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

Alexey Khrabrov, *University of Toronto;* Marius Pirvu and Vijay Sundaresan, *IBM;* Eyal de Lara, *University of Toronto*

https://www.usenix.org/conference/atc22/presentation/khrabrov

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

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

Alexey Khrabrov
University of Toronto

Marius Pirvu
IBM

Vijay Sundaresan
IBM

Eyal de Lara
University of Toronto

Abstract

Managed runtimes such as the Java virtual machine (JVM) rely on just-in-time (JIT) compilers to improve application performance by converting bytecodes into optimized machine code. Unfortunately, JIT compilation introduces significant CPU and memory runtime overheads. JIT compiler disaggregation is a technique that decouples the JIT from the JVM and ships compilation to a separate remote process. JIT disaggregation reduces overall memory usage; however, its communication overheads result in higher system-wide CPU usage.

JITServer is a disaggregated caching JIT compiler we implemented in the Eclipse OpenJ9 JVM. It improves system-wide resource utilization by enabling the caching of compiled native code and its reuse in JVMs running on different machines. JITServer is transparent to the application developer, and supports all the dynamic features in the JVM specification. In our experiments, JITServer reduced overall CPU cost by up to 77%, overall memory usage by up to 62%, application start time by up to 58% and warm-up time by up to 87%.

1 Introduction

Java virtual machines (JVMs) rely on just-in-time (JIT) compilers to improve the performance of Java applications by converting the bytecodes of the application into optimized machine code. Since this transformation is done at runtime, the JIT has the ability to tailor-fit the generated code for a specific application instance and its execution environment. On the downside, 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. Similarly, the data structures allocated by the JIT compiler create unpredictable spikes in memory usage, which increase memory footprint and can lead to performance degradations (due to paging) and out-of-memory failures. In our experiments, JIT compilation accounted for up to 50% of CPU time used during the start-up and warm-up phases, 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 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 Java workloads due to the high start-up overhead of JIT compilation, which needs to be amortized over a long execution period. 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).

JIT compiler disaggregation is a technique that addresses these overheads by decoupling the JIT from the JVM and running it in a separate remote process. The JIT no longer steals CPU cycles from the application, which leads to more predictable behavior and better quality of service, and improves application warm-up in CPU constrained environments. Memory footprint spikes are also eliminated, enabling smaller containers, higher application density, and reduced costs in the cloud. Moreover, remote JIT simplifies resource provisioning: the user only has to consider CPU and memory required for application execution, while compilation resources can be scaled independently of the applications.

While JIT disaggregation reduces overall memory usage, on the downside, it can result in higher system-wide CPU usage. The CPU cost and latency of each compilation in this setting is higher compared to local JIT due to communication overheads. JIT overheads are not eliminated, but rather transferred to a different host, at the expense of additional networking and serialization costs.

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. Unfortunately, reusing dynamically compiled native code across multiple JVM instances is a challenging task. JIT-compiled code cannot be simply plugged into a different JVM in the general case since it often contains pointers to runtime entities that are located at different addresses in different JVMs, and relies on runtime assumptions that might not hold in a different JVM environment. Due to the dynamic nature of the JVM, locating runtime entities and verifying assumptions across JVM processes is more difficult compared to relocating statically-compiled code in languages like C. The key idea is to use secure hashes of immutable class metadata to efficiently detect equivalent classes and methods across JVMs.

In this paper, we describe the design and implementation

of JITServer - our disaggregated JIT compiler in Eclipse OpenJ9 [6], a popular open source JVM. JITServer caches compiled native bodies and reuses them for future compilation requests for the same methods from other client JVMs. This enhancement dramatically decreases compilation latency for cache hits and significantly reduces overall resource usage by amortizing compilation costs over multiple clients. Caching compiled code at the server happens transparently and does not add any complexity to application development. Our main use case is running multiple application instances in a cloud datacenter in, e.g., containers with resource limits. Remote JIT might not be beneficial if the JVM has plenty of resources for local JIT, or if the network latency is high.

This paper makes the following contributions:

- 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 show that caching is necessary to achieve the full benefits of JIT compiler disaggregation.
- We describe the design and implementation of JITServer. Unlike previous work, our solution is compliant with the JVM specification, does not rely on simplifying assumptions, and is 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.
- We present the first (to the best of our knowledge) study of remote JIT in the context of cloud computing. We show that JITServer improves start time, warm-up time, CPU and memory usage, without trading-off peak throughput, allowing more efficient, higher density deployments of JVM-based applications in the cloud. In our experiments, JITServer reduced overall CPU cost by up to 77%, overall memory usage by up to 62%, application start time by up to 57% and warm-up time by up to 87%.

The rest of the paper is organized as follows: Section 2 provides a survey of related work and motivates our solution; Section 3 presents the design of JITServer and its novel mechanism for reusing compiled code in multiple JVMs; Section 4 evaluates the performance of our system; finally, Section 5 concludes the paper and explores future work directions.

2 Motivation and Related Work

In this section we present a survey of existing solutions for the JIT overhead problem and discuss their limitations.

2.1 Static AOT Compilation in the JVM

One way to circumvent the negative effects of JIT compilation is to use static ahead-of-time (AOT) compilation. HotSpot JVM used to include a (now deprecated [10]) static AOT compiler `jaotc` [9] that compiled the bytecode of an explicitly specified list of Java classes or `.jar` files into native code. GraalVM Native Image [7] compiles a Java application including all the classes it uses (determined by static analysis) into a standalone native executable, and can run parts of the application initialization code at AOT compile time [31].

However, an inherent limitation of static AOT compilation is the *closed world assumption*: all the code that can execute at runtime must be available at compile time. This assumption severely limits support for dynamic JVM features such as custom dynamic class loading, class definition and redefinition at runtime, and `invokedynamic` bytecodes. Static AOT only supports a subset of Java and JVM bytecodes.

On the performance front, static AOT compilers typically do not take advantage of the latest CPU features, because the code they produce must be compatible with a wide range of target machines. Moreover, the lack of runtime profiling information can lead to suboptimal performance. While it is possible to use profiling information at build time, it requires a realistic workload, which makes application development more difficult. In addition, performance profile of a given method can change between application phases, and achieving peak performance in such cases still requires dynamic recompilation at runtime.

2.2 Sharing Compiled Code between JVMs

JIT overhead can be reduced by caching and sharing compiled code among JVMs. Examples include ShMVM [20] (based on HotSpot), ShareJIT [32] (based on Android Runtime), and the Shared Classes Cache (SCC) [16, 19] and dynamic AOT compilation in OpenJ9. We focus on the latter as the more recent and practical implementation of this approach.

SCC in OpenJ9 is a memory mapped file used to cache compiled code and the internal representation of immutable class metadata. The SCC is populated in a *cold* run and is subsequently consumed by other JVM instances in *warm* runs. The SCC improves start-up and warm-up performance in the warm runs since the class metadata is already available in a pre-processed format and does not need to be parsed from the class files, and loading cached compiled methods is much less CPU-intensive than JIT-compiling them. This approach is called *dynamic AOT compilation*: methods are compiled and stored in the SCC during execution, in contrast with static AOT where the code is compiled before it runs.

Unfortunately, this approach does not completely eliminate the need for a JIT compiler for two reasons: (i) the hit rate in the SCC is not 100% as the set of compiled methods can vary from run to run; and (ii) dynamic AOT code can be slower that 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 are still JIT-recompiled with more optimizations in order to achieve peak throughput. Since such compilations are responsible for most of the JIT memory overhead, this approach cannot effectively reduce peak memory usage.

While it is possible to ship a pre-populated SCC with an ap-

plication, the complexities involved often make it impractical. The associated increase in image size can be significant up to hundreds of MBs (63-128% increase for the applications we used in our evaluation). A larger image size adds overhead on the critical path of deployment and contributes to the cold start latency. Dynamic AOT code makes assumptions about the execution environment such as target CPU instruction set, GC algorithm (its reference read and write barriers), and heap size (determines compressed pointer shift). Shipping a pre-populated SCC requires either maintaining multiple versions for all possible combinations of CPU generations and JVM parameters (which complicates deployment), or generating suboptimal portable code that works across all configurations.

