CloudJIT: A Just-in-Time FaaS Optimizer (Work in Progress)

Serhii Ivanenko
serhii.ivanenko@tecnico.ulisboa.pt
INESC-ID/Técnico, ULisboa
Lisbon, Portugal

Rodrigo Bruno
rodrigo.bruno@tecnico.ulisboa.pt
INESC-ID/Técnico, ULisboa
Lisbon, Portugal

Jovan Stevanovic
jovan.j.stevanovic@oracle.com
Oracle Labs
Belgrade, Serbia

Luís Veiga
luis.veiga@tecnico.ulisboa.pt
INESC-ID/Técnico, ULisboa
Lisbon, Portugal

Vojin Jovanovic
vojin.jovanovic@oracle.com
Oracle Labs
Zurich, Switzerland

Abstract

Function-as-a-Service has emerged as a trending paradigm that provides attractive solutions to execute fine-grained and short-lived workloads referred to as functions. Functions are typically developed in a managed language such as Java and execute atop a language runtime. However, traditional language runtimes such as the HotSpot JVM are designed for peak performance as considerable time is spent profiling and Just-in-Time compiling code. As a consequence, warmup time and memory footprint are impacted. We observe that FaaS workloads, which are short-lived, do not fit this profile.

We propose CloudJIT, a self-optimizing FaaS platform that takes advantage of Ahead-of-Time compilation to achieve reduced startup latency and instantaneous peak performance with a smaller memory footprint. While AOT compilation is an expensive operation, the platform automatically detects which functions will benefit from it the most, performs all prerequisite preparation procedures, and compiles selected functions into native binaries. Our preliminary analysis, based on a public FaaS invocations trace, shows that optimizing a small fraction of all functions positively affects a vast majority of all cold starts.

CCS Concepts: • **Software and its engineering → Runtime environments**; • **Computer systems organization → Cloud computing**.

Keywords: Function-as-a-Service, Ahead-of-Time Compilation, Just-In-Time Compilation, GraalVM Native Image

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622990

ACM Reference Format:
Serhii Ivanenko, Rodrigo Bruno, Jovan Stevanovic, Luís Veiga, and Vojin Jovanovic. 2023. CloudJIT: A Just-in-Time FaaS Optimizer (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3617651.3622990

1 Introduction

Function-as-a-Service has become an attractive alternative to run lightweight and short-lived computational workloads in the cloud. Such workloads can accommodate a high volume of parallel tasks where each individual function can scale independently, thus making cloud applications highly scalable and elastic.

In FaaS, cloud service providers manage infrastructure provisioning for executing functions, alleviating customers from resource management burdens and making function deployment as simple as uploading a snippet of code to the platform. On the other hand, it also gives cloud providers a substantial amount of control over computational infrastructure and function execution environment, thus enabling providers to optimize resource allocation and usage.

Functions are primarily written in high-level languages such as Python, JavaScript, and Java [1] which execute atop a language runtime. These runtimes, however, lead to high startup and initialization times that can take up a significant portion of the overall function execution time. In addition to the long initialization overhead, language runtimes that use a Just-in-Time (JIT) compiler, Java and JavaScript, are also subject to a warmup overhead. Finally, such runtimes also require a considerable memory footprint to be in memory.

JIT compilation is particularly effective for long-running applications, which give the JIT compiler the opportunity to identify snippets of code that are frequently executed and how they are executed. Before obtaining highly-optimized application code, the language runtime goes through several tiers of profiling data collection and compilation.

We claim that while long-running applications can benefit from JIT compilation, this is not the case for FaaS applications, where most functions are short-running. A recent study showed that more than half of all functions execute under one second [18]. A different study showed that the majority of serverless functions pay the price of long cold start times and warmup times but rarely benefit from highly optimized JITted code [13].

