

The Neutrino runtime is designed as single processing serving multiple requests with in-process sandboxed isolates on a serverless worker node. This worker process will spawn multiple isolates within sharing the same SubstrateVM. The main isolate is used to handle all requests forwarded from the platform. The main isolate is also responsible for loading the function code into the registry without executing it. In our design, we define a cold start as the workflow caused by a request whose corresponding serverless function is not yet loaded into the process. In case of a cold start, Neutrino first loads the native function image from a shared library (i.e., a .so file) into the function registry with dlopen() system call. Notice Neutrino only keeps one copy of the function code globally, as the immutable function code is shared between multiple function instances. Once the function code is loaded into the process, the following workflow is the same as warm starts. During warm starts, a new worker isolate is created every time handling a request. Once the response is computed, the isolate is immediately destroyed to prevent unnecessary garbage collection. Neutrino does not keep isolates cached or pre-heated.

3.1 Sandbox Design

Neutrino uses Graal SubstrateVM isolates as the basic building block for sandboxes. Isolates are much more lightweight to create compared to VMs. The cost to create an isolate is comparable to creating a thread.

In Substrate VM, each sandbox is an isolate that exists independently in the same process with separated heaps. One typical problem in traditional Java is that all tasks share the same heap, even for technologies like Java EE containers, which are designed to provide isolation between untrusted code. With isolates, cross-heap accesses are not possible, and one function can only access objects within its own heap. That is, cross-heap access is not allowed even if two tasks are running in the same process. This is enforced as Java does not allow pointer arithmetic or raw pointer dereference. While isolates' disjoint heaps provide a fundamental base for memory safety, it also makes passing requests and responses between the main isolate and the worker isolate difficult. Neutrino always serializes requests and responses into binary buffers for easier copying between isolates.

Isolates also mitigate the performance impact of garbage collection. In traditional Java, once a task allocates many objects, the garbage collection it triggers could affect all other tasks in the process since there is only one heap and memory management is global. In a serverless setting, garbage collection caused by one function could impact the performance of other functions. Isolates solve this problem by performing independent garbage collection. Additionally, Neutrino creates new sandbox isolates every time handling a request and immediately destroys them once requests are processed. Isolates are not pre-heated, cached, or reused. While this may seem inefficient, such a practice could avoid some costly garbage collections entirely. If an isolate is no longer needed, Neutrino tears it down directly and releases the one continuous heap memory as a whole without performing any garage collection. The low setup and tear-down costs justify not keeping isolates cached or reused.

3.2 Compiling Function Code

Due to security concerns, function code must be AOT compiled into native-native shared libraries by the serverless platform. While it may be tempting to let function developers compile their code locally and upload it to the code to avoid the costly AOT optimization process that performs close-world analysis, doing so could lead to an isolation breach. Since we rely on isolates to provide memory safety, we must make sure the code is compiled from Java, a language without the ability to perform raw memory access. A malicious user could embed instructions to generate native images that perform sensitive operations that would otherwise be prevented by the VM. Similarly, Neutrino should not allow functions to execute any native code which could make arbitrary memory accesses or direct system calls, essentially breaking the isolation.

Function developers could upload a JAR file containing compiled JVM bytecode (class files). The platform will compile this JAR file to a native-image shared library to be installed on worker nodes. However, precaution is needed for two reasons. First, AOT compilation with Graal is extremely resource demanding as performing close-world analysis on a typical application could use several gigabytes of memory and takes several minutes. Second, the compilation itself must happen in an isolated environment as some function code (e.g., static initializers) are executed and could be potentially malicious. This could be achieved with any existing external sandboxes (e.g., fully isolated hypervisors). Compilation performance is not generally concern as it happens when a user updates the code.

3.3 Security Model

Graal SubstrateVM isolates alone only guarantee memory isolation between sandboxes. F function could still affect the global system states with standard Java APIs. However, since Java is a managed language, Neutrino could limit untrusted code's privileges by imposing limits on APIs it can call. Therefore we identified four categories of APIs for taking different actions.

- `prohibited` APIs: directly manipulating the host system's states and cannot be emulated
- `restricted` APIs: manipulating the host system's states but could be controlled by emulation
- `sensitive` APIs: revealing critical system information that could be used for more advanced attacks
- `unregulated` APIs: all other APIs

This paper proposes a Java bytecode transformer to preprocess serverless function code before compiling them to native code. Since the platform would compile the serverless function code, the bytecode will inspect function calls to the standard libraries, and intercept these calls to a proxy function handler taking different actions according to API categories.