Managing the pre-populated SCC puts additional burden on application developers and increases complexity and cost of continuous integration and deployment. Caching methods compiled during warm-up requires simulating a realistic workload, which can be a complex task. Creating a fully populated SCC can also increase application build times by up to orders of magnitude since it can take minutes of application run time to achieve full warm-up. Anecdotal evidence (e.g. Docker images for OpenJ9 [2] and Open Liberty [11] - a popular Java framework optimized for OpenJ9, and the Java runtime in the OpenWhisk [4] serverless platform) suggests that in practice, the SCC is typically pre-populated only by starting up and shutting down an application instance, and does not include any methods compiled during the warm-up phase.

Another way to leverage the SCC is to share it locally between JVMs on a given host, populating it dynamically at runtime instead of pre-populating it at application build time. However, this approach also has drawbacks. Sharing the SCC between applications creates the potential for side-channel attacks and other security issues. Thus if sharing is limited to a single application, the scheduler is forced to pack instances of the same application on the same host, which can lead to "hot spots" during load spikes when multiple instances of the same application contend for (often oversubscribed) resources. This approach also increases applications' exposure to individual host failures. Managing per-application SCC volumes shared between containers also complicates deployment.

2.3 Checkpointing and Reusing JVM Processes

Another approach to circumvent JVM slow start is to checkpoint the state of a "warm" JVM process and restore it when starting a new application instance. Cloneable JVM [24], ReplayableJVM [30], and Catalyzer [22] explored checkpointing well-defined and deterministic state after the start-up phase of the application. Such snapshots do not include any methods compiled during warm-up under load, and still require JIT compilation to reach peak performance. Checkpointing also suffers from the same usability issues as shipping a pre-populated cache of compiled methods: additional developer effort and having to generate slower portable code or maintain multiple versions for different CPUs and JVM configurations.

HotTub [27] takes a somewhat different approach of reusing JVM processes to keep the JVM state "warm" for the next run of the same or similar application. Photons [23] co-locates multiple copies of a serverless function as threads within the same JVM instance. These systems only reuse compiled code within the same machine, and can only persist it across non-concurrent invocations by keeping the pre-warmed JVMs running, which incurs a significant idle footprint.

2.4 Remote JIT Compilation

Remote JIT has been originally proposed in research that focused on embedded, mobile, and IoT devices where local JIT is prohibitively expensive in terms of memory, CPU or energy consumption [17,18,21,25,28,29]. JCOD [21], MoJo [28] and VM* [25] are based on simplified JVMs and JIT compilers that either do not rely on dynamic JVM runtime information, or only support static AOT compilation of a subset of Java.

Lee et al. [26] describe a JIT compilation server based on Jikes RVM, a research JVM written in Java. To the best of our knowledge, their work is the state-of-the-art in the literature on remote JIT for the JVM. Its design assumes that all the information needed to compile a method is included in the compilation request, which becomes impractical in a complex modern JIT. The authors do not consider overall resource usage (including the server), and only use simulations to evaluate performance with multiple clients.

Foremost, these previous approaches do not reduce, and in fact can increase, system-wide CPU usage. Each individual remote compilation consumes more CPU time than its local equivalent (assuming homogeneous hardware) due to communication overheads, and takes more time overall due to network latency. As we show in our evaluation (see Section 4.2), remote JIT can increase overall CPU usage, especially for short-running workloads that are common in modern cloud computing. In this paper, we leverage caching compiled methods at the server to reduce the overall CPU cost by amortizing it over multiple clients. In addition, we show that for cloud workloads, JITServer significantly reduces overall memory consumption since the spikes of maximum memory usage from multiple clients are unlikely to align.

Azul Cloud Native Compiler [5] is a recently released remote JIT for the Azul JVM. Unfortunately, there is only limited technical information available about the design of this proprietary closed-source system. In particular, it is not known if it implements caching, or how it affects system-wide resource usage. We are unable to compare performance against it since its license forbids publishing benchmarking results.

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 (the main target of 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 dis-

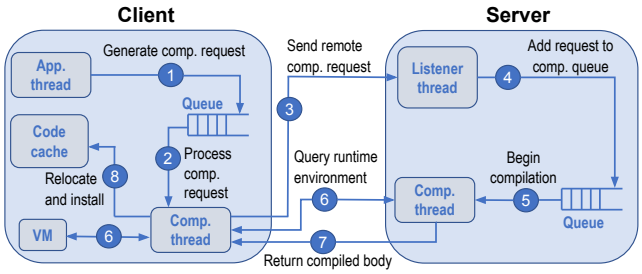

Figure 1: Remote compilation mechanism

tributed, 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 setup JITServer autoscaling can be reused many times across applications.

3 Design and Implementation

In this section we present an overview of JITServer design and implementation. We start with the description of the remote compilation mechanism, and then explain how we enable caching compiled methods to be reused by multiple clients.

3.1 Remote Compilation Mechanism

Figure 1 shows the high-level design of remote compilation in JITServer. The *client* on the left is a complete JVM running an application, while the *server* on the right is a separate, possibly remote process offering a JIT compilation service shared by multiple client JVMs connected to it. In this paper, we focus on the mechanisms. We plan to explore automatic sizing and scaling of JITServer resources in future work.

Existing remote JIT designs assume that each compilation is a simple request-reply operation that requires a single network round-trip since all the data required by the server is either already available there or included with the request [26]. While this was feasible for simple JIT compilers in previous work, such a design proved difficult to implement in a modern JVM. The JIT in OpenJ9 has over 100 optimization passes that use over 100 types of JVM state queries, compared to ~40 optimizations and 12 types of state in [26]. We took a different approach that resulted in lower implementation effort and complexity: a compilation request only carries data that is likely to be used by most compilations, and remaining data is requested from the client on demand, cached at the server, and invalidated when necessary to ensure correctness.

OpenJ9 runs a number of background threads that perform JIT compilation concurrently with application threads. It uses *tiered* compilation: methods are scheduled for compilation at lower optimization levels once they reach certain invocation thresholds, and particularly hot methods that consume a significant portion of CPU time are scheduled for recompilation at higher optimization levels. The bytecode interpreter and the sampling thread add compilation requests to a queue consumed by the compilation threads. The number of active threads is adjusted dynamically: additional ones are activated when the JIT is starved of CPU time (e.g. if there is a large number of application threads) or the queue becomes too large. Compilation threads compile one method at a time, and each compilation is a single-threaded task.

JITServer uses the same basic design with compilation queues and threads on both the client and the server. At Step 1, application threads add requests to the compilation queue. At Step 2, a client compilation thread dequeues a request and sends a remote compilation request to the server (Step 3). The client sends enough data for the server to start the compilation, such as the bytecodes of the method. A listener thread on the server accepts the connection and enqueues a compilation request (Step 4), and a compilation thread reads the request data from the network and starts the compilation (Step 5).

During remote compilation, the server may issue various queries to the client requesting information about classes, methods, fields, execution environment, etc. (Step 6). When the compilation is complete, the server sends the compiled body to the client (Step 7) along with metadata that needs to be instantiated at the client. The client performs necessary relocations (e.g. fixes up calls to runtime helper methods) and installs the compiled method in its own code cache (Step 8) - now it is ready to be executed. As long as the compilation queue is not empty, the client thread keeps the connection open and reuses it for subsequent requests.

Each compilation is performed by a dedicated pair of threads - one on the server and one on the client. The number of in-progress compilations is limited by the maximum number of client threads (16 in the current implementation). Our evaluation shows that this value works well in practice. We used a maximum of 128 server threads in our experiments.

