Reusing Highly Optimized IR in Dynamic Compilation

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

David Leopoldseder
Oracle Labs, Vienna, Austria

Petr Tůma
Charles University, Prague, Czech Republic

Abstract

Virtual machines (VMs) with dynamic compilers typically specialize compiled code to the state of the running VM instance and thus cannot reuse the code between multiple runs of the same application. The JIT compiler must recompile the same methods for each run of the application separately, which can prolong the application's warmup time. We propose a technique to reduce compilation time by reusing a highly optimized intermediate representation (IR). We achieve this by tracing compiler-interface calls during compilation. The validity of the specializations in the IR is verified during a replay stage, and the replay also facilitates the relocation of runtime object references. The IR is stored on a compilation server, which can compile it to machine code and provide the code to local or remote VM instances. We implemented a compilation server with IR caching for GraalVM, a high-performance production-grade Java Virtual Machine (JVM). We present an evaluation based on four industry-standard benchmark suites. In each suite, our approach reduces compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% on average while preserving peak application performance.

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 code reuse, compilation time, warmup, remote compilation, dynamic compilation, virtual machines

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.25

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

1 Introduction

Modern virtual machines (VMs) achieve high application performance by leveraging optimizing dynamic compilers. After a VM starts, the application typically experiences a warmup phase [2] until the performance-sensitive methods are compiled at the highest optimization level. The VM usually has to repeat these compilations for each run of the application. The latency of the compilation tasks is one of the factors contributing to the application's cold start. This problem is amplified when many VM instances execute the same application, such as in horizontally scaled deployments with demand-based elasticity and in serverless computing [5].

The goal of our work is to reuse compilation results from a past run of the same application on the same VM version and platform. For brevity, we refer to a single run of the application as a *VM run*, and we want to reuse compilations from a *source VM instance* in a *target VM instance*. Reusing past compilation results reduces the work performed by the JIT compiler in a VM run (compilation time). Depending on the workload's sensitivity to the latency of

Listing 1 An example where the compiler may specialize the compiled `log` method to the state of the running VM instance.

```
class Logger {
    static final int level = Integer.parseInt(System
        .getProperty("logLevel"));

    static final PrintStream stream = createStream();

    static void log(int logLevel, String message) {
        if (logLevel >= level) {
            stream.println(message);
        }
    }
}
```

compilation tasks, we can also reduce the time it takes for the application to reach peak performance (warmup time). However, it is challenging to reuse the code compiled by a highly optimizing compiler because the compiled code is specialized to the source VM instance.

To illustrate how a dynamic compiler tailors code to the source VM instance, consider the `log` method in Listing 1. The compiler may read the value of the `level` field at compilation time and embed its value in the machine code. Similarly, the compiler can read the value of the `stream` field, obtaining an object address that it can embed in the generated code.

These transformations improve performance by moving computation from run time to the compilation stage, but they pose a challenge to code reuse. For example, the compiled code implicitly assumes that the `level` field holds a particular value. The field's value could differ in another VM run, rendering the code incorrect for reuse. Moreover, the reference to the Java object constant obtained from the `stream` field requires relocation before we can reuse the code in another VM run. We give an overview of more problems with code reuse in Section 2.

The main idea of our paper is that the specializations of the code to the state of the running VM can be valid even in future VM runs of the same application. We can reuse compilation results from the source VM instance in the target VM instance, provided that it is possible to validate that the reused code would behave correctly in the target VM instance and relocate the references to VM objects. Both issues are solved by recording and replaying a *compilation trace*. To reduce the overall compilation time with this technique, we also need to ensure that validation succeeds for enough compilation units when the source and target VM instances execute the same application. We achieve this by modifying the compiler and caching the optimized intermediate representation (IR) rather than the compiled code because the IR is less specific to the target VM instance and easier to patch.

To reuse the IR compiled for method `log`, we must verify that the transformations performed during the original compilation for the source VM instance are correct in the target VM instance. Therefore, during compilation in the source VM instance, we record the calls from the compiler to the VM that query the VM's state, obtaining a compilation trace such as the one in Listing 2. In the target VM instance, we replay these calls to validate that they produce compatible results, and we use the computed VM object references as relocations. `%2` is a *trace variable* that stands for the `level` field, which we obtained as index `32` into the constant pool of the `Logger` class. The field's value read by the compiler was `4`,

Listing 2 A snippet of a compilation trace captured during a compilation of the `log` method from Listing 1.

```
                                 (%0 is the Logger#log method)
declaringClass(%0) = %1          (%1 is the Logger class)
   getBytecode(%0) = ...         (the result is binary data)
lookupField(%1,32) = %2          (%2 is the Logger#level field)
  isFinalField(%2) = true
readFieldValue(%2) = 4
lookupField(%1,48) = %3          (%3 is the Logger#stream field)
  isFinalField(%3) = true
readFieldValue(%3) = %4          (%4 is a Java object constant)
isConstantNull(%4) = false
```

which we can validate in the target VM instance. The value read from the `stream` field is an object constant stored in trace variable `%4`, which we can use for relocation. We explain compilation tracing and replay in Section 3.

We implemented the approach for GraalVM [28]. GraalVM is the HotSpot [29] Java Virtual Machine (JVM), where the C2 compiler [35] is replaced with the high-performance GraalVM compiler. Based on the GraalVM compiler, we created a compilation server [18, 19, 1, 24] capable of compiling code for local and remote client VM instances over the network. The server can also cache the IR intended for reuse and compile the stored IR into machine code. Thanks to this design, multiple client VMs can share and benefit from the IR cache. To minimize the overall compilation latency, the client VMs employ a *hybrid compilation* strategy: they offload only large compilations where the server can leverage the IR cache. All other compilations are completed using the VM's local JIT compiler. We describe the design in Section 4.

We evaluate the approach on the DaCapo 23.11 MR1 [3], Renaissance 0.15 [39], and ScalaBench 0.1.0 [44] benchmark suites. Compared to the unmodified GraalVM baseline, hybrid compilation using a co-located compilation server with a pre-populated IR cache decreases compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% in every evaluated suite on average. We verified using the DaCapo suite that the benefits of our approach remain substantial when the server and the client VM are separated by a local network. We present the evaluation in Section 5 and describe the related work in Section 6.

Although we focus on the context of a modern JVM, the key concepts of the approach and the challenges faced by our implementation are likely transferable to other similar platforms. Specifically, the technique to cache and reuse the optimized IR assumes that there is a well-defined and interceptable interface between the VM and the compiler and that the compiler starts with an IR that is not overly specific to the target VM instance (other than references to VM objects). We believe these assumptions to be a common design element of environments with dynamic compilation, though other technical details can also influence the practicality of the approach.

In summary, we contribute the following:

- An approach to reuse highly optimized IR by compilation tracing and replay.
- A case study on a production-grade VM outlining the changes to the compilation pipeline to leverage the approach to achieve an overall improvement in compilation metrics.
- An empirical evaluation on industry-standard benchmarks, which shows that the technique reduces compilation time and can improve warmup time while preserving the application's peak performance.

Listing 3 A class with a method whose compiled code may not be reusable.

```
class Filter {
    static final Options options = getOptions();

    void filter(Collection<Integer> numbers) {
        Predicate<Integer> isOutlier =
            (n) -> n > options.threshold;
        numbers.removeIf(isOutlier);
    }
}
```

2 Reusability of Optimized Code

The code compiled by an optimizing dynamic compiler is usually not reusable across VM runs. This section summarizes the obstacles to code reusability in a modern JVM [37], with practical examples from the HotSpot JVM [29]. These challenges are caused by dynamic code loading and by optimizations that specialize the code to the state of the target VM instance, which are concepts relevant to many modern VMs.

2.1 Equivalence of Loaded Classes

A dynamic compiler tailors the code to the definitions of the loaded classes at a given time. To reuse code compiled for a source VM instance, we must ensure that the classes on the target VM instance are equivalent to the classes loaded by the source VM instance. However, a modern JVM may generate classes at runtime and modify the bytecode for optimization purposes. An application can trigger class loading at any time, so we must not expect that the target VM has precisely the same set of loaded classes.

For example, consider the `filter` method in Listing 3. In the HotSpot JVM, the first execution of the method runs the initialization code for the lambda expression [27], which loads a dynamically generated class representing the function object instance. We must ensure that even the generated classes have compatible definitions between VM runs. However, the generated class is assigned a name that is not stable between VM runs [6]. In addition, HotSpot modifies an operand of a bytecode instruction to link it with a function object factory. As a result, the bytecode of the `filter` method may also differ between VM runs.

