A Pragmatic Approach to Replay Compilation

Andrej Pečimúth
Charles University, Czech Republic
Oracle Labs, Czech Republic

David Leopoldseder
Oracle Labs, Austria

Petr Tůma
Charles University, Czech Republic

Abstract

Dynamic compilers generate code based on the information provided by the virtual machine (VM) running the corresponding application. Due to the environment's non-deterministic nature, every compilation result is typically unique. This is a problem when reproducibility is desired, such as when debugging a crash of the JIT compiler or diagnosing performance problems. As a solution, we present a pragmatic approach to replay compilation that is suitable for integration in a production-grade VM. Our approach is based on instrumenting the VM's compiler interface, allowing us to record the compiler's queries and their results to the VM. We serialize them and use them to replicate the compiler's query results in a replayed compilation. Assuming the compiler is deterministic, this approach systematically ensures that the replayed compilation result is equivalent to the recorded one. The dynamic compiler is invoked directly without the need to execute the original application. A compiler developer can replay a compilation with additional diagnostic options or evaluate metrics such as compilation speed. We developed a working prototype for GraalVM, showing that replay compilation can be implemented without requiring extensive compiler or VM changes. We are working with the GraalVM developers to integrate it into the open-source compiler to unlock these benefits and new use cases for the community.

2012 ACM Subject Classification Software and its engineering → Dynamic compilers; Software and its engineering → Just-in-time compilers; Software and its engineering → Runtime environments

Keywords and phrases replay compilation, dynamic compilation, virtual machines

Funding This research was supported in part by projects GAUK 74824 and SVV 260821.

1 Introduction

Reproducibility of compilation results from a dynamic compiler is often desired but difficult to achieve. For example, when a JIT compilation crashes in a production virtual machine (VM), a compiler developer may need to diagnose the issue by reproducing the crashing compilation with additional diagnostic options. However, the process may require re-executing the application under the same load on the same platform, which is often impossible. Moreover, the crash could occur only in specific conditions that may not be satisfied in every VM execution. Reproducibility may also be required in other scenarios that require a fixed input to the compiler, for example, to measure compilation metrics such as compilation speed accurately.

It is often difficult to reproduce compilation results because they depend on multiple inputs from the VM that vary in time and between VM executions [12]. These inputs include, but are not limited to, the collected profiles and other runtime feedback, the set and content of the loaded classes, the VM configuration and memory layout, the target platform, the contents of compilation-final fields, and the resolution and initialization status of classes, methods, and fields. The input data differs due to the non-deterministic behavior of the VM, the application and the libraries, dynamic class loading, runtime bytecode modification,

dynamically generated classes, and implementation details of the VM's data structures. To replay a compilation and obtain an equivalent result, one needs to ensure that the inputs to the compiler are equivalent and the compiler itself is deterministic. Note that the goal of an accurate replay differs from that of code reuse [12] – the machine code originating from an accurately replayed compilation of an earlier VM instance is typically incompatible with other VM instances and impossible to execute.

Replay compilation is a well-known term in the community. Nonetheless, contemporary VMs lack support for dependable and accurate replay. Jikes RVM [7] could record information such as the optimization level of particular methods and profiling information. The VM can reuse these records in subsequent executions, which can be leveraged as part of a performance evaluation methodology [3, 1, 5, 15]. Similarly, the HotSpot JVM [13] enables replay compilation by recording selected runtime information [4]. However, the replay in HotSpot can fail and does not capture all inputs to the compiler, leading to varying compilation results. The IBM J9 VM (now known as Eclipse OpenJ9) [6] could systematically record all inputs to the compiler while achieving low overhead [9] by reusing information from core dumps. In OpenJ9, the approach was abandoned, citing issues with portability and fragility [8].

2 Pragmatic Replay Compilation

Our approach takes inspiration from the one in IBM J9 [9], but we prioritize maintainability and portability rather than recording overhead. We systematically record and replay all communication between the compiler and the VM, operating within the scope of a high-level object-oriented interface. The GraalVM compiler queries the VM using the JVM Compiler Interface (JVMCI) [14], which is an object-oriented interface implemented in Java that conveys information about VM objects, such as the resolved classes, methods, fields, constants, and other information. Every one of these VM objects is exposed to the compiler in the form of a Java object instance that implements a specific Java interface.

To record a compilation, we create proxies for JVMCI objects whose methods return information from the VM. The created proxies handle method invocation by invoking the intended receiver method and serializing both the arguments and the return value. To replay a compilation, we use proxy classes that handle method invocation by looking up the appropriate return value using the arguments as a key.

We serialize the recorded operations for every compilation unit in a single human-readable JSON file. This allows us to invoke the dynamic compiler programmatically to replay the recorded compilation without running the original application or requiring access to its bytecode.

For example, Listing 1 is a snippet of the recorded file from a compilation of the `Object.<init>` method. The file indicates that the root compiled method is represented by an instance of JVMCI's `method` class, and the instance ID is 0. The file then dictates the return values of method invocations that use instance ID 0 as a receiver. The recorded operations indicate that the `getCode()` method should return a single-element byte array (containing the `return` bytecode). Additionally, the declaring class of the method is an instance of JVMCI's `type` class with ID 1, whose `getName()` method returns `"Ljava/lang/Object;"`. The file captures all information required to complete the compilation. In addition, it includes the serialized compilation result (low-level intermediate representation), allowing us to verify the equivalence of the replayed compilation to the recorded compilation.