The server JIT compiler in our current implementation is identical to the local OpenJ9 JIT compiler and uses the same set of optimizations and heuristics that control its behaviour. All compilations are performed remotely. We focus on evaluating the benefits of JIT disaggregation and caching on their own in this paper, and plan to explore how the compiler can be improved in the disaggregated setting in our future work.

3.2 Caching JVM Runtime Information

Remote compilation of large or complex methods can require exchanging a large number of messages between the client and the server. To reduce the negative effect of network latency on performance, the server aggressively caches runtime information received from the clients. Note that this is different from caching the resulting compiled code, which we describe in Section 3.4. Caching is essential for performance: in an early JITServer prototype that did not implement such

caching, client threads alone used ~10x more CPU time than local JIT. Caching reduced the average number of messages per compilation from over 1000 to ~40.

The server handles requests from multiple clients in parallel and maintains their data in *client sessions* identified by a unique client ID included with each compilation request. A session is created when a client first connects to the server, and destroyed when the client terminates or stays inactive for a long time. Most of the information that does not change throughout the client's lifetime (e.g. JVM configuration and class metadata for primitive types) is sent with the first compilation request. Mutable client state needs to be synchronized with the server as classes are being loaded, unloaded, or redefined, and profiling information is updated. We describe how we handle caching of important types of client data.

Class Metadata Java classes are represented in OpenJ9 as data structures called ROMClasses, which represent immutable data, including method bytecodes, and RAMClasses, which contain the mutable data and point to the ROMClasses they are based on. Multiple RAMClasses can use the same ROMClass, e.g. if they are loaded by different class loaders.

The server maintains mirrors of the clients' ROMClasses and RAMClasses. To reduce memory consumption, it stores a single copy of each unique ROMClass that is shared by all the clients using it. To handle class unloading and redefinition, the client sends with each compilation request the list of classes unloaded or redefined since the previous compilation request, which are in turn deleted from the server cache.

Server compilation threads are synchronized with class unloading and redefinition using a reader-writer lock. Threads acquire this lock for writing before deleting the entries, and acquire it for reading while performing a compilation. When runtime information queries are sent to the client, the lock is released, and the client's reply will also indicate whether class unloading happened during this compilation, in which case the compilation is aborted and retried.

Class Hierarchy Table CHTable (class hierarchy table) is an OpenJ9 data structure that captures relationships between classes and enables fast discovery of such relationships used for speculative optimizations, e.g. devirtualization. The server mirrors the client's CHTable and keeps it up to date with incremental updates sent along with the compilation requests.

To guarantee functional correctness, the server must process class unloading/redefinition and CHTable updates in the same order as the client. This requirement imposes a partial order on compilation requests: *critical* requests that carry updates must be processed in the same order as they originated at the client, while other *non-critical* requests between subsequent critical requests can be reordered. We use a sequencing scheme to order critical requests: if the server receives an update with an out-of-order sequence number, it suspends the compilation thread until the update with the expected sequence number arrives. If the expected message is lost, after a timeout, the server clears cached data for the client and restarts the update mechanism by requesting the entire CHTable.

Profiling Data While interpreting Java methods, OpenJ9 collects profiling data about branch direction and targets of virtual and interface calls and `instanceof`/`checkcast` operations. The optimizer at the server makes extensive use of this data and issues many queries to the client. To reduce the number of messages, we batch profiling data requests by sending the data for all the bytecodes in the method in a single reply.

Unlike class hierarchy information, imprecise profiling data does not affect functional correctness of generated code. Profiling data for already compiled methods (which can be inlined into newly compiled ones) has stabilized and can be cached without significant effect on performance. In contrast, profiling data for interpreted methods continue to be accumulated, and using a stale version can lead to bad optimization decisions. Thus, the server caches this data only for the duration of the current compilation.

3.3 Reliability and Security

The client JVM in our design still includes a fully functional local JIT compiler. Out-of-process compilation makes Java applications more resilient to JVM crashes caused by intermittent software bugs in the JIT compiler. JITServer only maintains soft state, allowing transparent failure handling: after a server crash, the client JVM can switch to a different JITServer instance or compile the method locally (which is unlikely to trigger the same bug again) and continue execution, and the failed JITServer instance can be simply restarted.

We assume a security model where the server instance and all the client JVMs that connect to it are in the same security domain: the clients need to trust the server to generate correct native code. JITServer supports encrypted communication using OpenSSL. Encryption adds a relatively small but non-negligible overhead: it increases start time by up to 9%, warm-up time (to 90% of peak throughput) by up to 5%, and CPU time used by the JIT by up to 5%. Unencrypted communication can be used in settings with an isolated network (e.g. on-premises cloud deployments) to reduce the overhead.

Even if communication is encrypted, remote JIT can potentially introduce side channels for an in-network attacker that can observe the timing and sizes of messages exchanged between the server and the client JVMs. However, the same issue applies to any distributed or client-server application, and in practice does not prevent their widespread adoption.

3.4 Reusing Dynamically Compiled Code

OpenJ9 is capable of producing relocatable code (aka dynamic AOT code) that can be reused in a different JVM instance running on the same machine. It maintains a Shared Classes Cache (SCC) - a memory-mapped file shared by JVMs on the same host that stores immutable class metadata and dynamic AOT code. These cached methods include additional metadata that is used to locate and patch pointers to JVM runtime entities such as RAMClasses (see Section 3.2)

residing at different addresses in different JVMs, and to verify that assumptions (e.g. about class hierarchies) made during compilation still hold in a different JVM environment.

Dynamic AOT methods refer to SCC entities in their *validation and relocation records* by offset within the SCC. While this approach is very efficient when running on a single machine, this means that dynamic AOT code cannot be reused as-is on a different machine (with a different SCC). Since the order of class loading and dynamic AOT compilations is not deterministic, SCC entities will generally reside at different SCC offsets on another machine, making dynamic AOT code generated on another machine invalid. The fundamental source of this problem is the tight coupling of the dynamic AOT code with the SCC it is stored in. We propose a new scheme to store these artifacts independently.

Our caching mechanism is based on the existing OpenJ9 infrastructure for relocating dynamically compiled code, but uses a different way of identifying runtime entities such as classes across JVMs. Our novel scheme allows us to decouple compiled code from class metadata and enable its reuse in any JVM running on any host. We store compiled methods in a *serialized* format that refers to JVM runtime entities by globally unique identifiers instead of SCC offsets. When another client JVM requests a compilation of the same method, the server replies with a cached serialized version. The client then *deserializes* the method by finding the corresponding runtime entities, and then relocates and loads the native code.

Some compilations (e.g. hot methods at higher optimization levels) do not use the relocatable format in order to maximize the performance of generated code, since relocation limits optimization opportunities. The resulting hit rate in the JITServer cache is 85-93% during the start phase and 68-76% during warm-up in our experiments, depending on the application. In our future work, we will explore compiling more methods as relocatable code to increase the cache hit rate.

The performance of JIT-compiled code is affected by the quality of profiling data, leading to a trade-off between compiled code reuse and possible performance degradation due to differences in profiles across the clients. We focus on the common use case of sharing between instances of the same application where profiles are likely similar (e.g. horizontal autoscaling, FaaS, data-parallel computation). In other cases, we expect the effect to be limited since particularly hot method compilations most affected by conflicting profiles are not cached and rely on individual clients' profiles. We plan to investigate the effect of profiling data variability on the performance of reused native code in our future work.

3.5 Method Serialization Mechanism

The main building block of dynamic AOT relocations is identifying equivalent classes and methods across JVMs. Examples include *inlining guards* (checking an object's class before executing the inlined body) and calls to other compiled methods. Such instruction sequences contain addresses of RAMClasses and methods that need to be patched in a different JVM.

We add another level of indirection to relocatable code by effectively serializing relocation and validation records. For each runtime entity they refer to, the server stores a corresponding *serialization record* with enough information for a client JVM to find the entity and verify that it is equivalent to the one used in the original compilation. Each serialization record also contains the offset from the start of the AOT method body to the location of the corresponding SCC offset field stored in the validation or relocation record.