2.2 Specialization to Application State

The compiler may evaluate loads from static final fields and even final instance fields if the receiver is a constant. This transformation removes a potentially expensive operation from the code and may also enable other optimizations. For example, in the body of the lambda expression in Listing 3, the compiler may fold the load of the `options` field. If `threshold` is a final instance field, the compiler can fold the load of this field as well. Although these specializations improve code quality, they restrict reusability. In future VM runs, the values of the static and instance fields could differ, rendering the compiled code incorrect for reuse.

2.3 Speculations and Assumptions

The compiler's optimization decisions may involve speculation [11] based on the application's past behavior and profiling feedback. These speculations and assumptions narrow the scope where the compiled code is applicable, but they may improve performance.

For example, the call to the `removeIf` method in Listing 3 produces a null pointer exception if the call's receiver is `null`. If such a situation never occurred previously, the compiler may speculatively omit the exception-throwing branch and instead emit a deoptimization [16]. This allows the compiler to focus only on the likely execution paths, but such compiled code is only beneficial if `removeIf` is not invoked with a null argument.

The compiler can make assumptions about the loaded classes. For example, `List` is an interface with possibly multiple implementations, making the `removeIf` call indirect. However, if only a single implementation is loaded, the compiler converts the call to a direct one and records the assumption about the interface having a single implementation. The VM invalidates the compiled code when it loads another implementation of the `List` interface. Such assumptions limit the reusability of the code, but they typically improve performance.

Lastly, the compiled code may contain implicit assumptions about the VM's state. For example, in Java, classes have static initializers [25], which the runtime invokes when a class is first referenced. If a class is already initialized, the compiled code does not need to check its initialization status. However, such code may not be reusable in another VM run, where the same classes might not yet be initialized.

2.4 Coupling with VM State

The compiler may encode absolute addresses, relative addresses, and offsets of various VM objects in the compiled code. These values usually change between VM runs, making the compiled code not directly reusable. For example, the call to the `removeIf` method in Listing 3 is compiled with the callee's address embedded in the machine code.

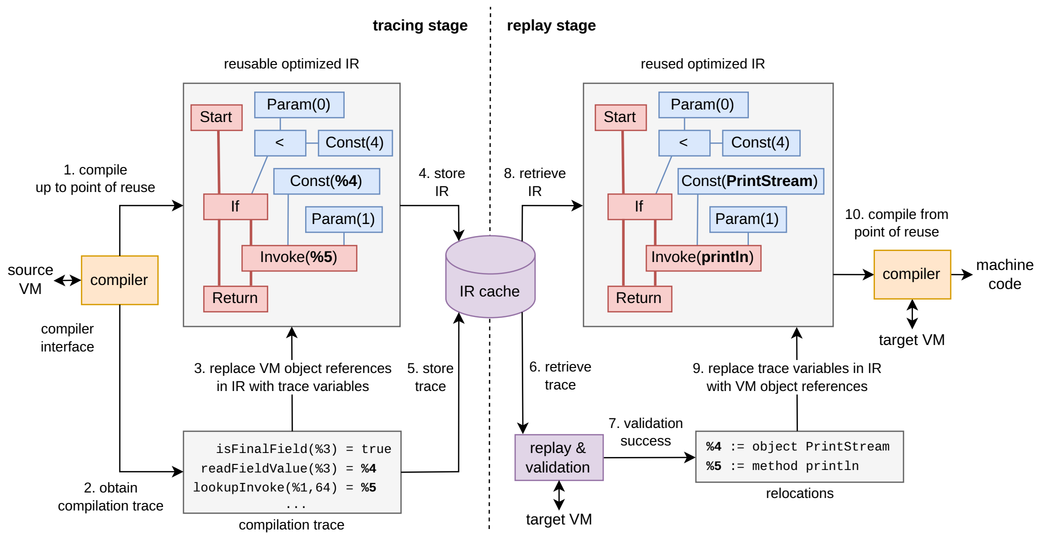
We could attempt to reuse the code by finding relocations for the referenced VM objects and patching their values in the machine code. Java objects have identities, so we must relocate them with identical objects to preserve the program's correctness. However, patching the values in the machine code is difficult because accessing the replacement value may require a different instruction sequence than the one emitted initially.

3 Compilation Tracing and Replay

Although dynamic compilers tailor code to the running VM, these specializations can often remain valid even in future VM runs, much like past profiles can inform future optimization decisions. To capture all dependencies of the compilation result on the state of the running application and source VM instance, we record the compiler's calls through the VM's compiler interface, yielding a compilation trace. In the target VM instance, where we want to reuse the compilation, we replay the compilation trace to validate that the compilation is compatible with the actual state of the running application and to compute relocations for VM object references.

Figure 1 shows a summary of compilation tracing, which this section explains in detail. In a *tracing stage*, we compile a method in a source VM instance up to a *point of reuse* to obtain the compilation unit's optimized IR (step 1). We use the compilation trace (2) to replace VM object references in the IR with trace variables (3). Lastly, we store the IR (4) along with the compilation trace in the IR cache (5). The IR depicted in Figure 1 is that of the `log method` from Listing 1. Interpreting the IR semantics is not essential to understanding the approach; Figure 1 uses Graal IR [10] as an illustration.

In a *replay stage*, we retrieve the compilation trace from the IR cache (step 6) to replay the trace in a target VM instance and validate the results. If the validation succeeds (7), we retrieve the cached IR (8) and replace the references to trace variables in the IR with references to VM objects (9). Finally, we compile the IR to machine code (10) and prepare it for execution on the target VM.



```
isFinalField(%3) = true
readFieldValue(%3) = %4
lookupInvoke(%1,64) = %5
...
```

```
%4 := object PrintStream
%5 := method println
```

Figure 1 Overview of compilation tracing and replay. The IR format is based on Graal IR [10]; thick lines are control-flow edges, thin lines are data-flow edges.

3.1 Point of Reuse

We cache partial compilation results (the optimized IR) [37] instead of the final compiled code because it is less specialized to the state of the target VM instance. To enable IR reuse, we trace the compiler-interface calls in the compiler passes up to the designated point of reuse. The compiler can resume the compilation pipeline starting from the point of reuse in order to compile the cached IR to machine code.

The point of reuse should be as late as possible in the compilation pipeline to minimize the amount of work necessary to finish the compilation. However, it must be early enough so that references to VM objects can be relocated. If the point of reuse is early enough so that it is still possible to optimize the IR, the transformations that make the IR overly specific to the running VM instance can be deferred [37] until after the point of reuse. With these transformations deferred, the compiled machine code can benefit from specialization to the target VM instance while keeping the cached IR reusable.

In summary, the advantages of caching the IR are that

(i) patching the IR is easier than the compiled code,

(ii) we avoid all coupling introduced by late compiler passes after the point of reuse, and

(iii) transformations that introduce coupling to the source VM instance can be deferred until after the point of reuse.

3.2 Tracing

Compilation tracing records inputs to the compiler that influence a method's compilation. Therefore, the trace contains all assumptions about the application's and VM's state. Moreover, the trace captures the origin of every VM object reference that appears in the IR.

In the tracing stage, we run the compilation pipeline up to the point of reuse, obtaining an IR and an associated compilation trace. The compilation trace is a list of the compiler's queries sent to the VM with their arguments and results.

A value in the compilation trace may be a literal or a trace variable substituting a VM object. For example, the bytecode of the compiled methods and the values of primitive constants are embedded in the trace as literals. In Figure 1, trace variable `%4` substitutes a Java object constant obtained by reading the value stored in field `%3`. Trace variable `%5` substitutes a VM's representation of a method.

After obtaining the IR and the compilation trace, we replace all references to VM objects in the IR with the trace variables. Finally, we store the IR and the compilation trace in the IR cache.

3.3 Replay

In the replay stage, we retrieve the IR and the associated compilation trace from the IR cache. The goal is to verify that we can safely reuse the IR without causing the application to misbehave and to finish compiling the IR to machine code.

We replay the trace by iterating over the individual operations. For each operation, we first replace the trace variables in the argument list with the instances of VM objects they substitute. After that, we execute the operation in the target VM and examine its return value. If the trace's result is a trace variable referenced for the first time, then the return value of the operation defines the value of that trace variable. Otherwise, we validate that the return value is compatible with the result captured in the trace. Compatibility can be verified by checking that the results are equal or with an operation-specific check, as we explain in Section 4.

If the validation fails for an operation, we cannot use the IR for the target VM instance at this time. If the validation succeeds, all trace variables have assigned values. We proceed by substituting the trace variables in the IR with their values. Finally, we run the rest of the compilation pipeline, starting from the IR and obtaining the machine code for the target VM instance.

