An Analysis of Compiled Code Reusability in Dynamic Compilation

Andrej Pečimúth
Charles University
Prague, Czech Republic
pecimuth@d3s.mff.cuni.cz
Oracle Labs
Prague, Czech Republic
andrej.pecimuth@oracle.com

David Leopoldseder
Oracle Labs
Vienna, Austria
david.leopoldseder@oracle.com

Petr Tůma
Charles University
Prague, Czech Republic
petr.tuma@d3s.mff.cuni.cz

Abstract

Large applications reliant on dynamic compilation for performance often run in horizontally scaled architectures. When this is combined with frequent deployment or demand-based scaling, hardware capacity is lost to frequent warmup phases due to the need to recompile the code after each start of the virtual machine (VM). Moreover, the individual VMs waste hardware resources by repeating the same compilations.

Offloading compilation jobs to a dedicated compilation server can mitigate these problems. Such a server can compile the code in a mode where the compilation result is reusable for multiple VMs. The goal is to save compilation resources, such as CPU and memory, and potentially improve the warmup time of individual VMs.

This paper investigates the options to reuse previous compilation results of a high-performance compiler. Rather than reusing machine code, we propose to reuse a pre-optimized intermediate representation (IR). Reusability is achieved by deferring VM-specific optimizations until the IR is compiled to machine code for a concrete VM. In an empirical study using the GraalVM compiler and the HotSpot Java VM, the slowdown of code compiled with deferred optimization ranges between a negligible impact and a 6x slowdown. However, the code still performs significantly better than the code compiled by a lower-tier compiler. Therefore, the presented approach can form the foundation for improving warmup times in certain workloads.

CCS Concepts: • **Software and its engineering** → **Dynamic compilers**; **Just-in-time compilers**; *Runtime environments.*

VMIL '24, October 20, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1213-5/24/10
https://doi.org/10.1145/3689490.3690406

Keywords: remote compilation, code sharing, JIT compilation, virtual machines

ACM Reference Format:
Andrej Pečimúth, David Leopoldseder, and Petr Tůma. 2024. 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), October 20, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689490.3690406

1 Introduction

Applications running on dynamic runtimes experience a warmup phase [2] when the JIT is compiling code. Until optimized machine code is available, a virtual machine (VM) executes application code via an interpreter or compiles it with a baseline compiler. Moreover, the JIT compiler itself competes for computing resources with the running application. The effect is temporarily degraded performance [3] of the running application.

For the above reasons, modern VMs are usually more efficient in long-running workloads where the resources spent for JIT compilation are amortized over a longer period. However, this does not match the deployment setup of many applications. In horizontally scaled setups, there are multiple VMs where each VM performs JIT compilation individually. If there is demand-based scaling, VMs may be stopped and restarted. VM cold starts are also a concern in serverless computing, where 96% of functions take less than a minute to execute on average [25]. Although cloud providers minimize the frequency of cold starts by keeping idle instances alive for several minutes, the cost of cold starts is traded for the idle instances that waste memory.

Frequent VM restarts amplify the issues associated with the warmup phase. When a VM stops, all compiled code is typically lost. The individual VMs repeat the same compilations whenever they start, wasting hardware resources on redundant work. End users may also observe the negative effects of frequent VM warmups, e.g., as order-of-magnitude increases [3] in request latency.

Remote JIT compilation [1, 14, 16] minimizes the overhead of JIT compilation on the VM executing the application by

```
static final String logLevel = System
    .getProperty("log.level");
void doWork() {
    if (logLevel.equals("verbose")) {
        System.out.println("in␣doWork");
    }
    doWorkInternal();
}
```

Listing 1. A static final field whose value may change between VM runs.

offloading the compilation jobs to a dedicated compilation server. The compilation server may additionally cache and reuse compilation results from previous VM runs. The result is a shortened warmup phase, fewer CPU- and memory-usage spikes caused by the JIT compiler, and fewer resources spent on compilation overall.