When the server performs a dynamic AOT compilation in response to a client request, it serializes the compiled method and stores it in its in-memory cache. Serialized methods are self-contained and can be persisted to disk and shared across multiple JITServer instances. In our future work, this will enable efficient autoscaling of JITServer resources by launching new instances with a warm cache persisted from an existing instance. When a different client JVM receives a serialized AOT method, it iterates through the serialization records, looking up the corresponding entities and updating the SCC offsets to them with their local versions. After successful deserialization, the client proceeds to store the resulting method body in its local SCC (so that execution environments that do not get torn down can take advantage of it in the next run), and loads the method as regular dynamic AOT code.

If any lookups or validity checks fail, deserialization is aborted and the client JVM requests a regular non-cached compilation from the server. Deserialization failures occur infrequently during normal operation: the failure rate for the applications we used in our evaluation is 1-5% during the start phase and less than 1% during warm-up. Such failures happen because the set of classes loaded by the time a method is compiled varies from run to run, therefore a small number of lookups fail as the classes have not yet been loaded.

Relocation and validation records can refer to the following SCC entities: *ROMClass* - immutable part of class metadata; *ROMMethod* - immutable part of method metadata including its bytecodes (part of ROMClass); *class chain* - a list of ROMClass SCC offsets for classes and interfaces a given class extends and implements.

Identifying Classes and Methods We identify a class across JVMs using a combination of its fully-qualified name and a secure hash (e.g. SHA-256) of the ROMClass. We use the hash to efficiently check that a client JVM's version of the class is the same as the one used during compilation. Since the ROMClass contains the full description of the class including the bytecodes of all methods, a matching hash guarantees correctness. We identify methods by their defining class, name, and signature (types of parameters and return value).

We use class chains to verify that the whole inheritance chain of a given class is the same across two JVMs. For example, if one of the superclasses or interfaces of a given class is redefined at runtime in the JVM that loads the method, the class chain will not match, even though the class itself has

not changed. The serialization record for a class chain is a list of class serialization records for each class in the chain.

Identifying Class Loaders Since Java classes can be loaded by application-defined class loaders, a class lookup in a running JVM requires a class loader (in addition to class name). We use the following heuristic for class loader identification: we associate each class with the identity of the first class loaded by its class loader. While a regular AOT method refers to a class loader by the class chain of the 1st class that it loaded, a serialized AOT method refers to it by the *name* of the first class that it loaded. We maintain the 3-way mapping between a class loader and the name and the class chain of the first class that it loaded, in each client JVM. We update this mapping at runtime when the JVM loads or unloads classes and creates or destroys class loaders.

The identification heuristic can fail in edge cases described below, however, that does not affect the correctness of compiled code. While we observed no class loader identification failures in our evaluation, they are still possible in rare cases.

Assume that in the compilation environment, RAMClass `C1` is the first one loaded by `L1`. Cached methods that refer to `L1` identify it by the name of `C` - the ROMClass that `C1` is based on. In the load environment, RAMClass `C2'` is the first one loaded by `L2'` - a different class loader, and then later RAMClass `C1'` is the first one loaded by `L1'` (the correct class loader matching the compilation environment) from the same ROMClass `C`. As a result, loading a cached method can result in using a wrong RAMClass (one loaded by the incorrectly guessed `L2'`). However, RAMClass pointers are only used directly in generated code in *guards* as described above. A mismatching RAMClass pointer can only affect performance (execution will take the slow path of making a virtual call) while correctness is preserved. In other types of relocations, RAMClass addresses are not present in the native code, and are only used to verify class chains and locate RAMMethods, which are shared by both RAMClasses. Another possibility is that `L1'` might load a different class first, and in this case cached methods that refer to it simply will not be loaded.

JVM Environment Compatibility Each AOT method is implicitly associated with an *AOT header* - a data structure that describes the compilation environment: CPU features, JVM configuration, etc. Local SCC stores a single instance of this structure. All JVM instances that store or load AOT code in the SCC must have a matching configuration. Serialized AOT methods store the AOT header of the JVM that it was originally compiled for. To serve a client compilation request from the cache, the server looks up a serialized method with a compatible AOT header.

Storage and Transfer Optimizations In order to make the serialized AOT method representation more compact for optimal storage and transfer, we store serialization records at the server separately from method bodies, and serialized AOT methods refer to them by unique IDs. When responding to a client compilation request with a cached serialized AOT method, the server sends the serialized method body along with all the serialization records it refers to that the client has not yet received. In order to reduce network traffic and deserialization overhead, the client caches serialization records received from the server, and the server keeps track of the record IDs that are already cached at the client.

The size of a populated JITServer cache of compiled methods is ~30-130 MB in our experiments depending on the application, which is smaller than the pre-populated local SCC (~65-170MB) since the JITServer cache only stores ROMClass hashes instead of full class metadata. Since the set of compiled method varies across multiple clients (even running the same application), the JITServer cache accumulates a larger number of methods compared to the local SCC which is only populated once. The cache size can be reduced by pruning the "tail" of less popular methods, e.g. ones that do not get reused within a certain time period. In our experiments, 7-12% of cached methods were never reused.

End-to-End Example We provide a simple example that illustrates how a compiled method body is serialized by the JITServer and later deserialized and loaded by a client JVM. Consider the following Java code:

```
abstract class A {
  abstract void m1();
}
class B extends A {
  void m1() { ... }
}
```

```
class C {
  static void m2(A o) {
    o.m1(); // inlined
  }         // as B.m1()
}
```

Assume that the JIT has inlined the call to `o.m1()` in `C.m2()` as a devirtualized call to `B.m1()` as profiling showed that the runtime class of `o` is normally `B`. The inlined body of `B.m1()` (see pseudo-assembly below) is preceded by a *guard* that checks that the RAMClass of `o` is indeed `B`, otherwise it jumps to the slow path that makes a virtual call.

```
   cmp rax, ramclass_B ; rax contains RAMClass of o
   jne slow_path       ; ramclass_B is hard-coded
   ...                 ; inlined body of B.m1()
.slow_path: ...        ; virtual call to o.m1()
```

Figure 2 illustrates the entities described below. The metadata of the dynamically AOT-compiled method `C.m2()` contains a relocation record for class `B` that will be used to patch the RAMClass address in the comparison instruction in the guard when the method is loaded in another JVM. This record stores the SCC offset of the class chain for `B`, and the SCC offset of the class chain identifying its class loader `L`. This identifying class chain is the one for the first class that was loaded by `L`. Assume that this class was `Object`, i.e. `L` is the bootstrap class loader. The class chain for `B` is a list of SCC offsets of ROMClasses `B, A, Object`. The class chain identifying `L` is a single SCC offset of ROMClass `Object`. The SCC offsets are only valid for the client JVM that originally requested this compilation.

To serialize `C.m()`, the server creates the following records corresponding to the relocation record for class `B`:

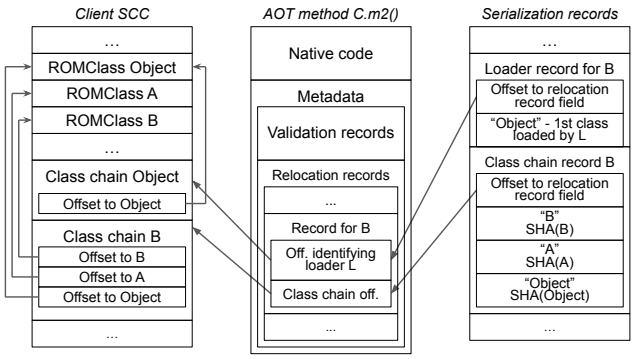

Figure 2: Serialization records for method `C.m2()`

- Class loader serialization record identifying `L` by the name of the first class it loaded - `"Object"`.
- Class chain serialization record for `B`.
- Class serialization records for `B, A, Object`. Each record contains the hash of the ROMClass and its name.