4 Implementation

Compilation tracing serves as a blueprint for validating the reusability of prior compilation results. To improve the overall compilation time using this approach, we must ensure that the cached IR is generally reusable, and we need a mechanism to share it among multiple VMs that might execute on different machines.

We implemented compilation tracing in GraalVM [28]. GraalVM is the HotSpot JVM [29] where the GraalVM compiler replaces the C2 compiler [35]. On top of this, we created a compilation server [19, 24] capable of IR caching, allowing local or remote VM instances to share a single IR cache.

Figure 2 shows an overview of our design, which this section explains in detail. In a *training run* (left side of Figure 2), a source VM instance executes an application and offloads compilation tasks to the compilation server (step 1). The server queries information from the source VM instance (2), caches the compilation result (3), and returns the compiled code to the source VM (4).

A subsequent run of the application may be executed as a *replay run*, which is sketched on the right side of Figure 2. In a replay run, the target VM obtains a compilation trace from the server (step A1), replays and validates it (A2), and returns the computed relocations to the server (A3). The server compiles the cached IR (A4, A5) and returns the compiled code to the target VM (A6). Compilation tasks that the target VM decides not to offload to the server are completed using the VM's local JIT compiler (step B1).

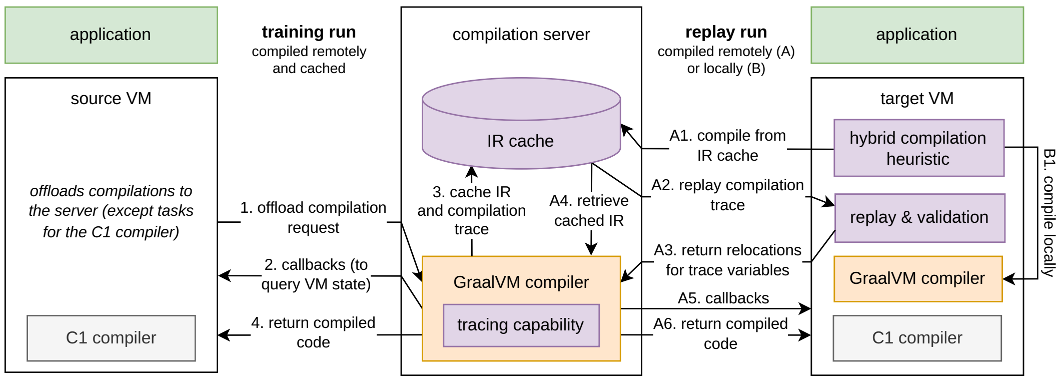

application
source VM
offloads compilations to the server (except tasks for the C1 compiler)
C1 compiler
training run
compiled remotely and cached
1. offload compilation request
2. callbacks (to query VM state)
4. return compiled code
compilation server
IR cache
3. cache IR and compilation trace
A4. retrieve cached IR
GraalVM compiler
tracing capability
replay run
compiled remotely (A) or locally (B)
A1. compile from IR cache
A2. replay compilation trace
A3. return relocations for trace variables
A5. callbacks
A6. return compiled code
application
target VM
hybrid compilation heuristic
replay & validation
GraalVM compiler
C1 compiler
B1. compile locally

Figure 2 Overview of the implementation with a compilation server.

The compilation server is a standalone Java application that uses the GraalVM compiler as a library. The compilation client is implemented as an additional entry point of the GraalVM compiler, with some modifications and additions to the compiler and the VM's compiler interface.

4.1 Practical Compilation Tracing

We implemented compilation tracing by instrumenting the classes comprising the VM's compiler interface, creating wrapper classes with a compatible interface. The traced operations generally align with the granularity of the compiler-interface [40] calls exposed by the VM. A key challenge of compilation tracing is handling information that varies between VM runs, such as the bytecode, profiles, and dynamically generated classes.

Reusing Graal IR. The GraalVM compiler's compilation pipeline is divided into two stages: the front end and the back end. The front end parses the bytecode into a sea-of-nodes-based IR [10], optimizes it, and lowers the IR to a control-flow-graph-based representation for the back end. The compiler performs the majority of the optimization effort in the front end. Based on our measurements (Section 5), the compiler spends on average about 81% of the time in the front end, 14% in the back end, and 5% in the garbage collector triggered after compilation or waiting for the VM to prepare the compiled code for execution. We cache and reuse the IR at a point relatively late in the front end – before the compiler lowers portable address nodes to machine-specific nodes. This allows us to patch the IR with VM-specific information and defer the folding of field loads after the point of reuse (Section 4.2), while still saving a significant portion of compilation time.

Tracing Profiling Information. The IR stored in the cache is compiled using profiles recorded by the source VM instance. When the target VM instance reuses the IR from the cache, the compiled code should be performant and unlikely to deoptimize [16] in the future. Therefore, the cached IR should ideally be compiled using accurate and mature profiles.

After a target VM instance starts, we want to reuse IR as early as possible to leverage the performance of the compiled code. By reusing cached IR early in an application's lifecycle, we can potentially benefit from the fact that the profiles used to compile the IR may more accurately characterize the application's future behavior than the profiles collected by the target VM instance up to that point. For this reason, the replay stage does not verify the

Listing 4 Encoding of the operation that returns the recorded receiver types.

```
getReceiverTypes(%0, 42) = {
  com.example.Map/com.example.Main: %1,
  com.example.Set/com.example.Main: %2,
}
```

equivalence of profiling information between the source and target VM instances, and the compilation trace does not capture any profiling information other than receiver-type profiles. Receiver-type profiles are used during replay to locate some VM objects.

Receiver-Type Profiles. The VM records the types of receiver objects at virtual call sites to guide the inlining [38] heuristics. The compiler queries the receiver-type profiles from the VM, and the receiver types may be referenced in subsequent operations. Therefore, we must assign trace variables to the recorded receiver types. We do this by capturing the receiver-type profiles in the compilation trace.

Listing 4 illustrates the encoding of these profiles in the form of globally unique type identifiers mapped to trace variables. In the example, the `getReceiverTypes` operation returns the receiver types for a particular call site identified by the bytecode offset 42. The recorded receiver types are `com.example.Map` and `com.example.Set` assigned to trace variables `%1` and `%2`, respectively.

Since multiple class loaders may load a class with the same name, the globally unique identifier of a type comprises the class name and an identification of the type's class loader. We identify the class loader using the heuristic introduced by JITServer [19]: the loader's identifier is the name of the first class it loaded. In the above example, `com.example.Main` is the class loader's identifier.

In the replay stage, the goal is to map the trace variables to the respective types identified by the globally unique name. If we cannot find a type (it may not be loaded yet), the replay cannot continue and ends with a validation failure.

Dynamically Generated Classes. During execution, the VM may dynamically generate classes whose class name is not a stable identity across VM runs. For example, the HotSpot JVM loads generated classes to support lambda expressions and other core features. These generated classes are loaded as *hidden classes* [6], which are intentionally designed not to be discoverable by name, and their VM-assigned names vary between VM runs. The VM may compile the methods of hidden classes, and they may also appear in profiles and get inlined. To successfully validate compilation traces referencing hidden classes, we create stable class names based on a digest computed from the binary data that defines the hidden class and also from the Java objects passed to the static initializer of the hidden class.

Additionally, the code of an application may dynamically generate classes without using the hidden class feature. We can successfully reuse the compilations referencing these classes if their definitions are stable between VM runs and their class names uniquely identify them. However, these conditions are not always satisfied: for example, the Catalyst Optimizer [13] from the Apache Spark framework generates and assigns the exact same name to every generated class. Our prototype does not support such a use case and experiences validation failures. The problem could be fixed by adding a digest of the class definition to the class name.

Bytecode Modification. The VM can modify the operands of some bytecodes for optimization purposes in a way that is not stable across runs. The compiler may use these values as arguments in compiler-interface calls. To ensure the replayability of the compilation trace, we do not embed any VM-specific operand values in the compilation trace. Instead, we use an indirection specifying where to find the actual value in the target VM instance.

For example, our compiler interface offers an operation called `lookupInvoke`, which takes a class and index argument and returns the representation of the method invoked at that index. The compiler may obtain the index from an operand of a bytecode instruction and call the `lookupInvoke` operation with the operand value. In the compilation trace, we encode the operand value indirectly using the expression `operandValue(%0, 16)`.

```
lookupInvoke(%1, operandValue(%0, 16)) = %2
```

During replay, we find the concrete index as the value in the bytecode of the `%0` method at bytecode offset 16. Then, we can replay the `lookupInvoke` operation with the concrete argument value specific to the target VM instance.