Recent works tackle slow cold starts and warmup overheads in different ways. Some works propose serverless systems backed by lightweight virtualization [12, 15, 16, 21]. These systems usually rely on Software Fault Isolation (SFI) to collocate several function executions in a single worker node. Such systems experience fewer cold starts since, instead of allocating a separate serverless worker with an entire virtualization stack, they allocate an isolated execution environment in an already existing worker node. However, such execution environments may still carry a burden of initial optimization duties that they will never take advantage of. Other studies propose forking function runtime processes instead of launching them from the ground up [11, 17]. However, many managed runtimes do not support forking. Other studies suggest restoring serverless workers from snapshots to avoid initializing virtualization stack [14, 22], but this approach induces additional storage and bandwidth overhead to manage snapshots. More importantly, both forking and snapshotting techniques still suffer from running an unoptimized function code in the interpreter mode. Deciding when to snapshot is also an open problem.

An alternative approach to reduce both cold start latency and warmup overheads is to replace Just-In-Time compilation with Ahead-of-Time (AOT) compilation. By doing so, the entire program code is compiled into a native binary beforehand that includes a minimal runtime necessary to run the application. Henceforward, the AOT-compiled application leads to a lower memory footprint, has lower startup latency, and has minimal warmup latency (as the code is not interpreted nor profiled).

There are some setbacks when using AOT compilation, however. First, natively built applications cannot benefit from speculative runtime optimizations since the original version of the code is not kept as part of the binary, and therefore de-optimizing the code is not an option. Second, AOT compilation is time-consuming (or costly in a cloud environment) as it requires multiple computationally intensive compilation phases. Third, AOT compilation operates under a closed-world assumption that limits/forbids dynamic features, such as reflection or serialization, which are commonly used in managed languages. Since AOT compilation eliminates the need for the managed runtime to execute a native binary, such dynamic features may be supported only partially or require configuration to be supported. Although it may be possible to have some of these features at runtime while using AOT compilation, explicit configuration is required. Creating this configuration, however, is non-trivial as it is error-prone and may be incomplete, leading to runtime errors. Finally, AOT compilation often results in larger binaries compared to the original application code.

In view of the above, we introduce CloudJIT, a Just-In-Time FaaS optimizer that automatically selects warm functions as candidates for AOT compilation. As the name implies, CloudJIT is inspired by how JIT compilers work, i.e., FaaS functions (methods in traditional JIT compilers) are monitored to detect warm functions, which are considered for AOT compilation. Since AOT compilation is expensive, only a restricted selection of functions should be picked up for compilation. Following the ideal of FaaS platforms (and JIT compilers), all these optimizations are transparent to the users, i.e., they do not require additional effort from the function developer's side. The core component of CloudJIT is an optimizing pipeline that the function goes through. This pipeline consists of several steps that produce artifacts that are later consumed in subsequent stages of the pipeline.

Our project is based on GraalVM [4] Native Image, an AOT compilation tool that converts bytecode into native code. Applications compiled with GraalVM Native Image do not require a full-blown JVM for execution and run native code with minimal startup and warmup overheads. CloudJIT takes advantage of the Native Image Agent [2], a tool that tracks dynamic feature access, to automatically create Native Image configuration files transparently to the user.

In our preliminary analysis (§5.1), we demonstrate the potential of this work by running a simulator using real FaaS traces [18]. We also validate these results with a prototype of our project in §5.2. Results show that CloudJIT optimizes a small number of functions that are responsible for the vast majority of invocations and, as a consequence, significantly reduces the overall memory footprint and optimizes a large portion of cold starts.

In summary, this work offers the following contributions:

- it proposes using AOT compilation as a replacement for JIT compilation of FaaS functions;
- it proposes a platform that automatically detects and builds functions that could greatly benefit from AOT compilation (i.e., acting as a cloud JIT optimizer);
- it demonstrates the feasibility of CloudJIT using a number of experiments and simulations using real-world traces from a FaaS platform;
- it evaluates the impact of CloudJIT on a realistic serverless deployment.

2 FaaS and Language Runtimes

FaaS functions are traditionally developed using high-level languages that run atop a language runtime that is instantiated and managed by the FaaS platform. These runtimes, however, occupy a significant amount of memory, in the

range of tens of MBs, and take tens of milliseconds to load and initialize [12]. To avoid paying the cost of loading and initializing runtimes on every request (i.e., a cold start), platforms re-use runtimes that were used to handle previous invocations of the same function and that are currently not being used. Doing so reduces the cold start overhead but introduces a considerable memory footprint overhead to maintain all these runtimes idle, waiting for future invocations. Fundamentally, FaaS platforms need to balance cold start overheads with memory footprint overheads. Language runtimes play a crucial role in that overhead as one of the most complex and expensive components (in terms of memory footprint and initialization time) necessary to prepare a FaaS execution environment.