Disable Prohibited APIs. Some APIs provided by the Java Class Library are not rather rarely used in serverless workloads. For example, access to TCP raw sockets is usually not used and there could be resource conflicts if two function instances try to listen on the same port. These non-essential APIs are not useful for serverless functions and expand the attack surface.

To disable certain API, we need to patch the runtime API to disallow certain functions. The bytecode transformer replaces function calls with prohibited API with calls to Neutrino-provided functions having the same signature but always throwing exceptions upon execution.

Emulate Restricted APIs. Restricted APIs are those commonly used but affect the system states. This type of APIs must be emulated so that impact on the global system state could be managed. An example is file system APIs. It is common for a serverless function to perform disk reads and writes, but a function should not access files belonging to another function, and functions should not be able to modify files that impact other system components' behaviours (e.g., /dev directory).

Similar to disabling prohibited APIs, we would perform bytecode transformation so calls into restricted APIs are redirected to a patched proxy. The patched version will emulate the APIs so the system state remains unchanged (while it appears to the serverless function otherwise). For emulating file systems, we could create an Overlay File System for each user function with the base layer being the actual root file system. When writing or deleting a file, the patched file system calls will update the path to the function's own file system instance, so the base file system is not changed.

Desensitize Sensitive APIs. Sensitive APIs are those very commonly used but could be used as the stepping stone for more advanced attacks. An example is that access to a high-precision timer makes it possible for malicious users to perform time-based side-channel attacks.

In this example, the high precision timer result must be desensitized so it returns a less precise result. Since serverless functions are typically short-lived, and an isolate is created for each function execution, we could make the timer always return the same result every time the timer function is called. This time could be the time when the request was received, or the time when the timer function is called the first time.

Passthrough Unregulated APIs. APIs that do not belong to the abovementioned three categories are unregulated. The bytecode transformer should not alter function calls to these APIs.

4 IMPLEMENTATION

Neutrino is implemented in two parts, both are publicly available on GitHub. The Neutrino repository [1] is mainly implemented using Java. It contains our sandbox runtime, a few C-bindings for Java, and a function handler template that serves as a boilerplate for writing Neutrino serverless functions.

We also fork an existing serverless platform, OpenLambda and embed Neutrino runtime into it [2]. Neutrino extension support as well as several benchmark scripts we used for evaluation is also included in the corresponding branch.

5 EVALUATION

We evaluate the Neutrino sandbox in terms of four aspects: startup latency, memory usage, CPU usage and concurrency level. We benchmark the cold start and warm start latency, and memory usage with Docker, SOCK, and Neutrino. This section describes the evaluation methodology and results for each aspect.

5.1 Environment Setup

We use an x86-64 system with a 6-core 12-thread Intel(R) Core(TM) i7-9750H CPU (@ 2.60GHz), 32 GB of RAM and a Samsung 970 EVO Plus NVMe SSD. The operating system is ArchLinux (2022.06.01) with Linux kernel 5.18.13. The GraalVM toolchain version is 22.1.0 CE (Java 11.0.15). At the time of evaluation, the Neutrino version was at commit `#5f0903b`, the OpenLambda version was at commit `#06951bb` of our fork. We compare Neutrino against Docker runtime and SOCK runtime shipped with OpenLambda at the same version.

5.2 Methodology

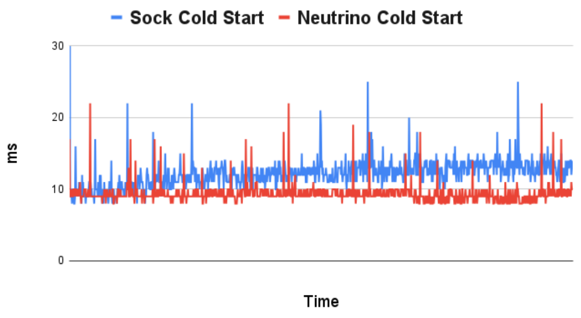
We install a Java function and a Python function to OpenLambda as the testing applications. The Java function is called when the sandbox type is set to Neutrino, and the Python function is for testing the Docker and SOCK runtimes. They use different functions because Docker and SOCK runtimes do not support Java. Both functions do the same thing, that is, simply return the payload of the user request. During the experiment, all the requests have the same payload: `{"hello": "world"}`, which represents the minimal workload. In order to obtain the accurate request process time, we wrote a Python script to send the HTTP request and record the timestamps before the request is sent when the TCP three-way handshake is done and after it receives the response. The startup latency is estimated by calculating the time difference between the two timestamps. Since both the OpenLambda worker and the user are running on the same machine and the workload is simple, the network communication time and the application execution time can be neglected. The main time-consuming process will fall into sandbox and runtime initialization. We also wrote a script to constantly capture the total memory used on the testing machine every half a second for comparing memory usage for different runtimes.