This paper is part of the research [22] on how to reap the benefits of remote JIT compilation with code caching in the GraalVM compiler [19] and other dynamic languages implemented on top of it. While attempting to reuse code compiled with a highly optimizing compiler, we encountered several interesting challenges that are not discussed in existing publications. Our findings may interest the community as we believe they apply to other similar compilers.

Optimizing dynamic compilers specialize the compiled code to the state of the application [10] and the VM, which improves performance but hinders code reusability. For example, the method in Listing 1 enables logging based on the value of a system property stored in a static final field. The compiler can evaluate the branch condition at compilation time and either keep or completely remove the logging code if it is disabled. Although this is a beneficial optimization, it is not correct to reuse the compiled code in another VM run if the field's value differs.

To keep the compiled code reusable, existing implementations of code caching [6, 13, 29] typically use a lower optimization level for compilations intended for later reuse. The lower optimization level reduces the peak performance of the cached code. For example, the field's value in Listing 1 is different when the VM has a different value of the system property. Consequently, the code compiled for reuse using this approach evaluates the branch condition for each method call at runtime.

Reusing code compiled with a lower optimization level, like in the existing work, does not necessarily transfer well to VMs such as GraalVM. In GraalVM, most of the compilation time is spent compiling with the highest optimization level using its top-tier GraalVM compiler. Only a fraction of the time is usually spent in the lower-tier client compiler [15], which employs less aggressive optimization passes. The client compiler already emits code relatively fast, so the potential compilation-time savings of caching the client compiler's code are limited.

The major contribution of our work is an analysis of why code compiled by a dynamic compiler might not be reusable. The causes stem from the non-determinism of the runtime environment and the executed application combined with aggressive optimizations performed by the dynamic compiler. We illustrate the problems on the HotSpot Java Virtual Machine (JVM) [26] and GraalVM compiler [19]. However, these reasons are conceptual and apply to various runtimes.

Unlike the existing work, we aim to reuse the results of previous compilations with high optimization levels. We propose to cache the intermediate representation (IR) instead of the compiled code since we can pre-optimize the IR in a way that is not bound to a specific VM. To leverage the performance gains from VM-specific optimizations, we apply them to the reusable IR. In summary, our approach is the following: (1) begin the compilation conservatively (avoiding or deferring optimizations that could break reusability), (2) cache the IR for later reuse, (3) at the time of reuse, apply the deferred optimizations, and finish the compilation.

This paper tests the feasibility of the proposed approach using an empirical study of deferring optimizations in the GraalVM compiler. We measure the peak-performance impact of deferring optimizations in industry-standard benchmarks [23] and analyze the effects on optimization decisions. Compared to the unmodified GraalVM compiler, the reduction in peak performance ranges from a negligible impact to a 6x slowdown in the most affected workload. The cause of the slowdown is related to the failure to optimize the usage of a particular dynamic feature of the JVM. However, the code compiled using the GraalVM compiler with deferred optimization is a significant improvement over the client compiler [15], which is the lower-tier compiler [27] used in HotSpot JVM.

In conclusion, we contribute the following:

- an analysis of the causes for code non-reusability in modern runtimes illustrated on the HotSpot JVM and the GraalVM compiler,
- an approach to reuse compilations results of an optimizing compiler by deferring VM-specific optimizations to a later stage,
- an experimental evaluation of the peak-performance impact of deferring optimizations in the GraalVM compiler.

2 Related Work

There are many past and ongoing efforts to improve the warmup of JVM implementations. Code sharing improves warmup by reusing code originally compiled by another VM because it decreases resource usage of the JIT compiler and also decreases the compilation latency (the time between the VM issues a compilation request until executable code is ready). Remote JIT compilation may improve warmup by offloading compilation requests, usually to a machine with more computational resources, which also works by decreasing the compiler's resource usage and compilation latency. Finally, we examine a broader set of approaches that improve warmup but are otherwise orthogonal to our work.

