JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud

Extended abstract; originally published in USENIX ATC'22

Alexey Khrabrov
University of Toronto

Marius Pirvu
IBM

Vijay Sundaresan
IBM

Eyal de Lara
University of Toronto

1 Motivation

Java virtual machines (JVMs) rely on just-in-time (JIT) compilers to improve application performance by converting bytecodes into optimized machine code at runtime. Unfortunately, JIT compilation can introduce significant runtime overheads in terms of processing power and memory. The extra CPU cycles needed for compilation can interfere with applications' progress, delaying their start-up, increasing their warm-up time or affecting the response time and quality of service (QoS). Similarly, the data structures allocated by the JIT compiler create unpredictable spikes in memory usage resulting in higher memory footprint. In our experiments, JIT compilation accounted for up to 50% of CPU time used during the start-up and warm-up phases of the applications, and for up to hundreds of MBs of memory footprint.

The competition for resources between the application and the JIT is more intense in CPU and memory constrained environments such as containers and VMs found in cloud datacenters that aim to maximize resource utilization and application density. Automatic scaling of cloud applications is done by launching and shutting down instances based on load. Frequent restarts of applications pose serious challenges to JVM-based workloads due to the high start-up overhead of JIT compilation, which needs to be amortized over a long execution period. Short-running application instances such as function-as-a-service (FaaS) are also becoming increasingly common in cloud computing. The memory overhead of JIT compilation is more significant for smaller (in terms of overall memory usage) application instances which are common in the cloud (e.g., microservices).

2 Limitations of Existing Work

One way to circumvent the negative effects of JIT compilation is to use static ahead-of-time (AOT) compilation [3, 5, 22]. Its inherent limitation is the *closed world assumption*: all the code that can execute at runtime must be available at compile time. This assumption is only compatible with a subset of Java and JVM bytecodes, severely limiting support for dynamic JVM features. Moreover, static AOT compilers often generate suboptimal code due to the lack of realistic runtime profiling data and compatibility with a wide range of target machines.

JIT compilation overhead can be reduced by caching and sharing compiled code among JVMs [7, 10, 11, 23]. This approach does not completely eliminate the need for a JIT compiler due to cache misses as the set of compiled methods can vary from run to run. Shareable compiled code is typically slower than regular JIT-compiled code since it has to meet certain constraints in order to be relocatable and usable in a different JVM instance. Therefore, performance critical methods still need to be JIT-compiled in order to achieve peak throughput. Since such compilations are responsible for most of the JIT compiler memory overhead, this approach cannot effectively reduce peak memory usage.

Using a cache of compiled methods in ephemeral execution environments (e.g., containers) that are typical in the cloud requires either (i) shipping a pre-populated cache with the application, or (ii) sharing the cache locally between JVMs on a given host (populating it at runtime). The 1st approach puts additional burden on application developers, increases the complexity and cost of continuous integration and deployment, and results in larger image sizes. The 2nd approach forces schedulers to pack instances of the same application on the same host, leading to "hot spots" during load spikes when multiple instances contend for (often oversubscribed) resources and increasing applications' exposure to host failures.

Existing work on JVM checkpoint/restore [13, 15, 21] has only explored snapshots of well-defined state of the JVM process after the start-up phase of the application (excluding any methods compiled under load), and suffers from the same usability issues as shipping a cache of compiled methods. Reusing live JVM instances [14, 18] only allows reuse of compiled code within the same host and incurs a significant idle footprint.

JIT compiler disaggregation addresses the compilation overheads by decoupling the JIT from the JVM and running it in a separate remote process. Previous work on remote JIT compilation [8, 9, 12, 16, 17, 19, 20] focused on embedded and mobile devices where local JIT is prohibitively expensive in terms of memory, CPU, or energy consumption, and has not considered overall resource usage including the compiler server. Existing systems are based on simplified JITs that either do not rely on dynamic JVM runtime information, or only support static AOT compilation of a subset of Java, or assume that all the information needed to compile a method is included in the compilation request, which becomes impractical in a complex modern JIT.

While remote JIT reduces overall memory usage (since the spikes of maximum memory usage from multiple clients are unlikely to align), on the downside, it can result in higher system-wide CPU usage, especially for short-running workloads common in cloud computing, as we show in our evaluation. The CPU cost and latency of each compilation in this setting is higher compared to local JIT due to communication overheads. JIT compilation overheads are not eliminated, but rather transferred to a different host, at the expense of additional networking and serialization costs.

3 Key Insights

We argue that in order to achieve the full benefits of disaggregated JIT, the compiler server resources must be effectively shared between multiple client JVMs by making it possible to reuse compilations of common methods. Taking advantage of the fact that there is often a lot of code shared between JVMs in cloud workloads (e.g., multiple application instances for autoscaling), we cache JIT-compiled code at the compilation server and reuse it in multiple client JVMs. The main goal of our design is to reduce the overall CPU usage by amortizing the compilation costs over many JVMs. Reusing compiled code happens transparently and does not add any complexity to application development. Figure 1 illustrates the architecture of our system.

