Towards a Serverless Java Runtime

Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, Sanhong Li
Alibaba Group, China
{lingyue.zyf, tianxiao.gu, yunyao.zxl, lei.yul, kuaiwei.kw, sanhong.lsh}@alibaba-inc.com

***Abstract*—Java virtual machine (JVM) has the well-known slow startup and warmup issues. This is because the JVM needs to dynamically create many runtime data before reaching peak performance, including class metadata, method profile data, and just-in-time (JIT) compiled native code, for each run of even the same application. Many techniques are then proposed to reuse and share these runtime data across different runs. For example, Class Data Sharing (CDS) and Ahead-of-time (AOT) compilation aim to save and share class metadata and compiled native code, respectively. Unfortunately, these techniques are developed independently and cannot leverage the ability of each other well. This paper presents an approach that systematically reuses JVM runtime data to accelerate application startup and warmup. We first propose and implement JWarmup, a technique that can record and reuse JIT compilation data (e.g., compiled methods and their profile data). Then, we feed JIT compilation data to the AOT compiler to perform profile-guided optimization (PGO). We also integrate existing CDS and AOT techniques to further optimize application startup. Evaluation on real-world applications shows that our approach can bring a 41.35% improvement to the application startup. Moreover, our approach can trigger JIT compilation in advance and reduce CPU load at peak time.**

***Index Terms*—Java, Java virtual machine, just-in-time compilation, ahead-of-time compilation, application warmup**

I. Introduction

Java has become one of the most popular programming languages, particularly for large-scale web applications, such as e-commerce, financial, logistics and entertainment. However, Java developers have always been plagued by the problems of *slow startup* and *warmup* of Java applications. This is because Java programs are compiled into bytecode and run in the Java virtual machine (JVM). We use *startup* to refer to the phase between the start of the JVM and the response of the first user request. We use *warmup* to refer to the phase between the start of the JVM and the peak performance.

Slow Startup. The slow startup is caused by class loading and application initialization. JVM loads each visited Java class dynamically to create a runtime metadata object [1] during startup. Large-scale Java applications usually have tens of thousands of Java classes or even more, which are packaged into thousands of JAR files. In practice, it can take *minutes* for those applications to start up before they are online and begin to process user requests.

Warmup. JVM starts the execution in the interpreter first and then gradually just-in-time (JIT) compiles hot methods into native code with optimizations guided by profile data [2]. However, JVM is not expected to have high performance at the beginning. Applications need carefully designed mocked or even real user requests to exercise critical execution paths to warmup the JVM.

Consequently, the above two limitations bring significant challenges to agile software development, testing and deployment, especially when thousands of application instances need to be deployed to the data center. Furthermore, these two issues are also obstacles for building modern serverless and cloud native software [3] using Java, where applications are run in containers and are managed by container orchestration software to achieve extreme scalability and automation.

We will have more opportunities to optimize application startup and warmup if the JVM could have *prior knowledge* of the applications. To this end, we adopt the **"recording and replaying"** methodology to deploy applications. In Alibaba, we first deploy an application at a small scale in the *staging environment* (i.e., a near replica of a production environment for testing) to collect runtime properties about the application. This deployment is called *beta release*. If the application runs without any issue in the staging environment, we then deploy the application at the full scale in the production environment. This deployment is called *full release*. In the full release, JVM could leverage those profiling data collected in the beta release and perform pertinent optimizations earlier.

To this end, we develop Atlas and JWarmup to accelerate Java application startup and warmup, respectively, where both of them are developed upon Alibaba Dragonwell [4], a downstream version of OpenJDK maintained by Alibaba. Atlas and JWarmup systematically record various runtime data of the execution of a Java application and reuse those data to speed up application startup and warmup in subsequent executions of the same application.

We evaluate the effectiveness of Atlas and JWarmup against benchmarks and real-world Java applications. Results show that Atlas reduces the startup time of a real-world application from 5.2s to 3.0s, with a 41.35% improvement. Meanwhile, JWarmup can trigger JIT compilation in advance and reduce CPU load at peak time.