2.1 Code Sharing

The existing implementations of compiled code sharing for the Java platform work on a single machine and employ lower optimization levels to make the code reusable. Shared Class Cache (SCC) [13] and ShMVM [6] store code in a file containing pre-parsed class metadata optimized for quick loading. ShareJIT [29] is a global code cache implemented using shared memory. We describe each of these systems in more detail.

SCC [13] is a mechanism in the OpenJ9 VM that allows storing class data and compiled code in a memory-mapped file. The code is compiled with a lower optimization level to facilitate reusability. Consequently, the performance of the cached code is higher than that of the interpreter but lower than that of the JIT-compiled code. The cached code is associated with validation records [8] so that the VM can ensure the code is compatible with the environment in which it is about to be loaded. The SCC also contains relocation records [7]: before the cached code is loaded, the VM uses these records to patch the addresses embedded in the code with process-specific addresses.

ShMVM [6] is an implementation of class-data and code sharing for an earlier version of the client compiler [15] in the HotSpot JVM [26]. References to process-specific addresses are solved using an indirection table: a pointer embedded in the code refers to an entry in the indirection table. The compiled code looks up the value of the entry in the indirection table, and the value is the actual process-specific address.

ShareJIT [29] is a global code cache for the Android Runtime. To make code shareable, ShareJIT disables optimizations that are specific to the VM process. For example, it does not embed the absolute address of a directly invoked method in the compiled code, as the address may change between VM invocations. Additionally, the compiler inlines only built-in methods to simplify the verification of whether the definition of the cached method is compatible.

Ř+ [17] is a system to cache the IR and compile it offline, targeting the R language. Ř+ keeps multiple versions of compiled methods specialized to different contexts. The context comprises both the global application context and the context at the call site, e.g., the type of a particular variable. While Java VMs (like HotSpot) generally do not dispatch method calls based on the calling context, maintaining multiple compiled method versions could benefit code sharing by increasing the cache hit rate. This is because assumptions and speculative optimization [10] specialize the compiled methods to a particular VM state.

2.2 Remote Compilation

JITServer [14] is a remote compilation server for the OpenJ9 VM. In this setup, the compilation server queries the client VM for information needed for compilation in multiple network calls. The authors show that JITServer particularly benefits containers executing a saturating workload with few computing resources. This is because remote compilation moves the overhead of the JIT compilation to the server, freeing up the resources available to the running application.

JITServer can additionally reuse previously compiled code. This feature is built on top of the SCC [13] functionality in OpenJ9. For a remote compilation request, the JITServer may reply with a serialized SCC entry to a client. This serialized entry contains references to classes and methods that require patching. After receiving the entry, the client looks up, validates, and patches the references. The implementation is based on global identifiers assigned to classes and methods. The global identifier of a class comprises its fully-qualified name and a hash of class metadata. Methods are identified by their declaring class, name, and signature. As the cached code is compiled with a lower optimization level, the hottest methods are recompiled by the JIT compiler.

Azul Cloud Native Compiler [1] is another example of a compilation server implemented for a production-grade Java VM. However, there is little technical information available. The available documentation does not mention compiled code caching, but the system reuses profiles from previous VM runs.

In the past, there were other compilation server implementations [16] usually focused on the objective of offloading the resource-intensive JIT compilation to a different machine. However, the past work is not easily transferable to contemporary high-performance VMs because the VMs and compilers are more complex, and the runtime has become more dynamic — even fundamental features such as lambdas require dynamic code generation. For instance, the compilation server for Jikes RVM [16] can fulfill the compilation request in just a single roundtrip because it has access to the application's class files. In contrast, nowadays, much of the information the server needs cannot be found in the class as it is generated on the fly.

2.3 Other Approaches

AOT compilation is another option to improve the warmup. This approach usually restricts the dynamic features of the runtime. For example, GraalVM Native Image [20] works under the closed-world assumption — all loaded classes must be known in advance. OpenJDK project Leyden [5] also aims to explore weaker constraints with different performance tradeoffs. The advantage of remote compilation is that it is more transparent and does not put constraints (such as the closed-world assumption) on the application.

