# Remote Just-in-Time Compilation for Dynamic Languages

Andrej Pečimúth
Charles University
Czech Republic
pecimuth@d3s.mff.cuni.cz
Oracle Labs
Czech Republic
andrej.pecimuth@oracle.com

## ABSTRACT

Cloud platforms allow applications to meet fluctuating levels of demand while maximizing hardware occupancy at the same time. These deployment models are characterized by short-lived applications running in resource-constrained environments. This poses a challenge for dynamic languages with just-in-time (JIT) compilation. Dynamic-language runtimes suffer from a warmup phase and resource-usage peaks caused by JIT compilation. Offloading compilation jobs to a dedicate server is a possible mitigation for these problems. We propose leveraging remote JIT compilation as means to enable coordination between the independent instances. By sharing compilation results, aggregating profiles, and adapting the compiler and compilation policy, we strive to improve peak performance and further reduce warmup times. Additionally, an implementation on top of the Truffle framework enables us to bring these benefits to many popular languages.

## CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Dynamic compilers.*

## KEYWORDS

remote compilation, just-in-time compilation, dynamic languages, virtual machines

**ACM Reference Format:**
Andrej Pečimúth. 2018. Remote Just-in-Time Compilation for Dynamic Languages. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 MOTIVATION

Modern applications, such as web applications, need to be able to accommodate varying levels of demand. Cloud platforms enable this elasticity [4] through horizontal scaling, which can be effectively implemented using container orchestrators. In this setup, the application is deployed across multiple independent replicas, each running in a separate container distributed across multiple nodes. Incoming requests are routed to these instances by the runtime. When there is an increase in demand, the orchestrator automatically starts additional instances to handle the load. Conversely, when demand decreases, the orchestrator terminates instances to optimize cost efficiency.

These applications are commonly implemented using dynamic languages that rely on just-in-time (JIT) compilation to achieve high performance. However, JIT compilation incurs runtime costs in terms of CPU and memory usage. These environments experience an often significant warmup phase [2], when the application's performance is below peak. A virtual machine (VM) may reduce the time it takes to warm up (i.e., warmup time) by utilizing and interpreter and multiple JIT compilers in a tiered setup [10]. The individual compilers offer varying optimization levels and compilation latencies.

To guide optimization decisions [13], VMs collect profiles of the running application. The quality of the compiled code depends on the quality of profiles. Profiles are often collected by compiling and running instrumented code [12], which incurs major runtime overhead. Thus, they are captured during a limited time window.

Cloud environments with automatic scaling exacerbate the disadvantages of JIT compilation. Whenever the orchestrator terminates a container, compiled code and profiling information is lost. Starting a new container involves compiling from scratch, and thus going through another warmup phase.

Moreover, JIT compilers cause spikes in CPU and memory utilization early in a container's lifetime. Overloading the CPU may cause a failure to meet latency expecations. These languase are usually garbage collected, which performs poorly in low-memory conditions. Thus, it is necessary to provision containers with sufficient resources. However, compilation jobs quickly become less frequent later in an application's lifetime. As a result, the resources provisioned for JIT compilation are not utilized. The underutilization of provisioned resources leads to poor cost efficiency.

Possible mitigations include ahead-of-time (AOT) compilation [11], caching [3] or sharing [17] compiled code, sharing profiles [12], and remote compilation [9]. Native Image [11] solves the problem of warmup and JIT-related resource usage by compiling Java applications AOT. However, there are restrictions related to dynamic features, and peak performance may suffer as runtime recompilation is unavailable. Another approach taken by some VMs is persisting compiled code [3] to the disk. The persisted code is used as a cache in subsequent VM invocations. The Hip Hop VM (HHVM) for the Hack language employs profile sharing [12] to reduce warmup and improve peak performance. Finally, OpenJ9 offers JITServer [9], which allows offloading compilation jobs to


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

a remote compilation server. By caching compiled code, JITServer achieves lower warmup time and an overall reduction in resource usage.

## 2 PROBLEM

Contemporary language runtimes are not well suited for elastic [4] cloud environments. Although the techniques we described tackle some of the issues, neither takes full advantage of the environment. JITServer [9], the state-of-the-art remote compilation server for Java, does not attempt to increase peak throughput by adapting the compiler nor profile aggregation. HHVM [12] runs in a single-VM setup and shares only profiles. We propose a multi-language remote compilation server as a means to improve warmup and peak performance. We can achieve this by sharing compiled code and profiles, pushing code to clients, an improved compilation policy, and an implementation on top of Truffle [16].

Tuning a JIT compiler for the remote-compilation scenario may yield peak performance improvements. Compiler design is a trade-off between compilation speed and the performance of compiled code. JIT compilers are further restriced to conserve resources, because they run along user code. However, this is not true for the compilation server: the server may spend as much resources as it has available.