II. Background

In this paper, we use JVM to refer to the HotSpot JVM in OpenJDK, which is upstream of many JVM vendors including Alibaba Dragonwell.

A. Lifecycle of Code Execution in JVM

Figure 1 illustrates the lifecycle of code execution in JVM.

Class Loading and Linking. The containing class of a Java method or field must be loaded into JVM before the method

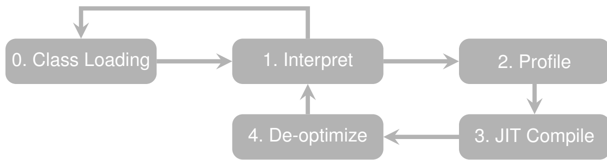

Fig. 1. The lifecycle of code execution on the JVM.

or field is accessed. During class loading, JVM locates the binary representation of the class with a particular name and creates a class metadata object in JVM [5]. The class metadata object represents the runtime type of the class.

Interpreter. Method bytecode is first executed by the interpreter. Interpretation is the cornerstone of the subsequent optimized execution. For example, JIT compiled code does not check whether a class is loaded. All classes are loaded during interpretation.

Profiling. The JIT compiler cannot perform sophisticated code analysis and code optimization when JVM does not have a complete picture of a Java method. JVM thus accumulates method execution counters and collects various method profiling data during interpretation.

Just-in-time Compilation. JVM submits hot methods to the JIT compiler to generate efficient native code. The JIT compiler leverages method profiling data to do aggressive code optimizations, e.g., inlining monomorphic virtual calls [6].

De-optimization. JIT can do speculative optimizations according to specific assumptions at JIT compile time. Later, some assumptions are broke due to changes to the runtime, e.g., new classes loaded. The compiled code then cannot be executed anymore. JVM leverages the de-optimization mechanism to dispose of the compiled code and fallback to the interpreter to continue the execution. JVM re-profiles the de-optimized method and may re-compile the method once it becomes hot again.

B. Our Understanding of Startup and Warmup

JVM loads each class on-demand during startup. The application starts to execute and perform necessary initialization. Consequently, *accelerating class loading and improving the performance of initialization code* during startup are two possible solutions for accelerating application startup.

The essentials to speed up warmup is to *identify and compile hot methods to native code as soon as possible*. In real-world scenarios, a server application can be scheduled to handle user requests immediately after it has started up. In practice, enterprise Java applications launch thousands of threads to process user requests, but JVM creates much fewer compiler threads to perform JIT compilation. There is inevitable resource contention between JIT threads and application threads during warmup. Therefore, hot methods cannot be compiled in time and are interpreted longer than expected. As a result, the designed critical performance metrics, such as response time and CPU load, cannot be met, which threatens the availability and stability of online services.

III. Approach

In this section, we elaborate on the technique details.

A. Recording

Figure 2 depicts the workflow of recording. Java applications are deployed in a staging environment to perform JWarmup recording. Currently, JWarmup records *touched methods* (i.e., methods that are executed) and various data used by JIT compilation.

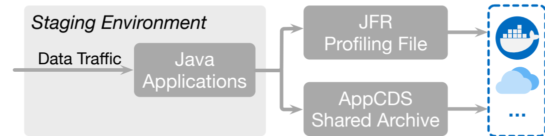

Fig. 2. The workflow of recording.

We leverage Java Flight Recorder (JFR) [7] to record data for JWarmup and store all data into a *JFR profiling file*. JFR is a tool for collecting diagnostic and profiling data about a running Java application with negligible performance overhead. We also use *Application Class-Data Sharing (AppCDS)* [8] to record used classes into a shared archive file, which can be used to accelerate class loading during replaying. Both the JFR profiling file and AppCDS shared archive are archived into the container image of the application or uploaded to the cloud storage. Hence, applications deployed in the production environment can use them for replaying.

One may note that JVM provides a flag `-Xcomp` to JIT compile each method eagerly at the first execution. However, due to unloaded classes and immature profiling data, such JIT compiled methods are easily de-optimized and fall back to interpretation soon.