OpenJDK project CRaC [4] makes it possible to snapshot and restore a warmed-up VM process. The application must be able to restore open files or sockets, so the mechanism is not transparent to the programmer.

In serverless computing, cloud providers usually keep the application instance running for about 20 or more minutes [28] even when if there is no function executing. The rationale is to improve the warmup of future function invocations. The downside is that the idle VM wastes memory. This approach is orthogonal to code reuse because code reuse aims to improve the warmup during a cold start, i.e., when a new application instance is provisioned.

3 Reusability of Compiled Code

This section examines why code compiled by a modern dynamic compiler is not reusable and what options exist to make it reusable. We present the conceptual reasons that apply to many languages beyond Java. We discuss our experience with the HotSpot JVM to illustrate these concepts in a practical implementation.

JIT compilers produce code that is specialized to the state of the VM and the application. Many of the reasons why code is not reusable stem from differences in the runtime environment across VM runs (e.g., varying memory layout) and different behavior of the running application itself. As the changes in the runtime's and application's behavior are outside the compiler's control, we say that the runtime and the application behave non-deterministically. The problem with non-determinism is that it renders the compiled code non-reusable in a future run because the state the code is specialized to will likely differ. We discuss the non-determinism of the runtime in Section 3.1 and the non-determinism of the application in Section 3.2.

The compiled code may depend not only on the VM state that changes between VM runs, such as the memory layout, but also on the state that changes during the single VM's lifetime. An example of this is assumptions about loaded classes and speculative optimization [10]. Generally, this kind of specialization enables more optimization opportunities for the compiler, leading to faster code, but it hinders reusability. We describe the optimizations based on VM state in Section 3.3.

As a related problem, the compiled code often contains references to VM objects, such as class and method references and object constants. To install the code[1] in a different target VM, we must find the matching VM objects in the target VM. The process is similar to relocation performed by a linker. Ensuring that it is possible to find a mapping for all VM object references in the compiled code could lead to further restrictions on the compiler. We describe these challenges in Section 3.4.

3.1 Non-Determinism in the Runtime Environment

Compiled code may not be reusable because the runtime environment behaves slightly differently in each VM run. For efficiency reasons, the compiled code often directly references VM objects. However, these objects typically reside at different addresses in another VM run.

To reuse compilation results with references to VM objects, we could try to patch the addresses in the machine code. However, this approach introduces the following challenges:

- The actual value of an address may dictate what kind of instruction sequence can access it, so the compiler may have to generate unnecessarily conservative code.
- The compiled code may access an offset within a VM object, which may also need relocation. For instance, the referenced object could be a virtual method table whose layout changes between VM runs.
- The referenced object may not have a unique global identifier, so it may be difficult to relocate the reference in another VM run. For example, the referent may be an object on the heap or a dynamically generated class (generated by the application or even the runtime).

For a practical illustration using the HotSpot JVM, consider the Java method in Listing 2. The code constructs highlighted in comments lead to code that is not directly reusable. We expect the problems will be similar to those of another

```
class Box {
    Object content;
}
void processBoxes(ArrayList<Box> boxes) {
    // (a) lambda expression
    Consumer<Box> consumer = box -> {
        // (b) VM object reference
        if (box.content instanceof
                String string) {
            // (c) constant reference
            String message = "Box:␣" +
                string;
            // (d) direct call
            System.out.println(message);
        }
        // (e) field write
        box.content = "foo";
    };
    // (f) virtual call
    boxes.forEach(consumer);
}
```

Listing 2. Compiled code reusability of this Java source is affected by the non-determinism of the runtime environment.

[1]Code installation comprises preparing the compiled code for execution and placing it in executable memory.

modern VM that was not designed with code reusability in mind.