Reusing JIT-compiled code across multiple JVM instances is a challenging task: the code cannot be simply plugged into a different JVM process in the general case since it contains pointers to runtime entities (e.g., class metadata and other compiled methods) located at different virtual addresses in different JVMs, and relies on runtime assumptions (e.g., about class hierarchies) that might not hold in a different JVM environment. Due to the dynamic nature of the JVM, identifying and locating runtime entities and verifying assumptions across JVM processes is

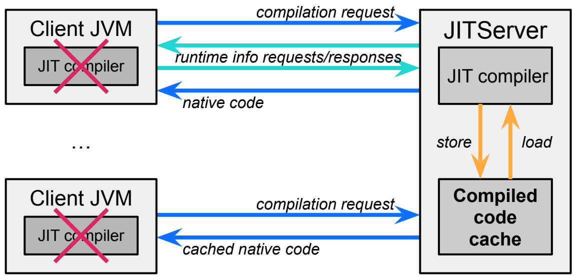

Figure 1: JITServer architecture

more difficult compared to relocating statically-compiled code in languages like C (where entities are simply identified by a globally unique name).

Our solution is to cache compiled method bodies at the JIT server in a *serialized* format that includes *serialization records* that describe how to adapt the code in another JVM instance. We use *relocation records* to update the addresses of runtime entities and *validation records* to verify the assumptions made by the compiler. Both relocations and validations can always be expressed in terms of Java classes, thus the main building block or serializing JIT-compiled code is identifying classes that are equivalent *at runtime* across JVMs (the same Java class definition can result in distinct runtime classes).

Serialization records store enough information to uniquely identify classes across JVMs from the point of view of the JIT compiler, effectively assigning a global ID to each runtime class. We identify class loaders (which are necessary to lookup classes at runtime) across JVMs by the name of the first loaded class - a heuristic that works well in practice. We use secure cryptographic hashes (e.g., SHA-256) of immutable class metadata (including method bytecodes) for each class and interface in the inheritance chain to efficiently verify class equivalency in any JVM instance.

Remote JIT is less susceptible to failures than other disaggregated designs (e.g., memory disaggregation) since it has no shared hard state. Unlike the case of mobile and IoT devices targeted in previous remote JIT work, network latencies in cloud datacenters are relatively low, and our evaluation shows that remote JIT performs very well in this setting. Modern cloud applications themselves are also typically distributed, thus remote JIT does not exacerbate the reliability and latency concerns.

Remote JIT shifts resource provisioning complexity from the application to the infrastructure, which can be arguably beneficial. Local JIT requires application developers to manage the complexity and extra costs of over-provisioning memory (which goes unused after warm-up) and CPU (to maintain QoS despite JIT activity during warm-up) for each JVM. Instead, the operator's effort to

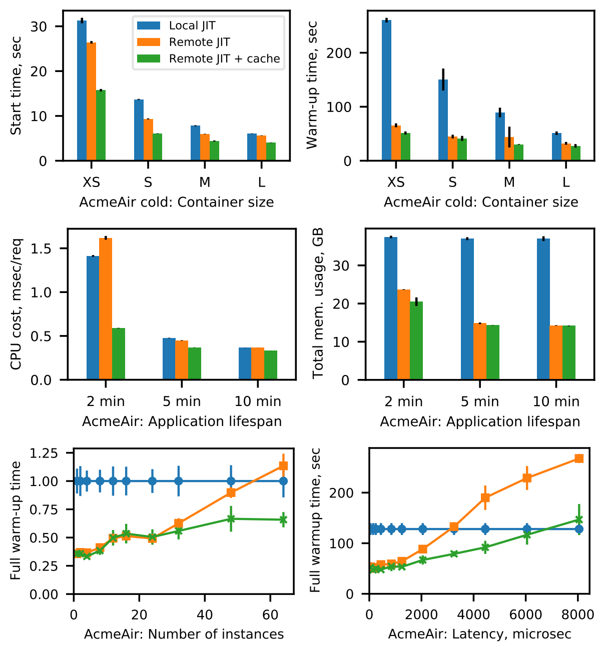

Figure 2: Evaluation results for AcmeAir *(lower is better)*

setup compiler server autoscaling can be reused many times across applications.

4 Main Contributions and Results

We describe the design and implementation of JITServer - our disaggregated JIT caching compiler in Eclipse OpenJ9 [2], a popular open source JVM. Unlike previous work, JITServer is compliant with the JVM specification and implemented in a production grade JVM with a sophisticated JIT. We provide insight into the challenges of implementing remote JIT in a dynamic environment such as the JVM and the ways to solve them, such as caching JVM runtime information at the server to reduce communication overheads and correctly invalidating these caches to support dynamic class loading and unloading.

We propose a novel mechanism that facilitates caching of compiled native code in a remote JIT compilation system and enables correct, transparent and efficient reuse of such code by JVMs running on different machines. We demonstrate that caching is necessary to achieve the full benefits of JIT compiler disaggregation.