When a different client JVM receives this serialized method, it performs deserialization as follows:

1. Find the class loader `L'` as the one that had a class with the name `"Object"` as the first class that it loaded. In this case it will be the bootstrap class loader.
2. Lookup RAMClass `B'` by name `"B"` in the class loader `L'`.
3. For each class in the class chain (i.e. `B'`, `A'`, `Object`), compute the hash of the ROMClass and compare it with the hash stored in the corresponding serialization record.
4. Update the values of the SCC offset fields in the validation record. The class chain offset will now point to the class chain for `B'` stored in the local SCC, and the loader identifying offset will point to the class chain for `Object`.

Assuming no failures, the method is now deserialized - all its validation and relocation records point to valid SCC entities - and can be stored in the local SCC for future reuse. The client then relocates the method body by patching the RAMClass address in the comparison instruction in the inlining guard so that it points to the RAMClass `B'`, and installs it in the JVM code cache. The compiled native code of `C.m2()` can now be executed correctly in this JVM.

4 Evaluation

Our evaluation answers the following research questions:

- Is remote JIT compilation efficient without caching?
- How does remote JIT compilation affect start-up and warm-up times and memory footprint of application instances with different CPU and memory constraints?
- What is the effect of caching on the *overall cluster-wide* resource usage and application density?
- Can remote JIT improve performance if the client JVM already has a pre-populated local SCC (see Section 2.2)?
- What is the effect of remote JIT and caching on the latencies of compilation requests, compared to local JIT?

Name	Framework	Database	Methods compiled	
			Start	Total
AcmeAir	Open Liberty	MongoDB	~3,000	~11,000
DayTrader		DB2	~5,000	~25,000
PetClinic	Spring	H2 (in-memory)	~5,000	~6,000

Table 1: Application benchmarks

Type	CPU	Memory	Storage
A	16-core AMD EPYC 7302P	256 GB	2× NVMe RAID0
B	14-core Intel Xeon E5-2680	128 GB	SSD

Table 2: Hardware configuration

- How does caching affect the scalability of JITServer (i.e. how many clients can it effectively serve at the same time)?
- Does caching allow JITServer to handle higher latency?

Applications We used the following 3 applications in our experiments: AcmeAir [1] - an airline booking system, DayTrader [8] - a stock trading platform, and PetClinic [15] - an animal hospital information system (the de-facto main benchmark for the popular Spring framework). All 3 are web applications; information about them is summarized in Table 1. We used Apache JMeter to generate the workload for all the applications. These applications are multi-tier, end-to-end benchmarks that are more representative of cloud workloads than benchmarks like SPECjvm2008 [14], SPECjbb2015 [12], and SPECjEnterprise2018 [13] typically used for JVM performance evaluation. Individual benchmarks in the SPECjvm suite are essentially microbenchmarks from the JIT perspective with a small number of JIT-compiled methods. SPECjbb and SPECjEnterprise are heavy, long-running (over 2 hours) workloads unsuitable for analyzing cold start performance. Moreover, DayTrader is a comprehensive JavaEE benchmark that uses most of the same technologies as SPECjEnterprise.

Experimental Setup We ran the experiments on a cluster of 11 machines as described in Table 2: 8 machines of type A and 3 slightly less powerful machines of type B. All machines run Ubuntu 18.04 and are connected with a 10 GBit/s Ethernet network (used in all experiments unless specified otherwise) and a 100 GBit/s Infiniband network.

Type A machines run the instances of the application, the database, and the JITServer, while type B machines run JMeter instances (one per application instance) that generate the load. We use a single JITServer instance running on a dedicated machine in all experiments. While co-locating JITServer instances with application JVMs is a viable deployment option, we evaluate JITServer in the fully remote setting to show "worst case" performance. Application instances run in Docker containers with 1 CPU and 1 GB of memory, unless specified otherwise. This container size is roughly equivalent to an AWS EC2 t2.micro instance which is commonly used for modern cloud workloads [3]. The number of JMeter threads is chosen to saturate the throughput of the application instance. The number of database instances is chosen for each benchmark such that the DB is not a bottleneck. The JVMs are configured to use the default heap size and GC policy. Each

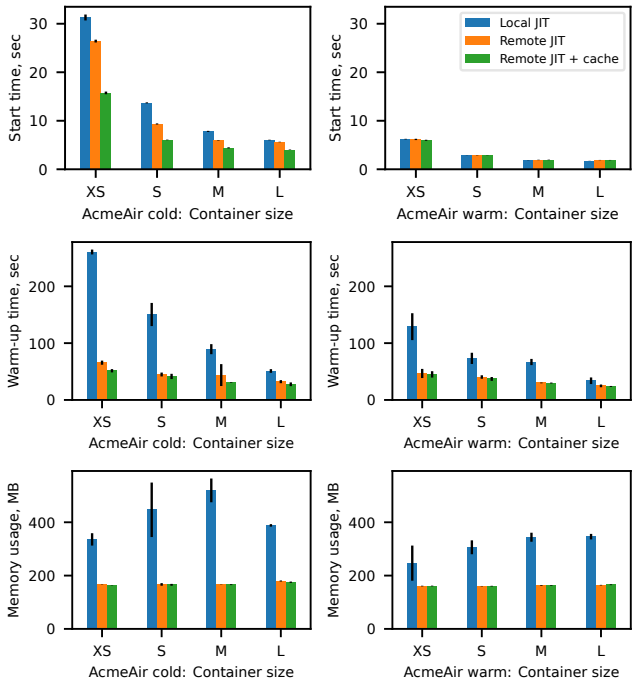

Figure 3: AcmeAir performance: (a) cold; (b) warm

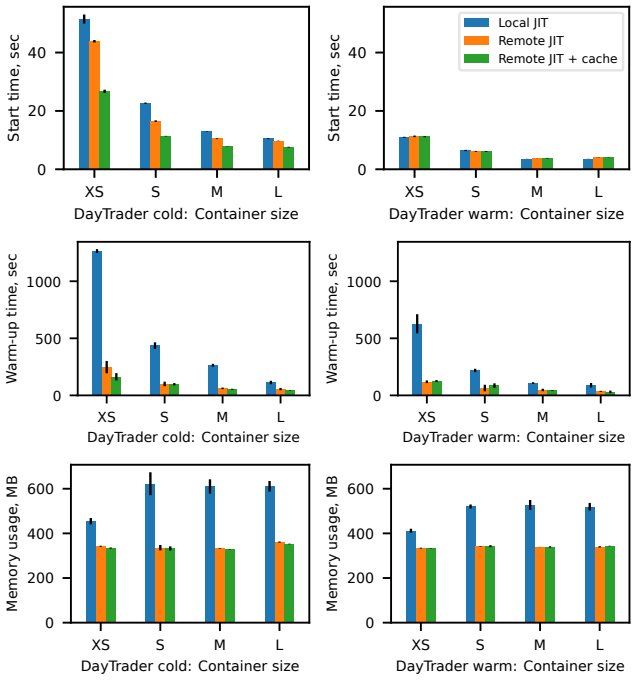

Figure 4: DayTrader performance: (a) cold; (b) warm

data point is averaged over 5 runs, unless specified otherwise; the error bars on the graphs represent standard deviation.

We compare application performance in three JIT compilation modes: *local*; *remote* without caching, and *remote with caching*. Depending on the experiment, we deploy application instances without a pre-populated local SCC (*cold runs* - default unless specified otherwise), or with an SCC populated by only starting an application instance (*warm-start runs*), or by also applying load to it (*warm runs*). While JITServer supports sharing the cache of compiled code across applications, in this paper we focus on the very common use case of multiple instances of the same application. We do not compare with static AOT compilation in HotSpot and GraalVM. Such direct comparison would not be fair since OpenJ9 is a different JVM, and static AOT only supports a subset of Java. We are unable to compare against the remote JIT compiler in [26] since its implementation has not been made available.