Each construct highlighted in Listing 2 hinders reusability because it introduces the following VM-specific code:

(a) The implementation of lambda expressions in Java [11] involves dynamically generated classes, and the compiled code references this class.
(b) The instance check leads to a reference to the `String` class in the compiled code.
(c) The string literal is compiled as an object reference.
(d) The direct call references the target method if it is not inlined.
(e) A field write may require supporting code for the garbage collector (GC) [18], which can reference GC-related objects.
(f) A virtual call may be compiled as a lookup in a virtual method table [24], where the method entry offset could change between VM runs in the HotSpot JVM.

3.2 Non-Determinism in Application Code

Optimizing JIT compilers take advantage of access to the application's state. For example, after a final static field is initialized, its value cannot change. Therefore, at the IR level, the compiler may replace all loads of this field with a constant node holding the field's value — we call this *field-load folding*. We showed an example of a foldable field load in Listing 1.

Field-load folding is an optimization available to JIT compilers but generally not possible for AOT compilers. This happens when the application code computes the field's value, and the compiler cannot determine the result statically. As illustrated in Listing 1, the initializer may compute a different field value in another VM run. In this situation, a JIT compiler has the edge over an AOT compiler because it needs to compile only for this particular VM run rather than all possible runs.[2]

More optimization opportunities may arise after the compiler replaces the IR node representing a field load with a constant IR node. The compiler may evaluate the expressions that become constant. In Listing 1, we showed that the optimizer could remove branching and potentially the dead code. All these transformations based on the constant's value may be invalid in another VM run if the field's value is different.

Suppose the constant is an object constant rather than a primitive one. In that case, the compiler may leverage all kinds of information about it, such as the exact type and the values of final instance fields. If the constant is an array, the optimizer may read its length and access the individual array elements. This information enables optimization that is often specific to the state of the application in this particular VM run.

[2]Native Image can execute static initializers at build time, but this changes the semantics of the language.

Unfortunately, it is relatively easy to write code that is non-reusable due to the above reasons. Consider the example in Listing 3, which shows an instance of the problem from the Java Class Library. The `getInstance` method returns a single instance of the service for each thread. However, the compiled code of `getInstance` would likely not be reusable. This is due to the non-deterministic initialization of the `threadLocalHashCode` field in combination with field-load folding and inlining.

The root of the problem is the `ThreadLocal` class, which stores a thread-local hash code in a private final field. This hash code comes from the atomic integer and is used to index into the map with thread-local values. If the same thread-local object, such as `Service#instance`, is initialized in a different order relative to other thread-local objects, the assigned hash code is different. During compilation, the compiler folds the code that computes the map index based on the hash code. This code then may get inlined into the `getInstance` method. As a result, the compiled code now depends on the state of the running application. Patching the compiled code would be difficult, as the constant value embedded in the code may not directly be the value of the `threadLocalHashCode` field but rather the result of a computation based on the field's value.

3.3 Optimizations Based on VM State

Some optimizations tailor [10] the generated code to the state of the running VM. The motivation is to improve the performance of the compiled code. For example, a JIT compiler may specialize the code based on the initialization status of a class or the state of a virtual call site [24]. However, in some cases, this VM state may change during the application's lifecycle. Therefore, the VM must explicitly track such assumptions and react accordingly whenever they are violated (e.g., by transferring execution to the interpreter and discarding the compiled code). Consequently, these optimizations restrict code reusability because the code is useful only if the assumptions about the VM state are satisfied.

In the HotSpot JVM and the GraalVM compiler, there are at least three mechanisms that enable optimization based on the VM state: (1) *assumptions* — restrictions placed on the loaded classes that may be violated by class loading (e.g., the assumption that an interface has a single implementor), (2) *speculations* — conditions verified at runtime by the compiled code (e.g., that the receiver of a call is not null), and (3) *implicit assumptions* in the compiled code that do not require a check (e.g., that a class is initialized).