The majority of FaaS functions run atop language runtimes [1]. Managed language runtimes offer multiple benefits that result from decades of research in areas such as Garbage Collection (GC), a sandboxed execution environment, and dynamic and speculative code optimizations, among others. However, while these features are important for long-running server applications, they are less important for short-running and lightweight functions. We argue that most of these features come at a high cost in terms of long runtime initialization and high memory footprint. We claim that traditional runtimes are not effective in FaaS environments.

Java, as one of the most popular languages in FaaS [13, 15] is a good example. It uses a tiered compilation approach [24] to obtain better-optimized code snippets. The JVM makes use of the interpreter, the C1 (client) JIT compiler, and the C2 (server) JIT compiler to achieve maximum long-term performance. The JIT compiler makes assumptions about program behavior and optimizes the code accordingly. In case the actual behavior diverges from the previously collected profiling data, the code gets deoptimized, and the runtime falls back to the interpreter tier to repeat the entire pipeline. Despite all this language runtime complexity, FaaS functions do not run for long enough to benefit from it [18]. For such functions, cold start takes up a considerable fraction of total execution time [23]. Additionally, all functions go through the slow warmup phase to collect profiles for future optimizations.

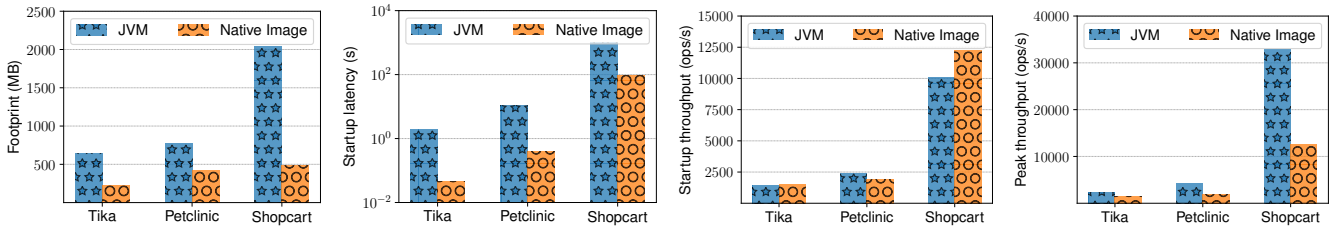
Recent studies approach the problem of FaaS long cold starts and high memory footprint in different ways. One widely adopted approach is to try to avoid cold starts by re-using warm function workers from the recently executed invocations. The duration of this time window differs from platform to platform and can last at least 10-20 minutes [19, 20]. This way, whenever a new invocation comes into the function within that period, the warm worker can be picked up to serve it, hence avoiding instantiating another worker. However, such an approach imposes memory overhead since function workers should be kept in the main memory for the keep-alive window. Besides, each parallel invocation triggers the creation of another worker that experiences a cold start.

Another way to reduce cold start time is to employ lightweight virtualization stacks to host separate invocations of the same function. Typically, the main idea behind this approach is to employ Software Fault Isolation (SFI) within language runtime processes. This enables platforms to collocate parallel executions of the same function code in a single runtime process. With such an approach, given that a function worker is already instantiated, the system creates a slim isolated execution environment inside the worker instead of starting up an entirely separate worker. However, SFI needs to be supported by an underlying language runtime. Also, it does not solve the problem of slow cold starts and warmups for functions that are invoked somewhat infrequently.

A promising alternative suggests using snapshotting to reduce the cold start time of FaaS functions. With this approach, a cloud service provider needs to take a snapshot of the entire function worker after its first deployment. This snapshot is stored persistently in external storage. Instead of instantiating new workers from the ground up, the FaaS infrastructure can restore new workers from the snapshot that was created for a particular function. This approach speeds up cold starts [14] but imposes additional overhead on storage since snapshots occupy more disk space than user-provided code. It may also affect network bandwidth as entire snapshots may need to be uploaded from the remote storage. Besides, snapshotting may introduce security vulnerabilities as all workers, restored from the snapshot, will have the same address space layout (which is randomized for security purposes). Finally, deciding when to take a snapshot is a fundamentally hard problem as has been noted recently [13]. All the aforementioned techniques attempt to solve the problem of long cold start latency. However, none of these solutions target slow execution right after the cold start due to a warmup in managed language runtimes.