4.1 Application Performance and Footprint

We measure how remote JIT compilation (with and without caching) affects the start-up and warm-up performance and memory footprint of a JVM instance. We run the applications in Docker containers of the following sizes: *XS* (0.5 CPU, 512 MB of memory); *S* (1 CPU, 1 GB); *M* (2 CPUs, 2 GB); and *L* (4 CPUs, 4 GB). When JITServer cache is enabled, it is populated by a single run of the application. We define the performance metrics as follows:

- *Memory usage* - peak resident set size (RSS) of the JVM.
- *Start time* - time since the start of the JVM process until the application is ready to handle client requests.
- *Warm-up time* - time from the moment the load is applied until the application reaches 90% of its peak throughput.

Figures 3(a), 4(a), and 5(a) show results for *cold* runs of AcmeAir, DayTrader, and PetClinic respectively. Remote JIT without caching reduces memory footprint by up to 68%, start time by up to 40%, and warm-up time by up to 80% compared to local JIT. The addition of caching reduces start and warm-up times even further (especially for smaller container sizes since they have less CPU available for local JIT): by up to 58% and 87% for start and warm-up respectively. There is still a non-negligible number of compilations (including heavy recompilations at high optimization levels) during warm-up that are not served from the JITServer cache: 24-32% depending on the application, compared to only 7-15% during the start phase. As a result, the effect of caching on warm-up time is relatively smaller compared to start time.

Figures 3(b), 4(b), and 5(b) present results for *warm* runs where the local SCC is pre-populated by a previous full run of the application. JITServer still significantly reduces warm-up time (up to 79%) and peak memory footprint (up to 61%), but has no effect on start time since almost all methods compiled during start-up are stored in the SCC and thus do not need to be compiled in a warm run.

Remote JIT with caching is more effective than only using a pre-populated SCC for reducing memory usage and warm-up time. Moreover, these improvements come "for free" as JITServer caching is transparent for the application developer, unlike using the SCC (see section 2.2). Besides, these results represent the advanced and most optimal way to use the SCC which requires very significant developer effort and is rarely

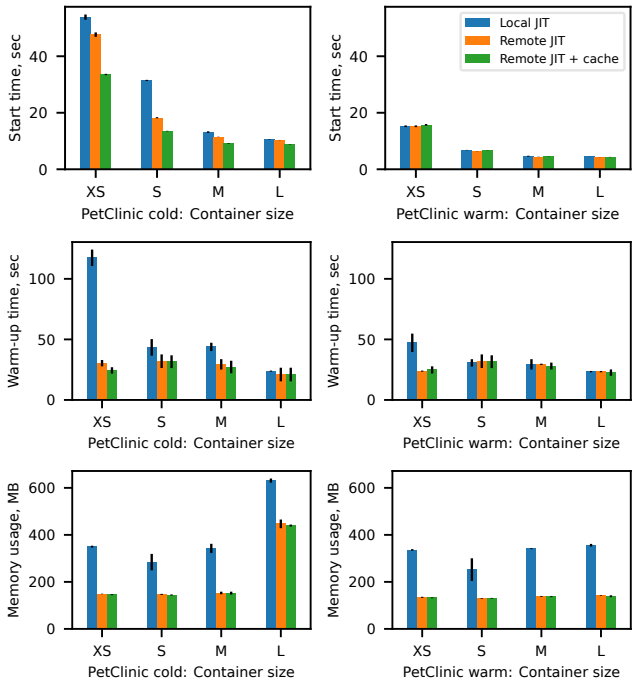

Figure 5: PetClinic performance: (a) cold; (b) warm

done in practice. Caching (local or remote) does not eliminate the need for a JIT compiler. Remote JIT compilation is still a win even with a pre-populated SCC as not all compilations can be cached, and JITServer can make better use of resources, reducing CPU contention and memory footprint.

Application instances reach equivalent peak throughput with local and remote JIT in all the experiments, which is expected since JITServer uses the same set of optimizations and heuristics as the local JIT compiler. The reduction in start and warm-up times is due to the higher degree of parallelism of JIT compilations since the server runs on an additional machine. Arguably, these results do not represent a fair comparison to local JIT as the extra CPU and memory resources used by the JITServer could be used to run other application JVMs instead. In the next subsection we consider *overall system-wide* resource usage, including the JITServer itself.

4.2 Overall System Efficiency

We evaluate the effect of remote JIT compilation (with and without caching) on the *system-wide* resource usage by emulating a cloud deployment where many application instances are brought up and down over a 1 hour period. 64 application slots (one per CPU) are spread evenly across 4 machines. Each slot is used to run a sequence of application instances that execute for a fixed duration (2, 5, or 10 minutes) and stop to be replaced by the next instance. The starting moments of the sequences are staggered with a 10 second interval. We run a single JITServer instance on a separate machine in remote JIT experiments. Each experiment is repeated 3 times.

Figure 6 shows total CPU cost and memory usage (*lower is better*) in *cold* runs. We define *CPU cost* (measured in milliseconds per request) as the amount of total CPU time used by all JVM instances and the JITServer, divided by the number of JMeter requests served (i.e. useful work done) by all application instances. We use *total* CPU time instead of e.g. aggregate throughput to account for the fact that the JITServer runs on an additional machine that could be used to run more application instances. *Total memory usage* includes the peak RSS of the the JITServer and all concurrent JVM instances.

Remote compilation without caching results in up to 21% *increase* in CPU cost compared to local JIT. On the other hand, caching compiled code at the server effectively amortizes the CPU cost over many clients and results in up to 77% reduction compared to local JIT. The effect is larger for shorter application lifespans since the start-up and warm-up phases with high JIT activity take a bigger portion of the run time. We expect application run times in the cloud to be on the lower end. The CPU cost improvements for PetClinic are relatively smaller since it has fewer methods compiled during warm-up (see Table 1). Remote JIT with caching also significantly reduced start times in these experiments - by 32-58%.

Remote JIT delivers benefits in terms of memory footprint: it reduces total memory usage by up to 62% compared to local JIT compilation. Peak total memory footprint is smaller with JITServer because memory usage spikes caused by heavy compilations from multiple clients are unlikely to align at the server. The 2-minute DayTrader runs are the exception; the extra memory footprint is due to faster heap expansion caused by higher allocation rates since the application reaches higher throughput compared to other JIT compilation modes.

The results for *warm-start* runs (with a pre-populated SCC) are shown in Figure 7. Remote JIT without caching can still have higher CPU cost (e.g. by 9% for 2-minute PetClinic runs) than local JIT with SCC, while JITServer with caching matches or surpasses the performance of local JIT with SCC. Remote JIT still achieves lower (by up to 45%) total memory usage. JITServer not only achieves better resource utilization than using a pre-populated local SCC, but does it transparently without the additional developer effort associated with managing the SCC (see Section 2.2). The two approaches can be combined, in which case adding JITServer reduces CPU cost by up to 53% compared to local JIT.

The main implication of these results for cloud computing is that remote JIT with caching allows to increase application density, even after accounting for the resources used by the JITServer. We can fit more application instances into the same amount of hardware resources since each instance requires less memory and uses the CPU cycles to do more useful computation. With caching, JITServer does not simply move the overhead around - it uses the resources more efficiently.