To reduce unnecessary de-optimizations, JWarmup needs to make a wise decision on when a method can be committed to JIT compilation before profiling data are mature. If too late, JWarmup fails to warmup the application in advance. If too early, JIT compiled methods are frequently de-optimized due to unloaded classes and immature profiling data. Our insight is using *method dependencies* to represent the prerequisite for compiling a method at runtime.

JWarmup identifies three categories of dependencies:

1) *Type dependency* is introduced by bytecode instructions (e.g., `new`, `invokevirtual` and `putfield`) that rely on type references in the JVM constant pool of the method [9].
2) *Object dependency* is introduced by bytecode instructions (e.g., `invokedynamic`) that rely on runtime objects in the JVM constant pool of the method [9].
3) *Profiling dependency* is introduced by bytecode instructions that rely on profile data [6]. The profile data could be a runtime type that must be loaded into the JVM but is not necessarily in the JVM constant pool of the method.

JWarmup also records the inline tree of a compilation task. This is because JIT compilers rely on various runtime information together to guide the compilation of a method.

That is, the compilation of a given method not only depends on bytecode and profile data of itself but also various other information in JVM. Inlining is one of such optimizations.

B. Speeding up Application Startup

The application can be deployed in the production environment after it has been tested thoroughly in the staging environment. We then propose Atlas to speed up the application startup by using data recorded in the staging environment. Figure 3 shows the workflow of Atlas. Atlas uses three techniques, i.e., ahead-of-time (AOT) compilation, profile-guided optimization (PGO) and AppCDS.

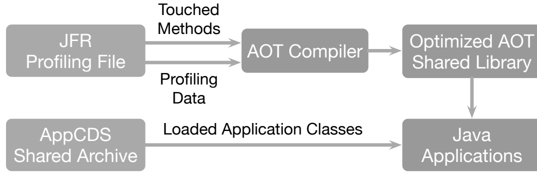

Fig. 3. Workflow of Atlas during replaying.

First, Atlas invokes `jaotc`, i.e., the AOT compiler [10], to compile Java classes to native code prior to launching JVM. If all the application code and Java libraries are AOT compiled, we will obtain quite large shared libraries, which is unnecessary and suffers overhead during code loading and linking. Therefore, we modify the AOT compiler to accept the JFR file generated by JWarmup and parse touched methods from it. The AOT compiler only needs to compile reachable Java methods.

Second, the AOT compiler cannot perform aggressive code optimizations since it does not have any profiling data. Similarly, we modify the AOT compiler to parse profiling data from the JFR file to perform profile-guided optimization (PGO). More details of PGO are discussed in Section III-D.

JVM can load the AOT shared library and execute highly optimized native code directly and bypass the interpreting execution. Moreover, Atlas leverages AppCDS to reduce the application startup time. We also developed a technique called *class pre-resolution* (see Section III-E) to further reduce class loading cost in the AOT code. With pre-resolution and AppCDS, Atlas can significantly reduce application startup time, since thousands of classes are loaded during startup.

C. Speeding up Warmup

During replaying, JWarmup first parses data of recorded JIT compilation data and listens to dependency resolution and initialization events. JWarmup commits a method for *warmup JIT compilation* once all dependencies of the method are ready, which mostly happens before the counter of the method exceeds the compilation threshold. JWarmup uses parsed data to reconstruct the compilation environment. The reconstructed environment can provide proper guidance for JIT optimizations. For example, the recorded inlining tree is used to guide the inlining decision. With JWarmup, the method can be compiled in advance for early warmup without compromising performance.

Java Code	Bytecode	BranchData
`if (cond) {`	`1: ifeq 11`	`ifeq, BCI: 1`
`o.x();`	`4: Invoking o.x()`	`Taken: 4999`
`} else {`	`8: goto 15`	
`o.y();`	`11: Invoking o.y()`	`Not Taken: 1`
`}`	`15: return`	

Fig. 4. The HotSpot VM uses `BranchData` to record branch taken and not taken frequency for the bytecode `ifeq`.