Assumptions enable the JIT compiler to perform optimizations, such as turning a virtual call into a direct call. These assumptions are not checked by the compiled code but rather by the VM during class loading. Assumptions can also work well with code reuse [17]: since assumptions are explicitly tracked, we can verify whether they hold in the target VM.

```java
class Service {
    static final ThreadLocal<Service> instance = ThreadLocal.withInitial(Service::new);
    static Service getInstance() { return instance.get(); }
}
class ThreadLocal<T> { // snippet from the Java Class Library
    private static AtomicInteger nextHashCode = new AtomicInteger();
    private final int threadLocalHashCode = nextHashCode.getAndAdd(HASH_INCREMENT);
    // ...
}
```

Listing 3. Compiled code reusability is hindered due to the accidental non-determinism of thread locals.

Speculations [10] allow the JIT compiler to optimize for the common path. For example, the compiler does not have to emit code for a branch that was never taken. Instead, the compiler emits code that transfers execution to the interpreter (deoptimizes). As is the case with assumptions, speculations can also work well with code reuse. Speculations also have the property that installing code with a failed speculation does not cause incorrect behavior because the conditions are explicitly checked in the compiled code.

Finally, some assumptions in the compiled code are implicit and unchecked, as they cannot be violated in the rest of the application's lifecycle. For example, in Java, classes have static initializers that the runtime must invoke before the class is used. Some JIT compilers [15] handle class initialization by compiling code that checks the initialization status of the referenced class and invokes the initializer if necessary (an initialization barrier). The barrier is unnecessary if the referenced class is initialized at compilation time. However, omitting the barriers hinders the reusability of the code.

3.4 Relocating Object References

As explained above, we may need to relocate object references to reuse previously compiled code. The compiled code may contain references to both VM objects (e.g., representing loaded classes) and heap objects. To reuse the code, we must map the objects from the source VM (for which the code was compiled) to the target VM (where we want to reuse it). We will illustrate that it is necessary to find logically equivalent objects in the target VM obtained by following the same process rather than simply mapping the objects by content. To make the process of mapping references easier, we could limit the compiler's optimization passes so that all object constants have known provenance and are thus relocatable.

We cannot map objects naively by content because an object's identity also affects program semantics. To illustrate how object identity affects semantics, consider methods `isFoo` and `isInternedFoo` from Listing 4. For both methods, an optimizing JIT compiler (such as the GraalVM compiler) generates code that merely compares the passed string reference `string` with a constant address and returns the result of the comparison. The only difference between the two compilations is that the constants refer to two distinct strings (even though they are equal by content) because the expressions `foo` and `foo.intern()`[3] evaluate to distinct references at compilation time. As a result, the compiled methods behave differently: `isFoo("bar")` gives a different result than `isInternedFoo("bar")` if the value of the system property is `bar`.

```java
static final String foo = System
    .getProperty("foo");
boolean isFoo(String string) {
    return string == foo;
}
boolean isInternedFoo(String string) {
    return string == foo.intern();
}
```

Listing 4. The identity of the constants referenced by the compiled methods affects the semantics of the program.

The example illustrates the challenge of code reuse when the JIT compiler is allowed to perform optimizations related to constant object references. To facilitate code reuse, we can forbid the compiler from folding operations on constants. For example, we may allow the compiler to embed object references originating in static final fields, but we would remember where each constant comes from. However, we would forbid the compiler from evaluating the result of the `intern` call with a constant receiver at compilation time. As a result, every constant reference in the IR has a known provenance, making it possible to map the constants to another VM.

4 Towards Reusing Performant Code

In this section, we introduce an approach that reuses pre-optimized IR from a previous run. The goal is to save compilation resources while achieving peak performance close

[3]In Java, the `intern` method returns a canonical reference to a string with the same contents — https://docs.oracle.com/javase/specs/jls/se22/html/jls-3.html#jls-3.10.5.