4.3 Compilation Request Latencies

We measure two compilation latency metrics: (i) *compilation time* taken to serve the request either locally or remotely, and

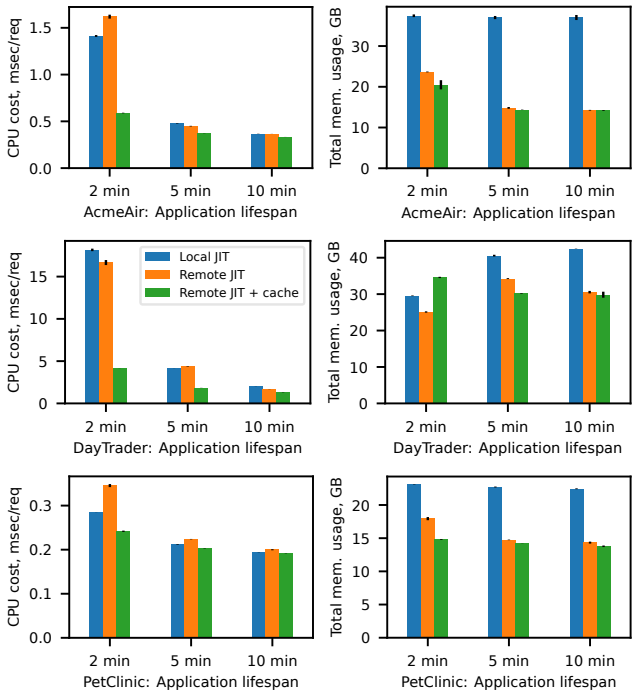

Figure 6: Overall efficiency in cold runs *(lower is better)*

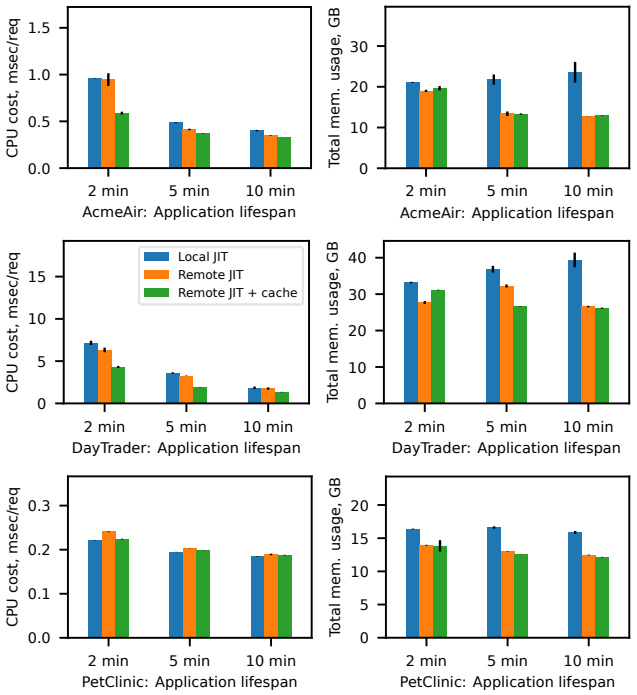

Figure 7: Overall efficiency in warm-start runs *(lower is better)*

(ii) *total queuing time* - from the moment the compilation is first scheduled by the JVM (e.g. the method reached its invocation threshold) until its completion. Figure 8 shows the resulting CDF (using a logarithmic scale on the X axis) for the AcmeAir benchmark when the JITServer is given all the resources of its host. Figure 9 represents the configuration with *equal CPU resources*: 2 CPUs for local JIT, and 1 CPU each for the JITServer and the client JVM. The results for other benchmarks are very similar and thus omitted.

The results show that individual remote compilations take longer than the local ones, unless the JITServer has more CPU resources. In the latter case, longer (more CPU-intensive) compilations take less time at the JITServer thanks to ample CPU resources, while cheaper compilations take longer than locally - their latency is dominated by communication. Total queuing times are still shorter than with local JIT even with limited JITServer CPU due to increased parallelism: the CPU work is overlayed with waiting for the network. Caching compiled code at the JITServer dramatically reduces compilation request latencies, thanks to the 68-76% cache hit rate.

4.4 Scalability

In order to determine how caching affects JITServer performance with an increasing number of clients, we run a variable number - between 1 and 64 (80 for PetClinic) - of application instances that use the same JITServer instance. Application JVMs start simultaneously and without a pre-populated local SCC, in order to maximize the load on the server. We measure the *full warm-up time* (sum of start and warm-up times) averaged over all concurrent JVM instances. Remote JIT is more efficient than local for a given number of clients only if the full warm-up time is lower. When the server becomes overloaded, the clients accumulate a backlog of compilation requests, which results in slower warm-up.

Figure 10 shows the full warm-up times for each number of instances (normalized to the local JIT configuration). We see that JITServer caching dramatically improves scalability: given the same amount of resources available to the server instance, it can sustain load from a larger number of clients, while improving their start-up and warm-up performance. The cost of JIT compilation is reduced and effectively amortized over a large number of concurrent clients by avoiding the majority of repeated compilations with caching.

These results represent JITServer performance in a worst case scenario. In a real deployment, we expect client JVM starts to be staggered, providing more opportunities for statistical multiplexing, which in turn should allow JITServer to handle an even larger number of clients without saturating. Additional JITServer instances can be brought up on demand, allowing linear horizontal scaling since JITServer only maintains soft state - the cache of compiled methods.

4.5 Effect of Network Latency

To determine how caching affects JITServer performance with increasing network latency, we run a single application instance with varying values of round-trip network latency between the client JVM and the JITServer. We measure the *full warm-up time* (start + warm-up) and compare it with using local JIT compilation: remote compilation improves performance if it results in faster warm-up. As latency grows,

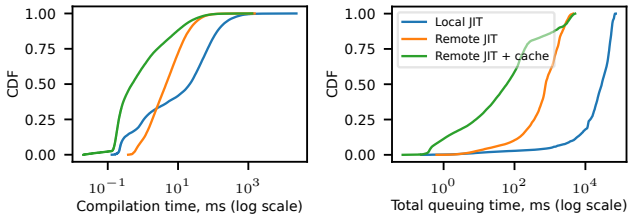

Figure 8: Compilation latencies: unlimited JITServer resources

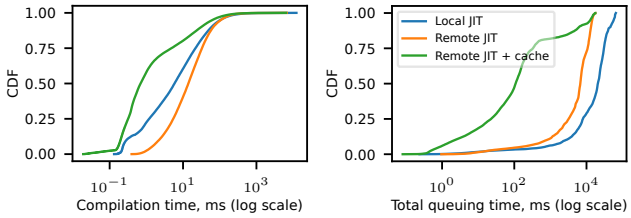

Figure 9: Compilation latencies: equal JIT CPU resources

each compilation request takes more time, and the server eventually becomes unable to compile methods faster than the local JIT compiler. We used the 100 Gbit/s Infiniband network for the smallest latency value - 15 microseconds. The 10 Gbit/s Ethernet network has a latency of 45 microseconds. We emulate the additional latency for subsequent data points using the `netem` module in the Linux kernel.

Figure 11 shows the results for latencies up to over 8 milliseconds. We can see that caching allows JITServer to tolerate higher network latencies (~4-8 ms, compared to ~2-4 ms without caching) since cache hits only incur a single round-trip. Caching compiled code reduces the average number of messages per compiled method by ~54% (from ~42-47 to ~19-22 depending on the application), and the total amount of data transferred per client by ~30% (from ~190-800 MB to ~140-580 MB depending on the application). On the other end of the spectrum, bringing the latency down to microseconds only slightly improves performance. JITServer performs very well for latencies in the hundreds of microseconds which are typical in cloud datacenters. Performance in high-latency scenarios can be further improved by increasing the maximum number of client compilation threads (see Section 3.1). In our future work, we plan to investigate using remote JIT compilation in high-latency environments such as edge computing.

Summary of Results Our evaluation shows that:

- Remote JIT cannot be fully efficient without caching - it often increases total system-wide CPU cost (by up to 21%).
- Caching compiled code allows JITServer to reduce cluster-wide resource usage (by up to 77% for CPU and up to 62% for memory) and increase application density.
- JITServer significantly reduces application start-up and warm-up times (by up to 58% and 87% respectively) and memory footprint, especially in smaller containers.
- Caching dramatically improves JITServer scalability and allows it to effectively handle more concurrent clients and tolerate significantly higher network latency (up to 8 ms).