3 Ups and Downs of AOT Compilation

Ahead-of-Time (AOT) compilation offers a way to reduce the cost of language runtimes. It not only reduces startup and warmup latency by avoiding loading, parsing, interpreting, and profiling the function code but also memory footprint by reducing the number of components that are not part of the runtime (the interpreter and JIT compiler may be removed from the runtime if only AOT compiled code is used). To show this, we conducted an experiment by running three Java functions based on popular frameworks: Tika (a PDF converter based on Quarkus [8]), Petclinic (a web application based on Spring Boot [9]), and Shopcart (a web application based on Micronaut [6]). The first function is a stateless application with a single functionality, whereas the latter two are complete web services operating with local in-memory storage to keep state. Figure 1 shows the startup, throughput, and memory footprint of different Java applications when running on GraalVM [4], a full-blown runtime (based on



a) Memory footprint b) Startup latency c) Startup throughput d) Peak throughput

Figure 1. Performance comparison of three microservice applications atop GraalVM (which includes a JIT compiler) and as Native Image AOT-compiled binary. Note the log-scale in b) Startup latency.

Table 1. AOT compilation maximum RSS footprint, total build time, and final binary size.

Benchmark	RSS (GB)	Time (sec)	Size (MB)
Tika (Quarkus)	6.41	97	89.7
Petclinic (Spring Boot)	8.21	200	189.3
Shopcart (Micronaut)	6.31	87	69.4

HotSpot [5]) including an interpreter and a JIT compiler, and GraalVM Native Image, an AOT compiled binary that bundles the application together with a lightweight Java runtime (SubstrateVM).

Results show that AOT compilation greatly reduces the memory footprint and startup latency, and improves startup throughput. The reason is that the AOT compilation process compiles the whole program into native code before it becomes available for execution. This native code is packaged in a self-contained executable binary that does not require the full language runtime for execution. Since the code is already compiled into native, no additional actions (such as tiered JIT compilation) need to be taken in order to execute the program. This means that AOT compilation eliminates the warmup phase along with its overheads, and the code executes at peak performance immediately after startup [25]. Native binaries utilize only necessary pieces of language runtimes, thus not carrying the burden of initialing unused resources. Finally, due to the same reason, natively built applications leave a smaller memory footprint. Unfortunately, AOT compilation comes at the expense of reduced peak throughput (in our experiment, peak throughput is measured after 15 seconds of execution). Even though many functions will not reach peak throughput in a realistic scenario, we plan on using Profile-Guided Optimizations to reduce the performance gap (§6).

AOT compilation offers faster startup, minimal warmup, and reduced memory. However, such an approach does not eliminate compilation overhead; it merely moves all code analysis and compilation routines to the phase preceding the actual application execution, nonetheless, avoiding such overhead at run time.

Native binary generation is a complex task that conventionally consists of several consecutive phases, including but not limited to code analysis, method inlining, code compilation, and heap pre-initialization. Our experiments show that the AOT compilation process for typical Java applications takes a significant amount of time and imposes high memory requirements on the system (see Table 1), and in general, is a resource-consuming job. Real-world FaaS clusters operate with a large number of functions, and building native binaries for all of them would require immense resources.

Besides, based on the collected profiles, JIT compilers perform speculative optimizations such as type inference, branch predictions, and aggressive method inlining. Traditional AOT compilers, on the other hand, cannot speculatively optimize the function code. Therefore, they generate native code by analyzing the source code only statically, and the resulting native binary will only feature conservative optimizations. That being said, native code generated by an AOT compiler may be less efficient than the JIT compiler code as it lacks dynamic runtime data that can drive more complete optimizations. Figure 1.d shows that JIT compilation achieves better peak throughput in the long run.