We present the first (to the best of our knowledge) study of remote JIT in the context of cloud computing. We evaluate JITServer performance using 3 representative end-to-end application benchmarks [1,4,6]. The main results for one of the benchmarks (AcmeAir) are shown in Figure 2. More detailed results (including other benchmarks) can be found in the full paper. Our evaluation shows that:

- Remote JIT cannot be fully efficient without caching and reusing compiled code - it often increases total system-wide CPU cost (by up to 21%).
- Caching allows JITServer to reduce cluster-wide resource usage (by up to 77% for CPU and up to 62% for memory), allowing more efficient, higher density deployments of JVM-based applications in the cloud.
- JITServer significantly reduces application start-up and warm-up times (by up to 58% and 87% respectively) and memory footprint, especially in smaller containers, without trading-off peak throughput.
- Caching dramatically improves JITServer scalability and allows it to effectively handle more concurrent clients and tolerate significantly higher network latency (up to 8 ms). Note that network bandwidth has limited effect on performance compared to latency since JITServer exhibits a latency-bound communication pattern (small request-reply message pairs).

5 Current and Future Work

We are currently investigating ways to improve the JIT compiler in the disaggregated setting such as utilizing profiling data from multiple clients. Caching and reusing profiling data via JITServer is a promising approach to further amortize JIT-related overheads across multiple JVMs for compilations that are not cached (up to 30% of methods depending on the application) by reusing the most "expensive" input to the JIT compiler (profiling data) and compiling methods earlier for future JVM instances with a similar profile.

We plan to explore prefetching of server-cached code to hide compilation latency and further reduce cold start times. We will study the trade-off between the performance of relocatable code and the JITServer cache hit rate, and explore sharing compiled code across applications that use common frameworks and libraries. We plan to apply JITServer to other workloads such as FaaS, microservices, and data analytics.

One of the main prerequisites for integrating JITServer into serverless/FaaS frameworks is the ability to automatically size and scale compilation resources. Efficient horizontal autoscaling requires provisioning multiple relative small (in terms of resources) JITServer instances that are started and stopped on demand. As we have shown in this paper, caching compiled code at the server is essential for reducing overall CPU usage and other performance aspects, thus starting new JITServer instances with a cold cache would be suboptimal. We are currently working on sharing the cache of compiled methods across multiple JITServer instances through persistent snapshots. The persistence also allows scaling a JITServer deployment down to zero while preserving the cache.

References

[1] AcmeAir sample and benchmark. https://github.com/blueperf/acmeair-monolithic-java.

[2] Eclipse OpenJ9. https://www.eclipse.org/openj9/.

[3] GraalVM native image. https://www.graalvm.org/reference-manual/native-image/.

[4] Java EE7: DayTrader7 sample. https://github.com/wasdev/sample.daytrader7.

[5] JEP 295: Ahead-of-time compilation. https://openjdk.java.net/jeps/295.

[6] Spring PetClinic sample application. https://github.com/spring-projects/spring-petclinic.

[7] D. Bhattacharya, K. B. Kent, E. Aubanel, D. Heidinga, P. Shipton, and A. Micic. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, PACRIM, pages 1–6, 2017.

[8] Guangyu Chen, Byung-Tae Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices. *IEEE Trans. Parallel Distrib. Syst.*, 15(9):795–809, September 2004.

[9] Guilin Chen, Byung-Tae Kang, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Energy-aware compilation and execution in Java-enabled mobile devices. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 34. IEEE Computer Society, 2003.

[10] Ben Corrie and Hang Shao. Class sharing in Eclipse OpenJ9. https://developer.ibm.com/tutorials/j-class-sharing-openj9/, 2018.

[11] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, page 155–177, Berlin, Heidelberg, 2002. Springer-Verlag.

[12] Bertrand Delsart, Vania Joloboff, and Eric Paire. JCOD: A lightweight modular compilation technology for embedded Java. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, page 197–212, Berlin, Heidelberg, 2002. Springer-Verlag.

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 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*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Cloneable JVM: A new approach to start isolated Java applications faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 1–11, New York, NY, USA, 2007. ACM.

[16] Joel Koshy, Ingwar Wirjawan, Raju Pandey, and Yann Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Ad Hoc Networks*, 6(8):1185–1200, 2008.

[17] Han B. Lee, Amer Diwan, and J. Eliot B. Moss. Design, implementation, and evaluation of a compilation server. *ACM Trans. Program. Lang. Syst.*, 29(4):18–es, August 2007.

[18] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 383–400, Berkeley, CA, USA, 2016. USENIX Association.

[19] Matt Newsome and Des Watson. Proxy compilation of dynamically loaded Java classes with MoJo. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, page 204–212, New

York, NY, USA, 2002. Association for Computing Machinery.

[20] Radu Teodorescu and Raju Pandey. Using JIT compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, page 76–95, Berlin, Heidelberg, 2001. Springer-Verlag.

[21] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019.

[23] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. ShareJIT: JIT code cache sharing across processes and its practical implementation. *Proc. ACM Program. Lang.*, 2(OOPSLA):124:1–124:23, October 2018.