The VMs may obtain more precise profiles while utilizing less resources overall by coordinating their efforts through the compilation server. As profiles are merely samples of the application's characteristics, we can obtain higher-quality estimates by aggregating the individually collected information. As soon there are enough samples, there should be no further profile collection to conserve resources. A natural place to coordinate these efforts is the compilation server, which could immediately use the profiles for compilation.

Another opportunity to reduce warmup is pushing compiled code to clients. As all clients run the same application, the server may determine which methods are needed by the clients. However, this in conflict with profile aggregation: when the server pushes code to a client, the client will not collect profiles. Thus, more elaborate compilation policies are required.

We propose an implementation of remote compilation for Truffle [16] to bring all the benefits to multiple popular languages. Truffle [16] is a language implementation framework built on top of GraalVM [7]. Implementations of popular dynamic languages using Truffle are available, some of which are as fast or faster [6] than the reference implementations. Truffle enables language implementers to construct a high-performance VM by writing an abstract syntax tree (AST) interpreter using the framework. This is made possible by creating a JIT compiler from the AST interpeter using partial evaluation.

## 3 APPROACH

Figure 1 demonstrates our approach. We propose a dedicated compilation server that serves multiple containerized VMs. The VMs run the same application, and the orchestrator may start or terminate containers. Client VMs utilize an interpreter and a level-one compiler optimized for compilation latency, such as the client compiler [10] from the HotSpot Java VM. The level-one compiler can

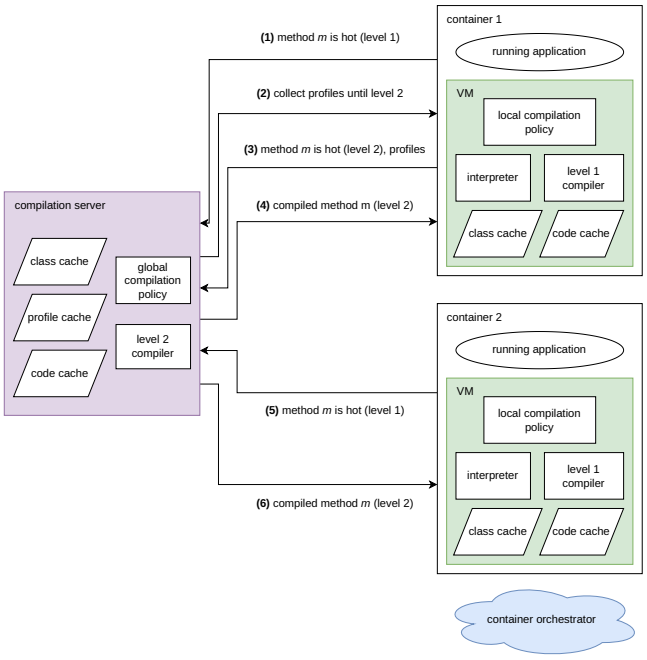


**Figure 1: Our approach: a remote compilation with code sharing and profile sharing.**

also compile instrumented code to collect profiles. The compilation server utilizes a level-two compiler striving for peak performance, i.e., the GraalVM Compiler [5]. The server facilitates code and profile sharing.

In the scenario shown in Figure 1, container 1 executes a non-trivial method *m*. The execution starts in the interpreter. After the method exceed a predefined number of invocations (i.e., it is *hot*), the local compilation policy informs the compilation server. The server's global compilation policy can now either let the VM collect profiles, schedule the method for level-two compilation, or return cached code. In this case, the former is true. When the method exceeds another invocation threshold, the VM sends the collected profiles to the server. The server compiles the method, caches the profiles and code, and returns code to the VM. When method *m* becomes hot in container 2, the global compilation policy pushes the cached code optimized for immediate peak performance.

We split the implementation into five stages. In the first stage, we implement a remote compilation sever based on GraalVM. The GraalVM Compiler [5] utilizes the Java Virtual Machine Compiler Interface (JVMCI) to communicate with the VM. The VM requests compilations using this interface, and the interface allows the compiler to access the required VM structures, such as information about classes, methods, fields, etc. Thus, remote compilation is feasible by forwarding JVMCI calls to the server. The technical challenge is reducing the number of JVMCI calls, as they are associated with network overheads.

In the second stage, we aim to increase peak performance through profile aggregation and compiler adaptions. As the clients collect

profiles and share them with the server, the accuracy of the profiles increases. Thus, compilation outcomes should improve over time.