Finally, GraalVM Native Image, the tool we use to AOT compile functions, operates under a closed-world assumption which limits dynamic code loading and other reflective and dynamic features of the Java language. As a consequence, only the elements known at compile time will be included in the final binary. These components must be either reachable for static analysis or specified explicitly in additional configuration files. We have conducted an experiment to measure the complexity of the configuration that needs to be collected to build Java applications that rely on popular frameworks with the Native Image (see Figure 2). Based on the results of this experiment, we can conclude that it may be a non-trivial task to manually prepare this configuration for applications that make extensive use of the dynamic features.

4 CloudJIT

We propose CloudJIT, a serverless infrastructure that takes advantage of AOT compilation to improve function cold start, reduce memory footprint, and reduce warmup latency.

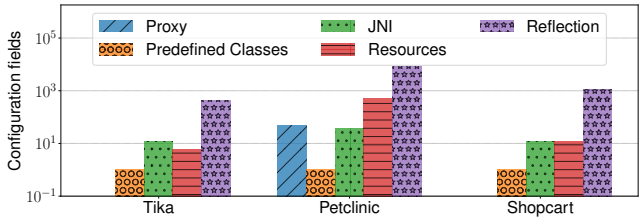


Figure 2. Number of parameters needed to fully configure microservices for AOT compilation.

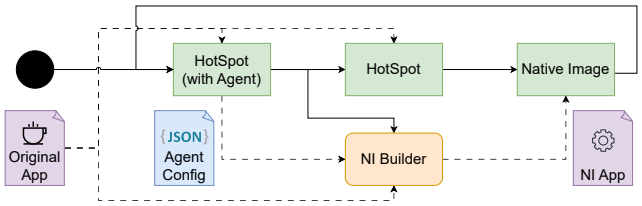


Figure 3. CloudJIT optimization pipeline. Solid lines represent state transitions, and dashed lines represent artifacts.

In CloudJIT, certain functions go through the optimizing pipeline, which consists of several consecutive stages. The system detects functions that will benefit from such optimizations the most and gradually applies the pipeline to these functions. The entire process of function optimization occurs transparently to the user, not requiring any intervention or source code modifications.

4.1 Function Optimization Pipeline

The optimizing pipeline is a central component of our system. The main objective is to prepare the function code for AOT compilation, proceed with actual compilation, and further improve the natively compiled code in a transparent way. This pipeline is composed of 3 sequential stages and 1 background process. Some stages and processes produce artifacts that are used as inputs to one or multiple successive stages or processes. Fundamentally, this pipeline implements a simple state machine for each function in the FaaS infrastructure. Figure 3 shows the transitions between the stages of the pipeline and how these stages produce or consume artifacts. We now briefly describe each stage and its transitions.

HotSpot (with Agent) is the first stage of the pipeline for every function. Upon the first invocation of a function, a HotSpot is launched and provided with the function code (a JAR file). HotSpot is launched with the Native Image Agent tool which automatically tracks dynamic features used at run time. The goal of this stage is to extract dynamic feature configurations for the function. The result of this stage is a set of JSON files, each file representing configuration for a particular dynamic feature.

Executing requests with a Native Image Agent enabled introduces performance degradation as a result of intercepting some events within the JVM. For this reason, in order not to deteriorate the performance of most invocations, we only deploy this stage once for each function, and it is only used to serve up to 1000 requests (a configurable threshold).

HotSpot function instances are launched if the following conditions are met: a HotSpot with the Agent is already running to serve concurrent requests or if the Agent configuration has already been produced; and if there is no Native Image built for this function.

During the HotSpot stage, the function is executed on top of a traditional JVM. This stage does not produce any artifacts. If some function never gets marked as worth optimizing, then this function will never proceed further through the pipeline. Otherwise, this stage is going to support function invocations when the function binary is being compiled.

NI Builder is a background process that is invoked whenever CloudJIT takes the decision to proceed with the function optimization. The primary objective of this process is to generate an AOT-compiled binary of the function. It uses the GraalVM Native Image compiler to accomplish the compilation process. Upon invocation, it accepts function bytecode and configuration as input and produces a native binary that enables the function to transition to the *Native Image* stage.