D. Profile-Guided Optimization

Profiling data allow the JIT compiler to perform adaptive optimizations and significantly improve performance. Since the AOT compiler does not collect profiling data, Atlas leverages method profiling data recorded by JWarmup to perform PGO. Basically, there are two kinds of profiling data: counter-based data and type-based data.

Counter-based data record numbers associated with bytecode instructions. Figure 4 depicts how JVM records branch profiling for the `if` statement, which is translated to instruction `ifeq`. Note that every bytecode instruction has a unique label. Profiling data are shown on the right-hand side of Figure 4. For `ifeq`, JVM records the execution count of the `true` branch (branch taken) and the `false` branch (branch not taken). In this example, the branch taken and not taken count are 4,999 and 1, respectively. Then we can see that the `false` branch is an unlikely one and the JIT compiler can leverage this information to perform adaptive optimization.

Type-based data record runtime types associated with bytecode instructions. Atlas mainly uses type-based data to perform virtual call inlining and generate a fast path for subtype checking.

```
Base b = ...;
b.foo();                                  Java code
-----------------------------------------------------
if (b->_klass == GOT.Child) {
  // method body of Child.foo()
} else {
  // De-optimization
  uncommon_trap();
}                                  Pseudo native code
```

Fig. 5. Example code generated by Atlas for virtual call inlining.

Virtual Call Inlining. Figure 5 illustrates an example of a Java virtual call and the optimized pseudo native code generated by Atlas. In Figure 5, `Child` is the subclass of `Base` and `Child` overrides method `foo()` declared in `Base`. Then `b.foo()` is a virtual call. Assuming that profiling data indicate that `b` in Figure 5 points to an instance of class `Child`. This information allows the AOT compiler to inline `Child.foo()` and generates *type guards* surrounding the inlined method. In Figure 5, `b->_klass` represents the type of `b` and `GOT.Child` refers to the resolved class `Child`. Therefore, if `b._klass` equals to `GOT.Child`, `b` should refer to an instance of `Child`. Then the inlining code (method body of `Child.foo()`) can be safely executed. Otherwise, an uncommon trap occurs. The AOT code is then

```
if (b instanceof Child) {
  ...
}                                    Java code
```

```
if (b->_klass == GOT.Child) {
  // subtype checking passed
} else {
  // De-optimization
  uncommon_trap();
}                           Pseudo native code
```

Fig. 6. Example code generated by Atlas for the fast path of subtype checking.

de-optimized. The virtual call is executed by the interpreter to maintain correctness.

Fast Subtype Checking. Atlas can further optimize subtype checking with profiling data, though the AOT compiler already performs fast subtype checking [2] for `instanceof` and `checkcast` by default. Specifically, if profiling data can tell there is one and only one type for type checking, the AOT compiler can create a fast path for type checking as shown in Figure 6. Assuming `Child` is the only witnessed type. The AOT compiler creates a type comparison, i.e., comparing `b._klass` and `GOT.Child`, as the fast path of subtype checking.

E. Class Pre-Resolution

An AOT shared library uses a global offset table (GOT) to store runtime type identifiers i.e., pointers to class metadata objects. At first, the GOT is empty. Once a class is resolved, the JVM updates the GOT accordingly. AOT code relies on calling a JVM internal function to resolve classes. According to our profiling, this call is quite heavy. If AOT code constantly triggers class resolution, it will introduce performance overhead.

AppCDS offers us an opportunity to optimize class resolution for AOT code. Since the shared archive is memory-mapped, the addresses of class metadata objects are known when the shared archive is generated. After the JVM is fully initialized and about to start the `main()` method, Atlas loads and resolves all the classes from the shared archive and fills the GOT. Consequently, the overhead introduced by repeatedly calling JVM runtime functions for class resolution is eliminated. The application startup time can be further reduced with Atlas.

IV. Evaluation

We used real-world applications to demonstrate the effectiveness of JWarmup and Atlas. Our evaluation answers the following research questions (RQs):

- **RQ1.** Is JWarmup able to commit JIT compilation earlier than the normal execution?
- **RQ2.** Is JWarmup capable of reducing JIT compiler CPU load in the peak time of data traffic?
- **RQ3.** Is Atlas able to improve the performance of the AOT code?
- **RQ4.** Can Atlas reduce application startup time?