Lastly, we must also validate that the bytecode of a particular method on the target VM instance matches the bytecode of this method on the source VM instance. We do this by comparing a canonicalized version of the bytecode, which does not contain any VM-specific values.

4.2 Improving IR Reusability

To improve the overall compilation time using our approach, we need to increase the chances that the IR stored in the cache passes validation so that we can reuse it. To this end, we validate the results of some operations with less strict checks than equivalence. Moreover, we use heuristics to defer optimizations that could break the IR's reusability.

Relaxed Validation Checks. We reuse the cached IR even if it is more general than required by the target VM instance. Specifically in the context of Java, if the cached IR leverages an assumption about the loaded classes (e.g., only a single implementation of an interface is loaded), we must check that the assumption holds in the target VM instance. However, if such a fact holds in the target VM instance but the cached IR does not leverage it, we consider this a validation success. Such a situation is expected when the cached IR is more mature than the state of the target VM instance.

Similarly, classes, methods, and fields in Java have initialization stages [25]. To facilitate IR reuse, we only require that a class, method, or field is in the same or later initialization stage on the target VM instance as in the source VM instance.

Patterns Preventing Reuse. If the compiler specializes the IR to computed constants whose values could change between VM runs, the IR may not be reusable. For example, consider the `ThreadLocal` implementation [4] shown in Listing 5. When a `ThreadLocal` instance is initialized, the `threadLocalHashCode` field is assigned a value based on a global counter. If a thread-local value is stored in a static final field, the thread-local's hash code is likely to differ between VM runs. The compiler may embed this field's value into methods that manipulate the thread-local value, leading to non-reusable IR and replay failures. In OpenJDK, the `ClassValue` implementation [41] also uses the same pattern, and similar patterns may appear in the code of applications. We can mitigate these problems using deferred optimization.

Listing 5 A snippet of the `ThreadLocal` implementation in OpenJDK (simplified).

```
class ThreadLocal {
    final int threadLocalHashCode = next.getAndAdd(INCREMENT);
    static AtomicInteger next = new AtomicInteger();
}
```

Deferred Optimization. We can allow the compiler to fold loads of computed constants while keeping the IR reusable by deferring [37] this optimization to the replay stage. In the context of Java, if we know that a particular field like the one in Listing 5 hinders reusability, we do not allow the compiler to fold a load of this field in the tracing stage. The compiler folds the field load in the replay stage when the VM-specific value of the field is known.

Deferring the folding of all fields is a disruptive change that leads to performance degradation [37]. Therefore, we target only fields that may cause validation failures. Our prototype implementation uses a simple condition: if the field's type is `int` and its name contains `hash`, we defer the folding. A more refined approach could select these fields based on past observations: if validation fails due to mismatching values read from a field, the field's folding would be deferred in future tracing stages.

4.3 Remote Compilation

We implemented a compilation server [19, 1, 24] with the capability to reuse the IR from past compilations. This allows multiple client VMs to share a single IR cache. Figure 2 shows a summary of the design. We implemented remote compilation and IR caching only for the top-tier GraalVM compiler since the low-tier C1 compiler [20] completes compilation tasks quickly and with a lower optimization level. To minimize compilation latency, the client VMs utilize hybrid compilation, in which a heuristic decides whether a method should be compiled by the VM's local JIT compiler or using the cached IR on the server.

Compilation Server. In a remote compilation setup, a client VM sends compilation requests to the server. The server compiles methods for the client VM's platform, utilizing only the information provided by the client. It is usually not possible to pack all the necessary information [19] in the forwarded compilation request. Instead, the server queries the client's VM state via remote callbacks. We employ aggressive caching and prefetching to minimize the number of remote callbacks. After completing the compilation, the server returns the compiled code to the client VM.

Training Run. The compilation server starts with an empty IR cache, which gets populated by a client VM that connects in training mode. In a training run, the compilation server handles requests with compilation tracing enabled. When the compilation pipeline reaches the point of reuse, the server stores the compilation trace and copies the IR into the cache. After that, the server completes the rest of the compilation pipeline and returns the compiled code to the client VM.

The IR cache is indexed by a symbolic representation of the compilation request. This symbolic representation comprises the fully qualified name, arguments, and the return value of the method to be compiled and the entry point if it is an on-stack-replacement (OSR) [12] compilation.

Replay Run. In a replay run, the client VM either chooses to compile locally or issues a remote compilation request. The decision is based on a server-provided list of compilation requests and other conditions, which we summarize later. In case of a remote compilation, the server looks up a cached compilation result in the IR cache and transfers the stored compilation trace to the client VM. The client VM replays and validates the trace. If validation succeeds, the client replies with the computed mapping of trace variables to VM objects. The server substitutes the trace variables with the VM object references in the IR, completes the compilation, and returns the compiled code. In case of a validation failure, the client VM completes the compilation request using its local JIT compiler instead.

The client VM must also handle recompilations. A recompilation of a method previously compiled from the cached IR may occur after deoptimization [16] or a violated assumption about the class hierarchy. In such a case, we cannot reuse the same IR to get a usable compilation result and prevent endless deoptimization. For this reason, the client VM handles recompilations by compiling from scratch using the local JIT compiler.

Permanent and Transient Validation Failures. Retrying to replay a compilation trace that previously failed can improve the latency of some compilation tasks. As explained above, the VM compiles a method using the local JIT compiler if the validation of a compilation trace fails. However, the code compiled by the local JIT may get invalidated later, and the VM may reissue an identical compilation request. We attempt to replay and validate the compilation trace again if the reason for the past failure is an uninitialized or unresolved class, unresolved field, or an uninitialized method. We consider these failures to be *transient* because they may be resolved by the VM performing the required initialization [25]. We consider other kinds of validation failures to be *permanent*, and the client VM never retries to replay the same compilation trace again.

Hybrid Compilation. To improve the overall compilation time, client VMs offload only the compilation requests where the IR cache is expected to bring a benefit. If there is no expected benefit, the client VM should avoid increasing compilation latency by communicating with the server. We estimate the potential savings based on the time the server spent compiling the bytecode to the reusable IR in the training run, which we refer to as the *pre-compilation time*. The pre-compilation time is an upper-bound estimate of the time a client VM can save by compiling the cached IR on the server rather than locally from scratch. Therefore, at the beginning of a replay run, the client VM obtains a list of compilation requests from the server for which the server has cached IR and the pre-compilation time above a client-provided threshold.

In the evaluated prototype, we set the pre-compilation time threshold to 20 milliseconds. The optimal threshold value depends on the workload, the communication latency between the client and the server, and the computing resources available to the client and server.

In summary, a client VM offloads a compilation request to the server if and only if all conditions are satisfied:

(i) The compilation request is in the server-provided list.

(ii) There was no previous attempt to complete this compilation request that ended with a permanent validation failure.

(iii) This compilation request was not already completed using the cached IR.

5 Evaluation

The goal of this section is to understand how hybrid compilation impacts the performance characteristics of standard Java and Scala workloads. We evaluate the workloads from major benchmark suites [3, 39, 44]. To ensure that the improvements are not achieved by offloading work to an additional machine or allocating more resources to the JIT compiler, we run the compilation server co-located on the same machine as the client VM, and we do not increase the number of concurrent compilation tasks. We first show the impact on key compilation metrics reported by the HotSpot JVM [29]. Then, we analyze the impact on peak performance and warmup time. Lastly, we evaluate how successful our prototype is in reusing and validating the cached IR and explain the reasons for the observed validation failures.

In a setup where a client VM connects to a remote server, the network latency adds to the compilation latency. For this reason, we consider low-latency environments (e.g., a fast local network) the most appropriate deployment option. The benefits would likely diminish in environments with higher latency. To demonstrate that the presented improvements are applicable to deployments on the local network, we also conduct experiments using the DaCapo benchmarks [3] where the client VM connects to a dedicated remote compilation server through the local network. We do not increase the number of concurrent compilation tasks to keep the results comparable to the co-located and baseline setups.

Notes on Performance Evaluation. Every benchmark suite we evaluate contains multiple benchmarks. The harness of every suite executes a benchmark by iterating a fixed workload multiple times in a dedicated JVM instance. The *iterations* usually get progressively faster as the VM warms up. The harness repeats the workload until it executes for at least the preset duration (e.g., 12 minutes). Due to the inherent non-determinism of the JVM, we also need to execute each benchmark for multiple VM runs [14]. The key metrics we evaluate are the wall-clock time it takes to complete an iteration (*iteration time* or *duration*) and the timestamps of their completion since the start of the JVM instance (*iteration timestamp*). In the rest of the section, we always aggregate absolute metric values using the arithmetic mean and relative metric values (ratios) using the geometric mean.