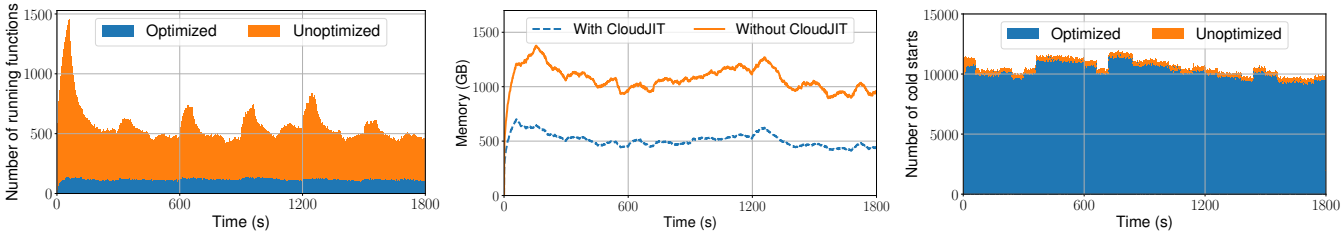
Native Image is the final stage of the pipeline, where the function is considered to be fully optimized. Throughout this stage, the function runs as a native binary, thus benefiting from reduced cold start latency, minimal warmup overhead, and reduced memory footprint.

If an AOT-compiled binary attempts to invoke some dynamic feature that it was not configured for, it may fail with an exception, and the corresponding function invocation will not succeed. In such cases, CloudJIT deoptimizes the function by rolling it back to the first stage of the pipeline. The invocation is then re-executed on a HotSpot with Native Image Agent to fulfill the request and update the function configuration. Then, the entire optimizing pipeline is repeated again. To avoid potential correctness issues, CloudJIT allows users to define what happens when a request fails. Users may decide to re-execute the request (on a HotSpot with the Agent instance) or simply return an error. In the latter, CloudJIT still deoptimizes the function but does not re-execute the failed request.

4.2 Which Functions to Optimize?

Manually AOT compiling applications is a slow and resource-consuming process (§3). Real production platforms manage numerous functions, and compiling all registered functions may be an overkill that does not always pay off. Therefore, not all functions are worth AOT compilation.

In §3, we discussed that AOT compilation predominantly improves cold start latency and instantaneous throughput. Based on this observation, we argue that functions that experience cold starts most frequently will benefit from the optimization pipeline the most. The frequency of cold starts for



a) Running functions b) Memory footprint c) Cold starts

Figure 4. *Simulating* the impact of AOT compilation using the Azure Functions invocation trace.

each function depends on multiple factors, such as function invocation pattern and duration of the keep-alive window (i.e., for how long a FaaS platform keeps function workers in memory after serving an invocation).

Our system defines a criterion to determine cold start regularity for each function. It counts how many cold starts happened throughout the time of the sliding window. If the amount of cold starts within this window exceeds a threshold, then the function is marked as worth optimizing, and the corresponding **NI Builder** process is launched. Optimization policies can be configured by adjusting the threshold and duration of the sliding window. Our default values are 10 cold starts in a sliding window of 1 minute.

5 Evaluation

We now assess how AOT compilation affects memory footprint and the number of cold starts in a realistic FaaS environment. To do so, we built an infrastructure capable of simulating invocation traces from the Azure Functions public dataset [18] at a cluster-wide level (§5.1). It considers each invocation from the trace and aggregates metrics for periods of 1 second. We implemented a CloudJIT prototype that we use to replay the Azure Functions trace at a level of a single serverless node with actual function invocations (§5.2).

5.1 Real-World Trace Simulation

In our simulation setup, we consider a 30-minute time segment from the Azure Functions dataset. The keep-alive time window is fixed for each function and is 10 minutes, the default value in OpenWhisk [7]. During the simulation, we measure the number of running functions, memory footprint, and the number of cold starts. We distinguish between optimized and unoptimized functions when measuring the number of running functions and cold starts, and we calculate the memory footprint for simulation with and without optimizations enabled. Figure 4 depicts results of our simulation execution.

Figure 4.a shows how many functions are running invocations at each moment of time. We can observe that a greater part of all functions do not get optimized, and only a small fraction of the functions are considered eligible for AOT compilation. The simulator defines the following criterion for optimization: if the function experiences more than ten cold starts over a 1-minute period, then it is deemed worth AOT compilation. The main takeaway from this plot is that in a real-world FaaS scenario, only a minor part of functions will ever get to be optimized with our pipeline.