The following are the steps to evaluate cold/warm start latency and memory usage for each runtime:

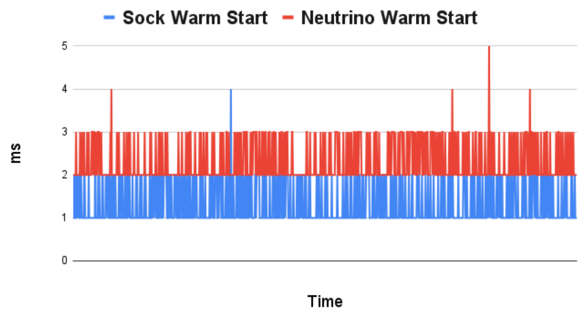
[1] https://github.com/tabjy/neutrino
[2] https://github.com/Shawolyyx/open-lambda

Table 1: Comparison of cold start and warm start latency among SOCK, Neutrino and Docker

	Cold start latency			Warm start latency		
	SOCK	Neutrino	Docker	SOCK	Neutrino	Docker
Average time (ms)	12.270	9.377	963.674	1.323	2.268	32.853
Max time (ms)	106	22	5752	4	5	368
Min time (ms)	8	8	554	1	2	24
Median time (ms)	12	9	853	1	2	29



(a) Cold start



(b) Warm start

Figure 2: Comparison of how cold start and warm start latency changes over time between SOCK and Neutrino

1. Create 1000 copies of the testing function with different function names
2. Start the script to record the total memory usage
3. Start a OpenLambda worker with the sandbox type to be tested
4. Send 1000 sequential requests using the Python script calling the 1000 functions with different names
5. Send another 1000 sequential requests calling the same set of functions
6. Kill the OpenLambda worker and stop the memory recording script

The first 1000 requests are for acquiring the cold start latency as OpenLambda will create a new sandbox and install a new function upon the first invocation of each function. The second 1000 requests are for testing the warm start latency as OpenLambda will reuse the existing sandboxes.

5.3 Startup Latency

5.3.1 Cold start latency. Table 1 shows the summary of the cold start and warm start latency for SOCK, Neutrino and Docker. Docker endures the longest cold start latency with an average of 963.674 ms and requires 554 ms even in the best case. SOCK and Neutrino reach a similar result. Specifically, Neutrino improved the cold start time by 25% compared to SOCK, from 12.270 ms to 9.377 ms on average. Figure 2a depicts the latency distribution among 1000 cold starts for SOCK and Neutrino. It can be seen clearly that SOCK tends to become slower as the number of sandboxes increases, possibly due to the increased usage of CPU and memory that are discussed in the later sections §5.4 and §5.5. In contrast, Neutrino stays fairly stable at around 9 ms.

Table 2: Comparison of cold start memory usage among SOCK, Neutrino and Docker

	SOCK	Neutrino	Docker
Memory used (MB)	5343	778	15426

5.3.2 Warm start latency. Table 1 shows that Docker performs the worst with 32.853 ms on average, so we will mainly focus on SOCK and Neutrino. From Figure 2b, we can see that both SOCK and Neutrino have quite stable warm start latency which fluctuates between 1-2 ms and 2-3 ms respectively during the 1000 warm starts. The main reason for the difference is that Neutrino significantly reduces the cold start latency by replacing JIT with AOT compilation. It still requires a tiny startup time for each function invocation. This is the trade-off between small cold start latency and slightly larger warm start latency.