Hardware and Software. We executed the experiments on 20 identical blade servers, each equipped with an Intel Xeon CPU E3-1230 v6 (with four cores) and 32 GB of main memory. The blades are located in the same enclosure with separate 1 Gbps Ethernet network cards connected through a dedicated network router. We disabled hardware multithreading and power management. The blades run Fedora Linux 35 with kernel version 5.16.11.

We implemented our prototype on top of a development version of GraalVM Enterprise based on a development version of OpenJDK 22. We use the same binaries and the same JVM options for the modified and baseline runs, using additional JVM options to enable the remote JIT functionality. This avoids the potential variance introduced by the ahead-of-time compilation of the compiler itself. In every benchmark run, we use a JVM option to disable compressed class pointers (`-XX:-UseCompressedClassPointers`) because our prototype does not implement patching the compression parameters in the cached IR. We disable isolation between compiler threads (`-XX:JVMCIThreadsPerNativeLibraryRuntime=0`) to enable data sharing required by the client's compilation threads. In addition, we prevent the VM from shutting down the compiler runtime after a period without compilation activity (`-XX:JVMCICompilerIdleDelay=0`) to avoid potentially costly reinitialization of the runtime for the client. The options `-Xms12G` and `-Xmx12G` set a fixed Java heap size, and we use the default garbage collector (`-XX:+UseG1GC`). Lastly, we use options to enable selected counters and timers in the GraalVM compiler to collect compilation statistics.

We use modified versions of the DaCapo 23.11 MR1 [3] harness and the ScalaBench 0.1.0 [44] harness to collect additional statistics. We execute the Renaissance 0.15 [39] benchmark suite with additional plugins to collect the same statistics. All DaCapo benchmarks are executed using the default workload size. We execute the ScalaBench benchmarks with default or larger sizes, selected to ensure the iterations are not excessively short. We exclude the workloads incompatible with OpenJDK 22.

5.1 Compilation Metrics

Experimental Setup. We measure the compilation metrics described below by running each benchmark for 12 minutes. On our platform, 12 minutes is typically sufficient time for the VM to warm up and then perform multiple iterations with peak performance. We manually verified that the tested benchmarks warmed up in the first 6 minutes of the run by plotting the iteration durations. For each benchmark, we repeat 30 VM runs of the baseline setup and the same number of VM runs with the hybrid setup that uses the cached IR for selected compilations. The client VM with hybrid compilation connects to a server running either on the same machine or a remote machine. In the *co-located* setup, the server runs on the same machine. In the *remote* setup, the compilation server runs on a dedicated machine, and all client VMs connect through the local network. Since the server is a Java application, we warm it up[1] with three unmeasured training and replay runs. Before each measured run of the client VM, we populate the IR cache of the compilation server with a single 12-minute training run of the same benchmark. To capture the potential variability between training runs, we do not reuse one training run for multiple replay runs.

Metrics. We collect the metrics reported by the JVM using the option `-XX:+CITime`. *Compilation time* is the sum of the wall-clock durations of all compilation tasks. Therefore, for the client VM with hybrid compilation, the compilation time metric includes the time spent on communication and serialization, as well as any failed attempts to reuse the cached IR. *Compiled bytecodes* is the total size of the bytecodes compiled in every compilation task, comprising the bytecode size of the root method and inlinees. *Compiled methods* is the total number of completed compilation tasks. Although we cache the IR only for the top-tier GraalVM compiler and not the C1 compiler [20], we report the sums of values for both compilers to present a complete picture. For multiple benchmark runs in the same setup, we report the arithmetic average of each metric value.

Statistical Evaluation. To ensure that the reported averages are not excessively impacted by random fluctuations, we used bootstrapping to compute 99% confidence intervals for all relative compilation and performance metrics reported in the paper. We did this by resampling with replacement from the VM runs of every benchmark in a particular setup. Because the confidence intervals are narrow relative to the reported effect size for every metric except peak performance, where the change is insignificant, we omit them from the presentation to enhance readability.

[1] We could avoid the server's warmup by compiling it ahead of time, similarly to how the GraalVM compiler is built.

Table 1 Compilation metrics in DaCapo benchmarks with a co-located server (lower is better).

benchmark	compilation time		compiled bytecodes		compiled methods	
	baseline	hybrid	baseline	hybrid	baseline	hybrid
eclipse	155.68 s	**−28.6%**	16.25 MB	**−9.5%**	30342	**−5.1%**
pmd	115.08 s	**−42.8%**	8.52 MB	**−13.2%**	18188	**−5.0%**
fop	95.19 s	**−33.6%**	8.83 MB	**−4.1%**	20191	**−3.0%**
spring	85.38 s	**−17.9%**	9.71 MB	**−3.7%**	25970	**−1.7%**
tomcat	72.84 s	**−15.2%**	9.82 MB	**−7.4%**	25230	**−3.3%**
cassandra	61.71 s	**−18.6%**	6.34 MB	**−9.6%**	19851	**−2.9%**
jython	56.75 s	**−38.4%**	6.24 MB	**−5.7%**	9591	**−6.0%**
zxing	48.87 s	**−59.3%**	3.01 MB	**−21.3%**	5705	**−7.6%**
kafka	46.88 s	**−27.9%**	5.40 MB	**−9.5%**	18418	**−2.9%**
h2	40.83 s	**−33.0%**	3.19 MB	**−21.7%**	5749	**−8.2%**
batik	38.91 s	**−20.4%**	2.45 MB	**−10.8%**	6302	**−3.1%**
xalan	18.76 s	**−28.2%**	1.69 MB	**−3.4%**	4521	**−0.6%**
luindex	18.31 s	**−33.8%**	2.23 MB	**−8.2%**	5950	**−2.9%**
lusearch	16.12 s	**−45.1%**	1.63 MB	**−14.9%**	4160	**−2.8%**
jme	12.43 s	**−40.2%**	1.43 MB	**−11.3%**	4480	**−2.1%**
sunflow	9.24 s	**−53.2%**	0.89 MB	**−18.4%**	2277	**−3.8%**
graphchi	9.04 s	**−43.2%**	0.65 MB	**−17.0%**	2389	**−2.6%**
biojava	8.75 s	**−51.4%**	0.82 MB	**−13.3%**	2553	**−3.7%**
avrora	8.30 s	**−45.0%**	0.88 MB	**−19.9%**	3209	**−2.4%**

Table 2 Changes in compilation metrics for every benchmark suite (lower is better).

benchmark suite	server setup	compilation time	compiled bytecodes	compiled methods
DaCapo	co-located	**−36.8%**	**−11.9%**	**−3.7%**
Renaissance	co-located	**−23.6%**	**−8.6%**	**−3.5%**
ScalaBench	co-located	**−35.9%**	**−9.8%**	**−4.3%**
DaCapo	remote	**−31.8%**	**−11.0%**	**−3.3%**

Results. Table 1 lists the compilation metrics for the individual DaCapo benchmarks with a co-located compilation server. The columns marked *baseline* show the arithmetic average of a metric value for all baseline runs of the given benchmark. The columns marked *hybrid* show the change in the arithmetic average of a metric value for the hybrid runs relative to the baseline. Hybrid compilation significantly reduces compilation time for every benchmark from 15.2% up to 59.3%. The size of the compiled bytecodes decreases by 3.4% to 21.7%, and the number of compiled methods decreases by up to 8.2%.

Table 2 summarizes the changes in compilation metrics in every evaluated suite. We aggregate the metric values for a benchmark suite as the geometric mean of the relative metric values of the individual benchmarks. In the setup with a co-located compilation server, the overall decrease in compilation time ranges between 23.6% and 36.8%, the size of the compiled bytes decreases by between 8.6% and 11.9%, and the number of compiled methods decreases by between 3.5% and 4.3%. The overall improvements in the DaCapo suite with a remote server are substantial but slightly lower than those in the co-located setup.

Discussion. The drop in compilation time is large in all benchmark suites, which is a result of multiple factors. The cached IR directly decreases compilation time by reducing the amount of work performed. As an indirect effect, faster compilations can lead to fewer compilation tasks being scheduled. This is because compilation tasks for the JIT compilers are scheduled when method invocation and loop back-edge counters overflow. By speeding up the compilation tasks, the compiled methods (including inlined methods) are executed via the compiled code that does not increment the counters. Consequently, these counters have fewer opportunities to overflow, and fewer compilation tasks may be scheduled. The data shows that both the total size of the compiled methods and their number decreased significantly, which is evidence of less work scheduled for the JIT compilers.