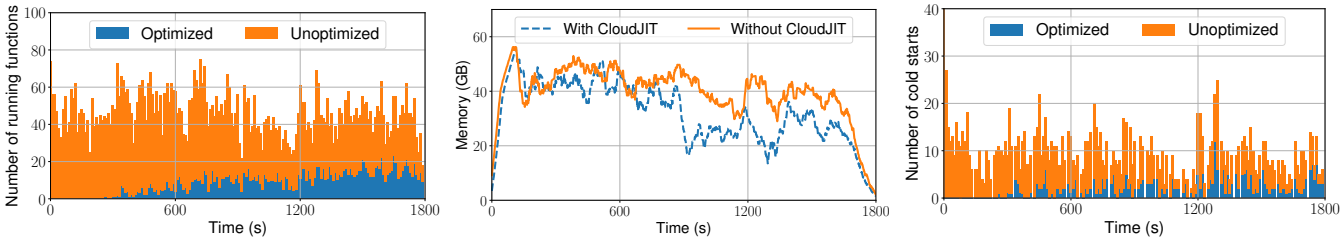
The following plot (see Figure 4.b) depicts the difference between cluster-wide memory utilization of the running functions with and without AOT optimization enabled. From the plot, we see that by applying AOT compilation to certain functions, the total memory footprint gets reduced by 52.6%. In order to calculate the memory footprint of the optimized function, the simulator multiplies its actual memory footprint value by 0.375, a ratio derived from the experiment depicted in Figure 1.a. Finally, Figure 4.c shows how many cold starts happen during each 1-second period throughout the trace simulation. We can see that optimized functions generate the vast majority of all cold starts in the system.

Takeaway. From the simulation results, only a small fraction of all functions in the cluster impose most of the cold starts and take up a considerable portion of the overall memory footprint. Therefore, a modest overhead of AOT compilation for these few functions can significantly reduce latency for the vast majority of cold starts and decrease memory requirements for most of the function workers.

5.2 Real-World Trace Execution

In the execution setup, we re-use the 30-minute time segment from the Azure Functions dataset, and we downscale the dataset to accommodate its workload in the local cluster node running Ubuntu 18.04.6 LTS (Linux kernel 5.8.5-050805) with an Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz and 128GB of DDR4 DRAM. The trace does not describe the functions being executed, and, therefore, we use a generic Java function that allocates memory according to the trace data and performs hashing operations to simulate the workload.

In order to execute the trace, we implemented our CloudJIT prototype on a single node that distributes invocations to local function workers. CloudJIT handles function registration, invocation scheduling, and manages workers. It also carries out the optimization pipeline functionality, such as



a) Running functions b) Memory footprint c) Cold starts

Figure 5. *Evaluating* the impact of AOT compilation using the Azure Functions invocation trace.

determining function eligibility for further optimization and initiating compilation tasks for eligible functions. Function workers are based on Firecracker microVMs [10].

We execute a down-scaled version of the original trace on a cluster node. In this setup, functions may lose their original invocation patterns and experience fewer cold starts. Therefore, during the execution experiment, CloudJIT marks functions for optimization if they receive more than two cold starts over 10 minutes. If we apply the same cold start configuration to execution as for simulation, the system may end up optimizing only a few functions (as it only sees a fraction of the original invocations). We plan to lift this limitation in the future by allowing CloudJIT to manage a number of nodes instead of a single one.

During trace execution, we collect the same metrics as during simulation: number of running functions, memory footprint, and number of cold starts. The distinction between optimized and unoptimized functions is preserved the same as in simulation. We also conducted two executions — with and without AOT optimization enabled to compare memory footprints. Figure 5 shows the results of the trace execution.

Figure 5.a demonstrates a similar tendency depicted in Figure 4.a. We observe that the first compiled functions start appearing in the trace after 5 minutes of execution. After 10 minutes of execution (the time needed to select and compile most of the eligible functions), an average of 27% of all running functions were optimized. Other functions were considered unworthy of AOT compilation.