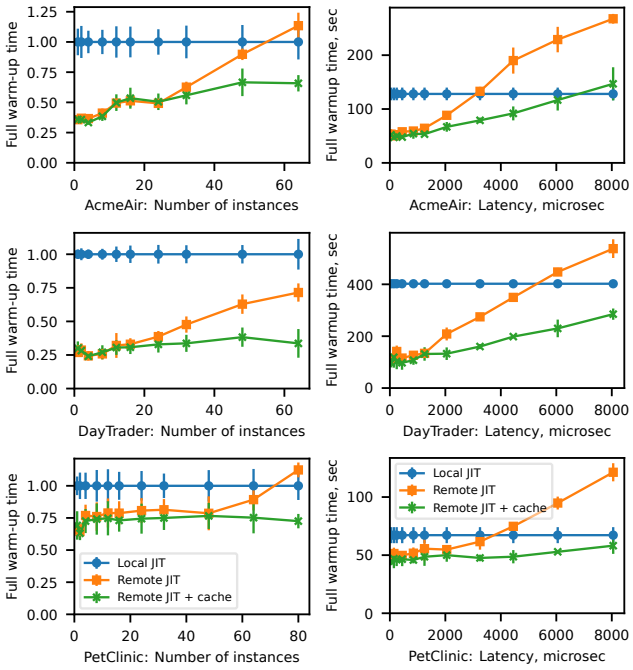

Figure 10: Scalability *(lower is better)*

Figure 11: Effect of network latency

5 Conclusion and Future Work

In this paper we explored JIT compiler disaggregation as a means to improve the performance and memory utilization of JVMs running in the cloud. We described JITServer, our disaggregated JIT compiler implementation in a production grade JVM, which supports the caching and reuse of compiled code across JVMs running on different hosts, and effectively amortizes JIT compilation costs over many client JVMs.

The experimental results showed excellent improvements in start-up time, warm-up time and memory footprint: JITServer is able to speed-up JVM start-up by as much as 58% and to reduce warm-up times by up to 87%, without a degradation in peak throughput. Moreover, the overall system-wide peak memory footprint is reduced by up to 62%, which should make it possible to reduce operational costs by increasing application density. Caching and reusing compiled code allows JITServer to reduce overall CPU cost by up to 77%, showing that JITServer enables cloud users to do more useful computation with less resources.

Our future work will explore prefetching of server-cached code and predicting hot methods 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. We will investigate ways to improve the JIT compiler in the disaggregated setting, e.g. utilizing profiling data from many clients, efficiently sharing compiled code across applications, and automatically sizing and scaling compilation resources. We plan to apply JITServer to other workloads such as FaaS, microservices, and data analytics.

A Artifact Appendix

Abstract

We provide the open source implementation of JITServer (as part of the OpenJ9 project), as well as the automated benchmarking platform that we used in our experimental evaluation. Our results have been successfully reproduced in artifact evaluation.

Scope

Our artifact can be used to run all the experiments described in Section 4 in order to validate the main claims in our paper, most importantly:

- JITServer can reduce application start time and warm-up time and system-wide CPU and memory usage for JVM-based applications running in containers with limited resources (which are common in the cloud).
- Caching dynamically compiled code at the JITServer is necessary to fully achieve the reduction in overall CPU usage and application start time.

Note that the experimental results are expected to be slightly different from the ones reported in this paper (even on the same hardware) since the OpenJ9 implementation has evolved since we conducted those experiments. However, the main conclusions should still hold.

Contents

The artifact consists of two parts:

1. The open source implementation of our system, which has been contributed to the OpenJ9 project. JITServer is implemented in ~25 KLOC of C++. The code is integrated into the rest of the OpenJ9 code base.
2. A set of scripts (Python, shell scripts, and Docker files) that automate the benchmark runs used in our evaluation and generate the resulting graphs. We also provide the logs generated by running the full set of experiments reported in this paper.

Hosting

All the parts of our artifact are open source and are hosted on GitHub. The OpenJ9 code base (including JITServer) is split across three separate repositories:

- Eclipse OpenJ9: https://github.com/eclipse-openj9/openj9;
- Eclipse OMR: https://github.com/eclipse/omr;
- OpenJDK extensions for OpenJ9 (we used JDK 8 in our experiments; newer JDK versions are also available): https://github.com/ibmruntimes/openj9-openjdk-jdk8.

The source code of our automated benchmarking platform is available at https://github.com/AlexeyKhrabrov/jitserver-benchmarks

Stable forks or branches of these repositories based on the 0.32.0 OpenJ9 release (with minor changes required by our benchmarking platform, e.g. collecting additional statistics) that were used for artifact evaluation are available as follows:

- OpenJ9: https://github.com/AlexeyKhrabrov/openj9/tree/atc22ae (commit `724db2932e5f0abb`);
- OMR: https://github.com/AlexeyKhrabrov/omr/tree/atc22ae (commit `ab24b66665961405`);
- JDK: https://github.com/AlexeyKhrabrov/openj9-openjdk-jdk8/tree/atc22ae (commit `0b8b8af39a5f1f2f`);
- Benchmarks: https://github.com/AlexeyKhrabrov/jitserver-benchmarks/tree/atc22ae (commit `8360d90b89744ad1`)

Requirements

While JITServer supports a wide range of Linux platforms, our benchmarking setup assumes Ubuntu 18.04 as the OS. It should be possible (although not necessarily easy) to tweak it to work on other Linux distributions. Newer Ubuntu versions might require downgrading to an older GCC version (OpenJ9 currently officially supports GCC 7, but should also support GCC 10). Different Linux distributions will need more tweaks, namely different ways of installing prerequisite packages.

Running the largest experiments reported in this paper requires a cluster of 11 machines with 16 CPU cores each, connected with a 10 Gbit/s network (or at least 1 Gbit/s) with round-trip latency between machines in the low hundreds of microseconds or less. Alternatively, the experiments can be run in a public cloud such as AWS on a cluster of virtual instances with roughly equivalent resources. The benchmark setup requires `sudo` permissions on all the machines. The required amount of storage space is approximately 30 GB on each node (not including the OS).

The hardware and software environment that we used in the experimental setup in our evaluation is described briefly in Section 4 and in more detail in the README document: https://github.com/AlexeyKhrabrov/jitserver-benchmarks/tree/atc22ae#environment-used-in-our-evaluation.

Usage

Detailed instructions describing how to set up and run the benchmarks and generate the results can be found in the README of the artifact repository: https://github.com/AlexeyKhrabrov/jitserver-benchmarks

Acknowledgements

We thank the anonymous reviewers and the shepherd for their feedback that helped us improve the paper, as well as the anonymous AEC members for their help in identifying and resolving issues in our artifact submission. We are grateful to the OpenJ9 developer community for their help in contributing

our code to the OpenJ9 project. This research was supported in part by IBM CAS Canada and an NSERC CRD grant.

References

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

[2] AdoptOpenJDK OpenJ9 official image - Docker Hub. https://hub.docker.com/_/adoptopenjdk.

[3] Amazon EC2 T2 instances. https://aws.amazon.com/ec2/instance-types/t2/.

[4] Apache OpenWhisk runtimes for Java. https://github.com/apache/openwhisk-runtime-java.

[5] Azul cloud native compiler. https://www.azul.com/products/intelligence-cloud/cloud-native-compiler/.

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

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

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

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

[10] JEP 410: Remove the experimental AOT and JIT compiler. https://openjdk.java.net/jeps/410.

[11] Open Liberty official image - Docker Hub. https://hub.docker.com/_/open-liberty.

[12] SPECjbb2015 benchmark. https://www.spec.org/jbb2015/.

[13] SPECjEnterprise2018 Web Profile benchmark. https://www.spec.org/jEnterprise2018web/.

[14] SPECjvm2008 benchmark. https://www.spec.org/jvm2008/.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.