Listing 1 A snippet of a recorded compilation unit serialized as a JSON file.

```
{
  "compiledMethod": {"tag": "method", "id": 0},
  "operations": [
    {
      "recv": {"tag": "method", "id": 0},
      "method": "getCode()",
      "res": {"tag": "byteArray", "bytes": [-79]}
    },
    {
      "recv": {"tag": method", "id": 0},
      "method": "getDeclaringClass()",
      "res": {"tag": "type", "id": 1}
    },
    {
      "recv": {"tag": "type", "id": 1},
      "method": "getName()",
      "res": {"tag": "string", "content": "Ljava/lang/Object;"}
    },
    ...
  ]
}
```

3 Prototype Evaluation

Our prototype implementation aims for minimal overhead when recording is disabled and minimal maintenance overhead to ensure that the feature can be integrated into the upstream GraalVM compiler. The prototype requires relatively minor modifications in 51 source files in the compiler and the HotSpot JVM, totaling less than 700 inserted lines and 300 deleted lines. On top of this, we added about 8600 new lines in 45 files. The semi-automatically generated proxy classes comprise about 38% of the added lines. We executed standard pre-merge tests and benchmarks (with recording disabled), and we did not detect any significant regressions in the compilation speed, memory allocated by the compiler, or the compiled code size. Our changes increased the binary size of the compiler[1] from about 50 MB to 52 MB, which is a consequence of introducing the proxy classes. In summary, however, we did not encounter substantial barriers preventing integration in the GraalVM compiler.

Conversely, a preliminary evaluation indicates a roughly 10x slowdown in compilation speed for the recorded compilation units. We consider the overhead acceptable for the intended use cases, and we have not attempted to optimize it yet because such optimizations could conflict with keeping the code maintainable or increase the overhead when recording is disabled. We can mitigate the effect on the running application by recording only compilations of a few selected methods. For example, when an unrecorded JIT compilation crashes, the compiler can immediately repeat the same compilation with recording enabled. This is a pragmatic strategy used by the GraalVM compiler, although it does not guarantee that repeated compilation will experience the issue that caused the crash. We can also enable recording only for the hottest methods, which may be appropriate for other use cases.

[1] The GraalVM compiler is compiled ahead-of-time to a native shared library using Native Image [10].

While our primary use case is debugging and diagnostics, we conclude by sketching other potential applications unlocked by replay compilation. Extending previous work [11], we can replay a compilation using two distinct compiler revisions. This could enable comparing how optimization decisions, code quality, and compilation speed change between two compiler revisions without any noise caused by the VM's non-determinism. A challenge with this and similar use cases is that the replay file may not contain all the required information when the replayed compilation diverges. We also plan to apply replay compilation to reduce the costs of tracking the compiler's performance [2] by running benchmarks only if the compilation results change.

References

1 Michael D. Bond and Kathryn S. McKinley. Continuous path and edge profiling. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005. doi:10.1109/MICRO.2005.16.

2 Lubomír Bulej, François Farquet, Vojtěch Horký, and Petr Tůma. Tracking Performance of Graal on Public Benchmarks. Presentation at International Workshop on Load Testing and Benchmarking of Software Systems (LTB) 2021, 2021. doi:10.6084/m9.figshare.14447823.

3 Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. *SIGPLAN Not.*, 43(10):367–384, 10 2008. doi:10.1145/1449955.1449794.

4 Tobias Hartmann. Debugging the Java HotSpot VM, 2020. Retrieved March 12, 2025. URL: https://cr.openjdk.org/~thartmann/talks/2020-Debugging_HotSpot.pdf.

5 Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. *SIGPLAN Not.*, 39(10):69–80, October 2004. doi:10.1145/1035292.1028983.

6 IBM. Eclipse OpenJ9, 2025. Retrieved March 14, 2025. URL: https://eclipse.dev/openj9/docs/.

7 Jikes RVM Contributors and Core Team. Jikes RVM User Guide, 2021. Retrieved March 14, 2025. URL: https://www.jikesrvm.org/UserGuide/.

8 Daryl Maier. Comment on OpenJ9 GitHub issue #701, 2017. Retrieved March 12, 2025. URL: https://github.com/eclipse-openj9/openj9/issues/701#issuecomment-347723660.

9 Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. Replay compilation: improving debuggability of a just-in-time compiler. *SIGPLAN Not.*, 41(10):241–252, 10 2006. doi:10.1145/1167515.1167493.

10 Oracle. Native Image, 2025. Retrieved March 18, 2025. URL: https://www.graalvm.org/latest/reference-manual/native-image/.

11 Andrej Pečimúth, David Leopoldseder, and Petr Tůma. Diagnosing Compiler Performance by Comparing Optimization Decisions. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 47–61, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3617651.3622994.

12 Andrej Pečimúth, David Leopoldseder, and Petr Tůma. An Analysis of Compiled Code Reusability in Dynamic Compilation. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL '24, page 43–53, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3689490.3690406.

13 John Rose. HotSpot Internals, 2019. Retrieved March 14, 2025. URL: https://wiki.openjdk.org/display/HotSpot.

14 John Rose. JEP 243: Java-Level JVM Compiler Interface, 2019. Retrieved March 12, 2025. URL: https://openjdk.org/jeps/243.

15 Narendran Sachindran and J. Eliot B. Moss. Mark-copy: fast copying GC with less space overhead. *SIGPLAN Not.*, 38(11):326–343, 10 2003. doi:10.1145/949343.949335.