Figure 5.b shows that after optimizing certain functions, overall memory consumption can be reduced by 30% on average. The percentage of reduced memory utilization is lower than in the simulation because the simulation metrics did not account for the memory requirements of the virtualization stack. The memory overhead of VMs pertains to all function workers regardless of whether or not the corresponding functions are optimized.

Finally, Figure 5.c illustrates that, after certain functions are AOT-compiled, an average of 30% of all cold starts relate to the optimized functions. As we downscale the trace for this experiment, the number of invocations is decreased for all functions evenly. Thus, some functions may lose their original invocation pattern with multiple cold starts over a short period of time. Therefore, such functions are deemed unworthy of optimization in our experimental setup.

6 Conclusions and Ongoing Work

We propose CloudJIT, a FaaS platform that automatically (and transparently) optimizes user functions based on the number of cold starts a function had recently. CloudJIT picks a small fraction of functions for optimization (therefore incurring a minor AOT compilation overhead) which still leads to a large portion of optimized cold starts and reduced memory footprint. CloudJIT is currently a work-in-progress research effort which we plan on extending to accommodate at least two new optimization stages: PGO and Checkpointed.

Profile-Guided-Optimization (PGO) execution, requires CloudJIT to build an instrumented version of the function binary to collect profiles, which later are used to build an optimized version of the binary. We plan on studying the performance improvements of PGO and determine how to filter functions that should advance to this stage of the pipeline;

Checkpointed execution is made possible by snapshotting a running function worker instance to disk for later reuse. Compared to previous works that propose using snapshotting at the hypervisor level (e.g., snapshot the entire VM), we plan on investigating the option of snapshotting at the sandbox level. This feature might be particularly effective for dynamic languages such as Python and JavaScript where AOT compilation is not possible [3].

Acknowledgments

We thank our shepherd, Elisa Gonzalez Boix, and the anonymous reviewers for the insightful feedback and help to improve the paper. This research project is supported by a grant from Oracle Labs and by national funds through Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 and by the CloudStars project, funded from the European Union's Horizon research and innovation program under grant agreement number 101086248.

References

[1] 2020. For the Love of Serverless: Lambda Adoption by Runtime. https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020. Accessed: 2023-06-29.

[2] 2022. Assisted Configuration with Tracing Agent. https://www.graalvm.org/22.0/reference-manual/native-image/Agent/. Accessed: 2023-06-29.

[3] 2023. Auxiliary Engine Caching. https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AuxiliaryEngineCachingEnterprise/. Accessed: 2023-06-29.

[4] 2023. GraalVM. https://www.graalvm.org/. Accessed: 2023-06-29.

[5] 2023. The HotSpot Group. https://openjdk.org/groups/hotspot/. Accessed: 2023-06-29.

[6] 2023. Micronaut Framework. https://micronaut.io/. Accessed: 2023-06-29.

[7] 2023. OpenWhisk - Open Source Serverless Cloud Platform. https://openwhisk.apache.org/. Accessed: 2023-06-29.

[8] 2023. Quarkus. https://quarkus.io/. Accessed: 2023-06-29.

[9] 2023. Spring Boot. https://spring.io/projects/spring-boot/. Accessed: 2023-06-29.

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[11] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.

[12] Rodrigo Bruno, Serhii Ivanenko, Sutao Wang, Jovan Stevanovic, and Vojin Jovanovic. 2022. Graalvisor: Virtualized Polyglot Runtime for Serverless Applications. *arXiv:2212.10131v1 [cs.DC]* (2022).

[13] 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*. 58–64.

[14] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[15] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 45–59.

[16] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[17] 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)*. 57–70.

[18] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[19] Mikhail Shilkov. 2021. Cold Starts in AWS Lambda. https://mikhail.io/serverless/coldstarts/aws/. Accessed: 2023-06-29.

[20] Mikhail Shilkov. 2021. Cold Starts in Azure Functions. https://mikhail.io/serverless/coldstarts/azure/. Accessed: 2023-06-29.

[21] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[22] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[24] Roland Westrelin. 2021. How the JIT compiler boosts Java performance in OpenJDK. https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk/. Accessed: 2023-06-29.

[25] 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. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (oct 2019), 29 pages. https://doi.org/10.1145/3360610