To adapt the JIT compiler for remote compilation, we can tune the compilation budget or prepare a customized phase plan for the optimizer. A possible risk of such a solution is an increased compilation latency and a consequent regression in warmup time. To counter this, we can employ two compiler configurations in the compilation server: the first to decrease compilation latency, the second to outperform local JIT compilers.

In the third stage, we enable caching and sharing compilation results among multiple clients. The aim is to reduce overall resource usage and the warmup time [9] of newly started VMs.

Compilation caching interplays with profile aggregation. After the server compiles and caches a method for future use, it may still receive additional samples to refine the profiles. Thus, whenever the server detects a significant shift in the profiles, it discards older compilation units and recompiles methods taking advantage of the refined profiles.

To allow code sharing, the clients must have the same set of loaded Java classes, with identical bytecodes for each method. We must also track speculative assumptions in the compiled code and the assumptions violated by each client. As another technical difficulty, the native code may contain pointers to internal data structures, which differ between VM invocations. Thus, the code requires relocation.

In the fourth stage, the server is allowed to prematurely push optimized code to a client, as illustrated for container 2 in Figure 1. The aim is to further reduce warmup times and overall resource utilization. In this scenario, the client skips level-one compilation and immediately installs level-two code. As a consequence, the server will not receive any new profiles. The global compilation policy may choose this action when it is confident that client will need the compiled code and the profiles are already accurate.

Finally, in the fifth stage, we leverage the remote compilation server for Truffle [16] workloads. A Truffle client can directly send the AST of a method it wishes to compile. The server runs the partial evaluator and returns compiled code. As there are many Truffle implementations of popular languages, multiple runtimes may benefit from this.

## 4 EVALUATION METHODOLOGY

The goal of this research is to answer whether we can utilize remote compilation to further improve performance metrics of applications running in horizontally scaled setups. In particular, we aim to answer the questions below for Java and Truffle workloads.

- Does the remote compilation server with profile aggregation (from stage two) increase peak performance without regressing in warmup time?
- Does the remote compilation server with caching and profile aggregation (from stage three) improve peak performance and warmup time?
- Does the remote compilation server with caching, profile aggregation, and code pushing (from stage four) further decrease warmup time with improved peak performance?

To evaluate peak performance and warmup time, we run a benchmark on a fixed number of containers, each with predefined resource limits. We compare a locally compiled configuration with a configuration using the remote server. The remote compilation server runs in an independent container. For Java, there are web-based workloads such as AcmeAir [1], DayTrader [8], and PetClinic [15]. These applications require that we generate the requests for them. We can also measure less specialized workloads from the Renaissance [14] benchmark suite. After the last implementation stage, we can measure workloads available for other languages on Truffe [16].

## REFERENCES


[1] [SW], Acme Air Sample and Benchmark 2023. URL: https://github.com/blueperf/acmeair-monolithic-java.

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.*, 1, OOPSLA, Article 52, (Oct. 2017), 27 pages. DOI: 10.1145/3133876.

[3] Dev Bhattacharya, Kenneth B. Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic. 2017. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 1–6. DOI: 10.1109/PACRIM.2017.8121911.

[4] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. 2014. Elasticity in cloud computing: a survey. *Annals of Telecommunications*, 70, 7-8, (Nov. 2014), 289–309. DOI: 10.1007/s12243-014-0450-7.

[5] [SW], Graal Compiler 2023. URL: https://www.graalvm.org/latest/reference-manual/java/compiler/.

[6] [SW], GraalPy 2023. URL: https://github.com/oracle/graalpython.

[7] [SW], GraalVM 2023. URL: https://www.graalvm.org/.

[8] [SW], Java EE7: DayTrader7 Sample 2022. URL: https://github.com/wasdev/sample.daytrader7.

[9] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. 2022. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, (July 2022), 869–884. ISBN: 978-1-939133-29-62. https://www.usenix.org/conference/atc22/presentation/khrabrov.

[10] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5, 1, Article 7, (May 2008), 32 pages. DOI: 10.1145/1369396.1370017.

[11] [SW], Native Image 2023. URL: https://www.graalvm.org/latest/reference-manual/native-image/.

[12] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 340–350. DOI: 10.1109/CGO51591.2021.9370314.

[13] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wuerthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In (Feb. 2019). DOI: 10.5281/zenodo.2328430.

[14] Aleksandar Prokopec et al. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 17. DOI: 10.1145/3314221.3314637.

[15] [SW], Spring PetClinic Sample Application 2023. URL: https://github.com/spring-projects/spring-petclinic.

[16] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2013). Association for Computing Machinery, Indianapolis, Indiana, USA, 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581.

[17] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. 2, OOPSLA, Article 124, (Oct. 2018), 23 pages. DOI: 10.1145/3276494.


Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009