A. RQ1. Early Warmup Compilation

We used a real-world e-commerce application of Alibaba to evaluate JWarmup. First, one application instance was deployed to the staging environment for recording. After that, the recorded data file was uploaded to the Alibaba Cloud Object Storage Service (OSS). In the full release, every application instance downloaded the recorded file from the OSS and started replaying. Data traffic was automatically injected once the application finished startup and was discovered by the scheduling system. We used JFR to record the CPU load of the Java process and compiler threads. JFR started recording while the application was launching and sampled CPU load every second for three minutes. Figure 7 illustrates the CPU load of the C2 compiler.

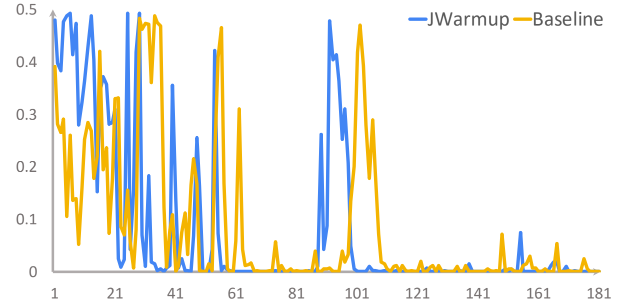

Fig. 7. C2 compiler CPU load when JWarmup is disabled and enabled.

In Figure 7, the horizontal and vertical axes represent time and CPU load, respectively. Line —— and —— show the C2 compiler CPU load when the JWarmup is enabled and disabled, respectively. From Figure 7 we can see that the C2 compiler CPU load peak of JWarmup comes earlier than the baseline, which indicates that JWarmup is able to eagerly trigger JIT compilation for application warmup.

B. RQ2. Reducing Peak Time CPU Load

Figure 8 compares the Java process CPU load when JWarmup is disabled and enabled. Since JWarmup commits hot method JIT compilation in advance, peak time CPU load is reduced when data traffic coming into the system. This advantage assures the stability and availability of online services.

C. RQ3. Improving Code Performance for AOT

Atlas leverages PGO to improve the performance of the AOT code. In this section, we use Java Micro-benchmark Harness (JMH) to evaluate the effectiveness of PGO. We wrote two micro-benchmarks, which can measures the effectiveness of profile-guided virtual call inlining and fast subtype checking, respectively. Table I shows the results of the two benchmarks.

In Table I, virtual call inlining and subtype checking represent scores of two micro-benchmarks. The column Baseline shows scores of AOT code and the column PGO are scores of optimized AOT code by using method profiling data. For virtual call inlining, PGO brings a 39.41% performance

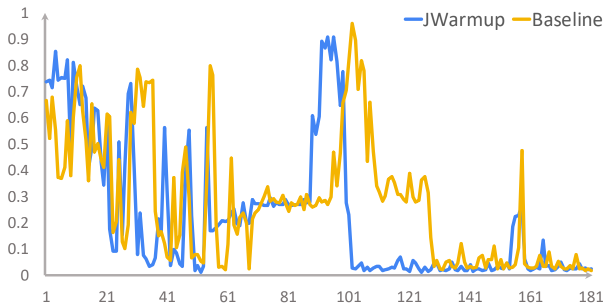

Fig. 8. Java process CPU load when JWarmup is disabled and enabled.

TABLE I
MICRO-BENCHMARK RESULTS WHEN PGO IS DISABLED AND ENABLED.

	Baseline	PGO	Improvement
Virtual call inlining	42,968,335.45	59,901,311.35	39.41%
Subtype checking	38,492,467.53	38,983,705.35	1.28%

improvement, which shows the power of method profiling and inlining. Since profile-guided fast subtype checking only saves some memory operation, PGO brings a 1.2% improvement.

D. RQ4. Speeding up Application Startup

In this section, we used PetClinic, a real-world Spring Boot application, to demonstrate that Atlas is capable of accelerating application startup. Table II shows the startup time of PetClinic under five configurations: (1) Atlas is disabled (baseline), (2) only AppCDS is enabled, (3) only original AOT is enabled, (4) both AppCDS and original AOT are enabled and (5) Atlas is enabled. The time measured for startup time by a particular configuration is the average of 20 runs.