5.4 Memory Usage

We evaluate the memory usage for SOCK, Neutrino and Docker by monitoring the total memory used on the testing machine during the first 1000 function invocations as there are not notable memory changes during warm starts. The total memory used after spinning up 1000 sandboxes is shown in Table 2. Docker requires the longest cold start time as well as up to 15 GB of memory for initializing 1000 sandboxes. SOCK performs better than Docker but it also consumes 5 GB of memory for initialization. Neutrino only uses 778 MB of

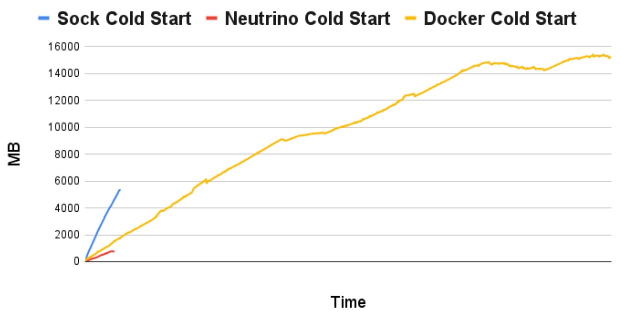


Figure 3: Comparison of how memory usage changes over time among SOCK, Neutrino, and Docker

memory for the whole cold start procedure. Figure 3 illustrates how the memory usage changes over time for the three types of sandboxes. Neutrino shows the smallest upward trend compared to the other two. The result also reveals that the improvement of Neutrino in memory usage provides a stable cold start time that will not increase as the number of sandbox initialization increases.

5.5 CPU Usage

During the evaluation experiment, we also notice that both SOCK and Docker require more CPU resources. Unfortunately, we do not find a reliable tool to monitor CPU usage. We notice that the CPU usage goes up to 100% during the SOCK and Docker execution period, while Neutrino only uses around 47% CPU. The result indicates that the SOCK increasing cold start latency is also related to the higher CPU usage.

5.6 Concurrency Level

We attempted to measure the concurrency level that SOCK and Neutrino can handle using Apache Bench [1]. Nevertheless, during the benchmark process, we noticed that request failures started to happen once the number of concurrent requests reached 34 for both SOCK and Neutrino using OpenLambda. We suspect that there are some scheduling issues in the OpenLambda implementation that caused this limitation. Therefore, we are not able to perform a concurrency evaluation at this moment. It is worth noting that Neutrino itself is capable of handling thousands of concurrent requests without failures.

6 FUTURE WORKS AND LIMITATIONS

6.1 Security Model

Due to time constraints, we did not implement the security model proposed in the architecture design (§3). However, since the bytecode transformer is a standalone component of Neutrino, we do not expect any coupling with the existing code base. The design is also generic and could be used in other scenarios while a Java program's abilities must be controlled.

6.2 Language Support

By the current Neutrino version, we only support the applications written by Java, while other JVM languages like Scala and Kotlin could be easily supported with minor tweaks. And it is still possible to execute other language applications like Python and JavaScript with Truffle Language Implementation Framework. The framework essentially compiles code into a universal and JIT-optimizable graph that can be interpreted by SubstrateVM.

6.3 Streaming Request/Response

Since cross-heap memory access is strictly prohibited, even the main isolate that handles the incoming requests cannot access worker isolates. They can only share data by serializing the entire request/response into a string and passing that to the worker isolates. Streaming request/response is therefore not supported at this point.

7 CONCLUSION

GraalVM provides AOT native image compilation to JAVA applications with negligible startup overheads. Neutrino explores a new type of sandbox by embedding GraalVM. Benefiting from Neutrino, serverless functions can reduce 25% and 99% cold start latency, and 85%, and 95% memory usage compared to SOCK and Docker respectively.

REFERENCES

[1] 2022. AB - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/current/programs/ab.html
[2] 2022. Azure functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview
[3] 2022. Cloud functions. https://cloud.google.com/functions
[4] 2022. GraalVM. https://www.graalvm.org/
[5] 2022. Native Image. https://www.graalvm.org/native-image/
[6] 2022. Safe and Efficient Hybrid Memory Management for Java. https://www.oracle.com/technetwork/java/jvmls2015-wimmer-2637907.pdf
[7] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson
[8] David Musgrave. 2022. Lambda. https://aws.amazon.com/lambda/
[9] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes
[10] Guillaume Smet and Max Rydahl Andersen. 2022. Supersonic subatomic java. https://quarkus.io/
[11] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: Application initialization at build time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360610