Another indirect effect that can save compilation time is the higher maturity of the cached IR, i.e., the compilations may have a lower chance of deoptimizing [16] on an assumption or speculation [11] and requiring a recompilation. This is because the cache stores the last completed compilation of a method from the training run, and, potentially, the IR may not contain speculations or assumptions that are bound to be violated later during the VM run.

During development, we found that our prototype can sometimes suffer from an excessive number of recompilations. This happens when the compiler deoptimizes from a compiled method to an earlier block in the control-flow graph, and the VM executes the bytecode instruction that triggered the deoptimization as part of a more mature compiled method rather than in the interpreter. Consequently, the VM fails to update the profiles for future recompilations. Our prototype can trigger this problem due to the presence of compiled code of varying maturity (from the IR cache and locally compiled). Changing how speculations are tracked in the IR [11] could improve the results, and we reported the issue to the GraalVM compiler developers. A few runs of the DaCapo `luindex` benchmark in the remote setup seem to suffer from excessive recompilation, inflating the difference between the co-located and remote setups. Without `luindex`, the compilation time savings in the remote setup are 33.6% instead of 31.8%.

The improvements vary between the benchmarks because multiple variables determine the success of our approach. For example, some benchmarks use code patterns that lead to non-reusable code, which we analyze later. In some workloads, the VM often compiles different methods across two VM runs, leading to cache misses. Lastly, compilation units differ in their size, and IR caching may benefit larger compilations more.

The difference between the co-located and remote setups can likely be attributed to the communication latency. A possible mitigation for this problem is hiding the latency by increasing the number of concurrent compilation tasks [18, 19].

5.2 Performance Metrics

We use the measurement data collected from the experiment introduced above to analyze how our approach impacts peak performance and warmup time.

Peak Performance. The quality of the code compiled by a dynamic compiler is sensitive to factors such as the timing of compilation tasks. Therefore, we consider it a good practice to confirm that performance is unaffected. We measure the peak performance in terms of the average iteration time when the VM is warmed up. We acknowledge that the workloads may not reach a single steady state [2], and variation in iteration time is expected. As a pragmatic choice, we compute the arithmetic average of the durations of iterations completed after the 6-minute mark since VM start (lower is better). Note that the workloads typically warm up in the first 6 minutes of the run on our platform.

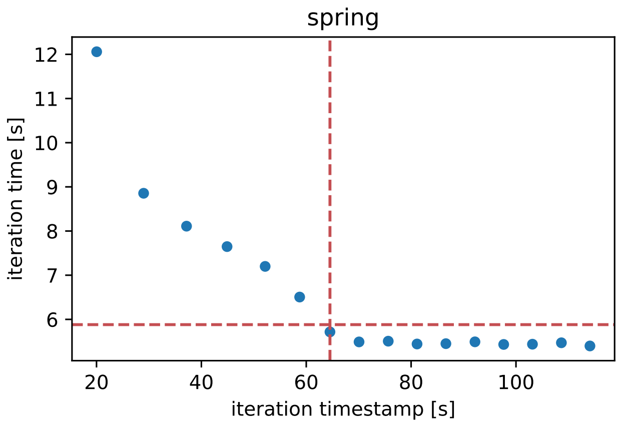


Figure 3 Iterations from the beginning of a baseline run of the DaCapo `spring` benchmark, highlighting the warmup threshold (horizontal line) and the warmup time (vertical line).

Warmup Time. There are multiple options for precisely defining warmup, but there is no standard and straightforward option. Related work [19] uses 90% of peak throughput as the threshold to consider an application warmed up. Since we evaluate benchmarks consisting of multiple workload iterations, we define warmup time as the timestamp of the first iteration where the iteration time is below a warmup threshold. We compute the warmup threshold analogously to the related work [19] as 1/(90%) of the mean iteration time after the 6-minute mark (the peak performance metric). We performed a sensitivity analysis to verify that the overall results vary by no more than a few percent with values of the warmup threshold parameter between 1/(80%) and 1/(95%). Due to performance anomalies that we explain below, we exclude the `scala-kmeans` and `gauss-mix` benchmarks from the warmup time evaluation in the Renaissance suite.

To visualize our warmup detection method, Figure 3 plots the iteration timestamps and durations of the DaCapo `spring` benchmark from the start of a single run. The horizontal line is the computed warmup threshold. The vertical line marks the computed warmup time.

Benchmarks with high iteration times present a potential problem for our and similar warmup detection methods. Because we determine warmup only after an iteration completes, we risk overestimating small changes in warmup time that occur within an iteration. This concern is reduced in benchmarks with low iteration times. We verified that restricting the evaluation to 25% of the benchmarks with the lowest baseline iteration time yields similar overall results.

We encountered these performance anomalies that are relevant to warmup detection:
1. high variance in iteration time within a VM run (in DaCapo `xalan` and other benchmarks),
2. the application experiences a small but continuous speedup (in DaCapo `fop` and other benchmarks),
3. slowing down after the first iteration (in DaCapo `kafka`), and
4. large shifts in performance (in Renaissance `scala-kmeans` and `gauss-mix`).

These anomalies occur in both the baseline and hybrid setups. We manually inspected plots similar to Figure 3 to verify that the problems caused by the first three anomalies are mitigated by the warmup threshold. In DaCapo `kafka`, the first iteration is already below the warmup threshold, so the warmup detection yields reasonable results. The performance shifts in Renaissance `scala-kmeans` and `gauss-mix` cause significant variance in the measured

Table 3 Performance metrics in DaCapo benchmarks with a co-located server (lower is better).

benchmark	warmup time		iteration time	
	baseline	hybrid	baseline	hybrid
cassandra	104.69 s	**−15.2%**	11.73 s	**−0.2%**
pmd	84.15 s	**−45.1%**	1.99 s	**+0.8%**
eclipse	76.03 s	**−24.6%**	12.43 s	**+0.2%**
spring	64.23 s	**−17.7%**	5.27 s	**+0.1%**
jython	63.63 s	**−5.6%**	2.99 s	**−1.0%**
fop	54.69 s	**−37.5%**	0.48 s	**−4.7%**
h2	45.36 s	**−7.0%**	8.46 s	**−1.0%**
tomcat	35.75 s	**−0.3%**	30.24 s	**−0.1%**
luindex	32.12 s	**−16.6%**	4.50 s	**−0.1%**
zxing	30.85 s	**−60.5%**	4.04 s	**+0.5%**
lusearch	25.55 s	**−6.9%**	9.17 s	**−0.6%**
graphchi	23.76 s	**−0.7%**	5.29 s	**−0.2%**
sunflow	22.54 s	**−7.3%**	8.96 s	**+1.5%**
biojava	17.77 s	**−4.2%**	6.33 s	**+0.2%**
xalan	16.50 s	**−23.1%**	2.63 s	**−0.3%**
jme	16.11 s	**−1.0%**	6.88 s	**+0.1%**
kafka	15.19 s	**−0.7%**	14.20 s	**−1.4%**
batik	13.45 s	**−21.0%**	1.54 s	**+0.4%**
avrora	6.15 s	**+4.0%**	4.93 s	**−0.3%**

Table 4 Changes in performance metrics for every benchmark suite (lower is better).

benchmark suite	server setup	warmup time	iteration time
DaCapo	co-located	**−17.4%**	**−0.3%**
Renaissance	co-located	**−13.1%**	**−0.2%**
ScalaBench	co-located	**−21.2%**	**0.0%**
DaCapo	remote	**−17.3%**	**−0.2%**

warmup time. In multiple runs, warmup is detected at the point of a performance drop occurring several minutes into the run, which can substantially impact the average warmup time. The performance shifts may be caused by inlining instability [36], which is a known compiler issue that manifests in these benchmarks. For the above reasons, we consider these benchmarks unsuitable for comparing warmup time between two systems, and we exclude them when computing the overall warmup change in the Renaissance suite. The DaCapo and ScalaBench suites do not experience performance shifts of this nature.

Results. Table 3 lists the performance metrics for the individual DaCapo benchmarks with a co-located compilation server. As before, we show the absolute values for the baseline runs and the relative values for the hybrid runs. Our approach improves the warmup time for almost all DaCapo benchmarks. The peak performance of the hybrid runs is usually close to the baseline, with few minor deviations in both directions.

Table 4 summarizes the changes in performance metrics for every evaluated setup. In the setup with a co-located server, the overall decrease in warmup time is between 13.1% and 21.2%. The iteration time of the hybrid setup does not significantly deviate from the baseline. The overall warmup improvement in the DaCapo suite with a remote server is similar to that of the co-located setup.