For the baseline, the application startup time is 5.22s. When the AppCDS is enabled, PetClinic started up in 3.84s with a 26.51% improvement. Meanwhile, the original AOT even causes performance degradation. PetClinic started up in 5.25s. Besides, PefClinic startup in 3.82s with a 26.88% improvement when both AppCDS and original AOT are enabled. Finally, Atlas enabled PetClinic to startup in 3.06s with a 41.35% improvement. By combining AppCDS and profile-guided optimized AOT, Atlas can significantly reduce the application startup time.

V. DISCUSSION

ReadyNow! [11], [12] developed by Azul Systems saves JIT compilation information during application running. When replaying, it firstly performs class loading and initialization and JIT compilation according to the profile recorded, then executing user code. Since static initializers usually are not side-effect-free, eagerly triggering class initialization rather than in an on-demand manner may cause unexpected program behaviors. JWarmup commits warmup JIT compilation in a conservative but safe manner, i.e., using method dependencies as the prerequisite of compiling a method.

PGO is an effective and widely used compiler optimization technique [13], [14]. Atlas provides a pragmatic approach to

TABLE II
PETCLINIC STARTUP TIME UNDER FIVE CONFIGURATIONS.

	Baseline	AppCDS	Original AOT	AppCDS + Original AOT	Atlas
Startup Time/s	5.22	3.84	5.25	3.82	3.06
Improvement	N/A	26.51%	-0.47%	26.88%	41.35%

leverage JVM runtime data to reduce application startup time, where AppCDS helps reuse class metadata and method profiling recorded by JWarmup can improve code performance generated by the AOT compiler.

VI. CONCLUSION

Large-scale Java application startup and warmup problems have baffled Java developers all the time. In this paper, we adopt the “recording and replaying” method and present Atlas and JWarmup to mitigate these two problems. Evaluation on real-world applications shows that JWarmup is able to commit hot method compilation in advance and reduce peak time CPU load and Atlas is capable of reducing application startup time. Both JWarmup and Atlas will be open-source in Alibaba Dragonwell in the future. JWarmup and Atlas provide pragmatic solutions in practice for developing modern serverless and cloud native software using Java.

REFERENCES

[1] S. Liang and G. Bracha, “Dynamic Class Loading in the Java Virtual Machine,” in *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 1998)*, 1998, pp. 36–44.
[2] C. Click and J. Rose, “Fast subtype checking in the hotspot jvm,” in *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, 2002, p. 96–107.
[3] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, “Cloud programming simplified: A berkeley view on serverless computing,” *CoRR*, vol. abs/1902.03383, 2019.
[4] Alibaba Group. (2021) Alibaba dragonwell 11. [Online]. Available: https://github.com/alibaba/dragonwell11
[5] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Oracle America, Inc, 2021.
[6] M. Paleczny, C. Vick, and C. Click, “The Java hotspot™ server compiler,” in *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.
[7] Oracle. (2021) Java flight recorder. [Online]. Available: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm
[8] I. Lam. (2018) JEP 310: Application class-data sharing. [Online]. Available: https://openjdk.java.net/jeps/310
[9] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, “The Java Virtual Machine Specification Java SE 12 Edition,” 2019.
[10] V. Kozlov. (2018) JEP 295: Ahead-of-time compilation. [Online]. Available: https://openjdk.java.net/jeps/295
[11] Azul Systems. (2021) About ReadyNow! [Online]. Available: https://docs.azul.com/prime/Use-ReadyNow.html
[12] S. Ritter. (2021) Aot or jit - faster startup, faster code or faster both? [Online]. Available: https://vimeo.com/532992142
[13] K. Pettis and R. C. Hansen, “Profile guided code positioning,” in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 16–27.
[14] D. Chen, T. Moseley, and D. X. Li, “Autofdo: Automatic feedback-directed optimization for warehouse-scale applications,” in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 12–23.