Discussion. Our approach significantly improves warmup time in most workloads, which we likely achieve primarily by reducing the latency of compilation tasks. When compilation latency is reduced, the VM can start executing efficient code earlier. However, due to the nature of dynamic compilation, compilation time reductions do not transfer linearly to warmup improvements. Just by changing the latencies of compilation tasks, we also impact the collected profiles, the methods selected for compilation, and, consequently, how the compiler optimizes code. Lastly, compilation speed is only one of the many factors that influence warmup.

The presented approach aims not to degrade peak performance. In the vast majority of benchmarks, there is no significant impact, and the overall change is also small. However, in individual benchmarks, our approach can change when a particular method is compiled (compared to the baseline), which in turn impacts profiles and optimization. In the Renaissance suite, a small number of workloads are susceptible to performance changes in either direction. These changes are likely caused by inlining instability [36].

5.3 Analysis of IR Reusability

We evaluate the success of our prototype in reusing the cached IR based on the relative size of the bytecode compiled from the cache, which we refer to as the cache hit rate. We also analyze the reasons for the observed validation failures and discuss the options for improving IR reusability in future work.

Cache Hit Rate. In every 12-minute run of the hybrid setup, we compute the cache hit rate as the ratio of the bytecode size of the compilation tasks served from the IR cache and the total bytecode size of all compilation tasks. We use the geometric mean to aggregate the metric.

In the co-located setup, the cache hit rate is about 69% in the DaCapo suite, 64% in the Renaissance suite, and 77% in the ScalaBench suite. In the DaCapo suite with a remote server, the cache hit rate is 71%.

The results show that the majority of the compiled bytecodes originate from the IR cache. The client VMs never reach a cache hit rate of 100% in any of the runs since we deliberately try to compile small methods using the local JIT compiler to keep compilation latency low. The other factors that decrease the cache hit rate are cache misses, recompilations, and validation failures.

Transient Validation Failures. The evaluated benchmarks experience a varying number of validation failures. *Transient failures* are the most common type of validation failures, comprising between 89% and 96% of the validation failures in every evaluated benchmark suite. As explained before, they are caused by the cached IR referencing a class that is not yet loaded by the target VM or referencing a class, method, or field that is in an earlier initialization stage than required to reuse the IR. These failures occur in almost every benchmark because we attempt to reuse mature IR as early as possible. The average number of transient failures varies between benchmarks. DaCapo `eclipse` experiences these failures most frequently, averaging more than 250 failures per run.

Permanent Validation Failures. All other types of validation failures are permanent and together comprise between 4% to 11% of the observed failures in every benchmark suite. The vast majority of Renaissance workloads are affected by permanent failures. In contrast, about half of the DaCapo benchmarks and the vast majority of ScalaBench benchmarks do not experience permanent failures at all.

Class hierarchy mismatch is a permanent failure that occurs when a class hierarchy query on the target VM returns an incompatible result. For example, it occurs when an interface has a single loaded implementation during the tracing stage but multiple loaded implementations during the replay stage. This kind of failure occurs in most Renaissance benchmarks and in about a third of DaCapo benchmarks, but only a low number of compilation tasks are affected on average: no DaCapo benchmark experiences more than 4 of these failures per run.

The *field value mismatch* permanent validation failure occurs when the compiler folds a read of a static final field and the value differs from the expected one. The failure occurs in most Renaissance benchmarks and in about a third of the DaCapo benchmarks. The `cassandra` benchmark is the most affected in the DaCapo suite, experiencing about 13 such failures on average.

Lastly, *type definition mismatch* and *method definition mismatch* are permanent failures occurring due to an incompatible definition of a class or a method. For example, an incompatible class may have different fields, and an incompatible method may have different bytecodes. These failures occur frequently in only a few of the benchmarks. In the Renaissance `dec-tree` benchmark, these incompatibilities are caused by the classes generated by the Catalyst Optimizer [13] from the Apache Spark framework. As we explained previously, our prototype cannot distinguish between the multiple generated classes in this case since they all share the same class name.

Mitigating Validation Failures. We invested a substantial engineering effort to minimize the number of validation failures, and there are still options for further improvement. One of the problems with our prototype is recording the compilation trace also for the expanded inlining candidate methods [38], even if these candidates are not ultimately inlined. There is an opportunity to avoid the validation failures caused by the inlining candidates, which we could achieve by filtering out the operations irrelevant to the inlined methods.

We could mitigate many of the transient validation failures by loading pre-initialized classes [47], which is a planned but so far unrealized feature of the OpenJDK project Leyden [15]. The client VM could trigger the loading of a pre-initialized class when the class is first referenced in a compilation trace. Pre-initialization requires caching [48] the values of the static final fields, which would also mitigate the failures caused by field value mismatch. Together, these features would open up the option for eagerly compiling [18] the cached IR and prefetching the code compiled at the highest optimization level.

6 Related Work

Reusing Compilation Results. Shared Class Cache (SCC) [17] is a technology to improve startup and warmup performance in the OpenJ9 JVM. SCC stores preprocessed class data in a memory-mapped file, and it can also store compiled code. Reusability of the code is achieved by compiling it with a lower optimization level [19] and patching the compiled machine code [8]. Due to the lower performance of the cached code, the JIT compiler must recompile the hottest methods to reach peak throughput. SCC uses a process similar to compilation tracing to establish the provenance of VM objects [9], such as methods and classes. However, the mechanism is not used to facilitate specializing the code for the running VM instance. In our approach, we leverage compilation tracing almost without any restrictions on the optimizations that can be applied. Thus, the code compiled from our IR cache achieves peak performance and does not require recompilation. ShareJIT [46] and ShMVM [7] are systems similar to SCC that facilitate sharing code compiled with reduced optimization level on a single machine. `jaotc` [21, 22] was a tool to precompile Java class files ahead of time to native code for the HotSpot JVM.

Pečimúth et al. [37] explain the reasons why highly optimized code is not reusable in GraalVM. The authors suggest caching the IR and making it reusable by deferring all optimizations that potentially specialize the IR to the running VM. However, this approach reduces peak performance in many workloads. In our work, we instead let the compiler perform the optimizations that specialize the IR to the VM, and we employ compilation tracing to facilitate IR reuse without sacrificing peak performance.

The Truffle language implementation framework [45] can store the compiled code along with other data structures in a file [31] to improve warmup in future VM runs. The guest language running in this mode (e.g., JavaScript) must satisfy specific conditions. For example, the language implementation cannot speculate on object identity, since it would lead to a guaranteed deoptimization. The approach we present does not have this restriction because replaying the compilation trace allows us to relocate object references in the IR.

Ř+ [26] is a system to reuse compiled code for the R language. In Ř+, the compiler tracks its assumptions and stores them along the compiled code. The code repository may contain multiple versions of the same function, each fit to a different context. At run time, calls are dispatched to the appropriate version based on the current context. Although the HotSpot JVM does not perform such contextual dispatch, our IR cache could potentially benefit from storing multiple versions of a method. For example, the most mature version of a method is sometimes not usable early in the application's lifetime due to assumptions about the loaded classes.

Remote Compilation. JITServer [19, 18] is a remote JIT compilation server for the OpenJ9 JVM. The design of our compilation server, with callbacks for VM runtime information and the caching of their results, is inspired by JITServer. JITServer leverages compiled-code caching based on OpenJ9's SCC [17] infrastructure. Therefore, the code loaded from the cache requires recompilation at the highest optimization level. The main use case presented by the authors is remote compilation for resource-constrained containers in cloud data centers. Resource-constrained containers benefit from the offloading of compilation tasks to a remote server and achieve warmup improvements even without requiring code caching. The compilation server running on another, non-constrained machine effectively extends the computing resources available to the VM running in a container. JITServer can hide the latency of the offloaded compilation tasks by issuing more concurrent compilation tasks. It can also trigger remote compilation eagerly and prefetch [18] the compiled code to the client VMs.

In contrast, our work focuses on reusing the IR compiled at the highest optimization level without needing recompilation to reach peak performance. We decrease the compilation latency by reusing the cached IR. We present an evaluation where the compilation server is co-located with the client without increasing the number of concurrent compilation tasks or dedicating more computing resources to JIT compilation.

Cloud Native Compiler [1] is a production-grade compilation server that can reuse profiles from past JVM runs. However, there is little technical information available. The compilation server for Jikes RVM by Lee et al. [24] is an earlier work in which compilation tasks are served in a single request without callbacks. However, this is difficult to achieve in a modern JVM due to the dynamicity of the runtime and method inlining.

Improving Warmup. OpenJDK project Leyden [15] is a comprehensive effort addressing the startup and warmup phase of Java applications via various techniques, such as storing compiled code in class-data archives [42], ahead-of-time loading of classes in the linked [23] or initialized [47] state, and other techniques that may require new language features. OpenJDK project CRaC [30] mitigates warmup time by checkpointing running Java applications (creating snapshots), which can then be restored. However, the checkpointed application must manage resources such as open files and sockets, so the process is not transparent to the programmer. Jump-Start [34] is a technique that improves warmup by sharing profiling information for the HipHop VM, which eliminates the overhead of collecting profiles.

Ahead-of-Time (AOT) Compilation. Native Image [32] builds native executables by compiling Java class files ahead of time with the closed-world assumption (i.e., classes cannot be loaded at run time). AOT compilation can eliminate warmup but may impact peak performance. The performance of dynamic languages like JavaScript often depends on speculation and runtime feedback, so AOT compilers [43, 33] usually cannot produce as efficient code as JIT compilers.

7 Conclusion

This paper introduces a novel technique to reuse highly optimized IR in dynamic compilers. We do this by tracing the arguments and results of compiler-interface calls, allowing us to verify that the assumptions made during a compilation of reusable IR hold in a particular target VM instance. Using this approach, we can relocate the IR's VM object references and compile the IR for the target VM instance. A compilation server caches the IR from previous runs of the application and can compile it to machine code for local or remote client VMs. An empirical evaluation based on the GraalVM compiler and industry-standard benchmark suites shows that the approach reduces compilation time by 23.6% to 36.8% and warmup time by 13.1% to 21.2% in each suite without hindering peak performance.

References

1 Azul. Cloud Native Compiler, 2024. Retrieved August 14, 2024. URL: `https://docs.azul.com/optimizer-hub/about/cloud-native-compiler`.

2 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. `doi:10.1145/3133876`.

3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006. `doi:10.1145/1167515.1167488`.

4 Josh Bloch and Doug Lea. ThreadLocal.java in OpenJDK, 2023. Retrieved September 2, 2024. URL: `https://github.com/openjdk/jdk/blob/fe9f05023e5a916b21e2db72fa5b1e8368a2c07d/src/java.base/share/classes/java/lang/ThreadLocal.java`.

5 Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 58–64, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3458336.3465305`.

6 Mandy Chung. JEP 371: Hidden Classes, 2020. Retrieved February 20, 2025. URL: `https://openjdk.org/jeps/371`.

7 Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code sharing among virtual machines. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, pages 155–177, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. `doi:10.1007/3-540-47993-7_7`.

8 Irwin D'Souza. Ahead Of Time Compilation: Relocation, 2018. Retrieved August 22, 2024. URL: `https://blog.openj9.org/2018/10/26/ahead-of-time-compilation-relocation/`.

9 Irwin D'Souza. Ahead Of Time Compilation: Validation, 2018. Retrieved August 22, 2024. URL: `https://blog.openj9.org/2018/11/08/ahead-of-time-compilation-validation/`.

10 Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, February 2013. URL: `http://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf`.

11 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2542142.2542143`.

12 Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, pages 241–252, USA, 2003. IEEE Computer Society. `doi:10.1109/CGO.2003.1191549`.

13 Apache Software Foundation. CodeGenerator.scala in Apache Spark, 2025. Retrieved February 20, 2025. URL: `https://github.com/apache/spark/blob/bbb9c2c1878e200d9012d2322d979ae794b1d41d/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/codegen/CodeGenerator.scala#L1516`.

14 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1297027.1297033`.

15 Dan Heidinga. Choose your own performance; a Project Leyden update, 2024. Retrieved August 23, 2024. URL: `https://openjdk.org/projects/leyden/slides/leyden-heidinga-devnexus-2024-03.pdf`.

16 Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. Association for Computing Machinery. `doi:10.1145/143095.143114`.

17 IBM. Introduction to class data sharing, 2024. Retrieved August 14, 2024. URL: `https://eclipse.dev/openj9/docs/shrc/`.

18 Alexey Khrabrov. *Disaggregated Just-in-Time Compilation for the Java Virtual Machine*. PhD thesis, University of Toronto, June 2024. URL: `http://hdl.handle.net/1807/140205`.

19 Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 869–884, Carlsbad, CA, July 2022. USENIX Association. URL: `https://www.usenix.org/conference/atc22/presentation/khrabrov`.

20 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), May 2008. `doi:10.1145/1369396.1370017`.

21 Vladimir Kozlov. JEP 295: Ahead-of-Time Compilation, 2018. Retrieved September 2, 2024. URL: `https://openjdk.org/jeps/295`.

22 Vladimir Kozlov. JEP 410: Remove the Experimental AOT and JIT Compiler, 2021. Retrieved September 2, 2024. URL: `https://openjdk.org/jeps/410`.

23 Ioi Lam, Dan Heidinga, and John Rose. JEP 483: Ahead-of-Time Class Loading & Linking, 2024. Retrieved August 27, 2024. URL: `https://openjdk.org/jeps/483`.

24 Han B. Lee, Amer Diwan, and J. Eliot B. Moss. Design, implementation, and evaluation of a compilation server. *ACM Transactions on Programming Languages and Systems*, 29(4):18–es, August 2007. `doi:10.1145/1255450.1255451`.

25 Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java® Virtual Machine Specification: Initialization*, chapter 5.5. Oracle, Java SE 22 edition, 2024. URL: `https://docs.oracle.com/javase/specs/jvms/se22/html/jvms-5.html#jvms-5.5`.

26 Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2), October 2023. `doi:10.1145/3622839`.

27 Oracle. LambdaMetafactory.java in OpenJDK, 2021. Retrieved March 2, 2025. URL: `https://github.com/openjdk/jdk/blob/fe9f05023e5a916b21e2db72fa5b1e8368a2c07d/src/java.base/share/classes/java/lang/invoke/LambdaMetafactory.java`.

28 Oracle. GraalVM Overview, 2024. Retrieved August 14, 2024. URL: `https://www.graalvm.org/latest/docs/introduction/`.

29 Oracle. HotSpot Internals, 2024. Retrieved August 14, 2024. URL: `https://wiki.openjdk.org/display/HotSpot`.

30 Oracle. Project CRaC, 2024. Retrieved August 14, 2024. URL: `https://openjdk.org/projects/crac`.

31 Oracle. Auxiliary Engine Caching, 2025. Retrieved February 26, 2025. URL: `https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AuxiliaryEngineCachingEnterprise/`.

32 Oracle. Native Image, 2025. Retrieved February 26, 2025. URL: `https://www.graalvm.org/latest/reference-manual/native-image/`.

33 Oracle. Truffle AOT Overview , 2025. Retrieved February 26, 2025. URL: `https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AOTOverview/`.

34 Guilherme Ottoni and Bin Liu. HHVM jump-start: boosting both warmup and steady-state performance at scale. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 340–350. IEEE Press, 2021. `doi:10.1109/CGO51591.2021.9370314`.

35 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

36 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, pages 47–61, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3617651.3622994`.

37 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, pages 43–53, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3689490.3690406`.

38 Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 164–179. IEEE Press, 2019. `doi:10.1109/CGO.2019.8661171`.

39 Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 17, 2019. `doi:10.1145/3314221.3314637`.

40 John Rose. JEP 243: Java-Level JVM Compiler Interface, 2019. Retrieved February 27, 2025. URL: `https://openjdk.org/jeps/243`.

41 John Rose. ClassValue.java in OpenJDK, 2024. Retrieved September 2, 2024. URL: `https://github.com/openjdk/jdk/blob/b1163bcc88a5b88b9a56d5584310f1d679690ab2/src/java.base/share/classes/java/lang/ClassValue.java`.

42 John Rose. JEP draft: Ahead-of-Time Code Compilation, 2025. Retrieved February 25, 2025. URL: `https://openjdk.org/jeps/8335368`.

43 Manuel Serrano. JavaScript AOT compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 50–63, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3276945.3276950`.

44 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM. `doi:10.1145/2076021.2048118`.

45 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2509578.2509581`.

46 Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. `doi:10.1145/3276494`.

47 Jiangli Zhou. Java Class Pre-resolution and Pre-initialization, 2020. Retrieved February 27, 2025. URL: `https://cr.openjdk.org/~jiangli/Leyden/Java%20Class%20Pre-resolution%20and%20Pre-initialization%20(OpenJDK).pdf`.

48 Jiangli Zhou and Thomas Schatzl. Caching Java Heap Objects, 2019. Retrieved February 20, 2025. URL: `https://wiki.openjdk.org/display/HotSpot/Caching+Java+Heap+Objects`.