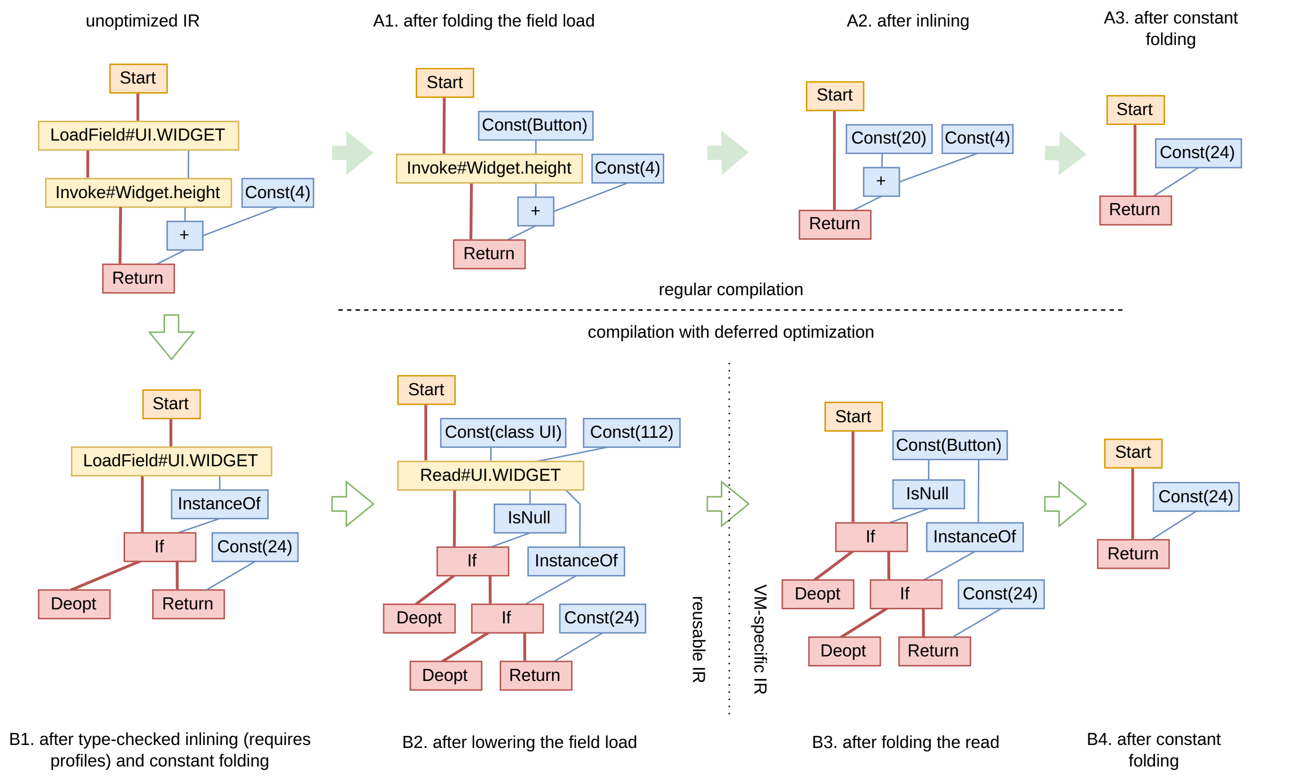

Figure 1. Graal IR [9] showing how the compiler optimizes `heightWithMargin` from Listing 5: a regular compilation (top) vs. a compilation with deferred optimization (bottom). Thick lines are control-flow edges; thin lines are data-flow edges.

```
interface Widget {
    int height();
}
class Button implements Widget {
    public int height() {
        return 20;
    }
}
// definitions of other widgets...
class UI {
    static final Widget WIDGET =
            new Button();
    static int heightWithMargin() {
        return WIDGET.height() + 4;
    }
}
```

Listing 5. A snippet of an application with a user interface.

to the code compiled from scratch. Therefore, the reusable IR is pre-optimized in such a way that it is correct to finish the compilation in another VM run. For maximum performance, the compiler may leverage profiling information and speculative optimization [10] in the reusable IR. VM-specific optimizations that would invalidate the IR for reuse are deferred to the final compilation stage for each particular VM. This stage, where we reuse the IR and apply VM-specific optimization, is later in the compilation pipeline to save as many computational resources as possible. This section describes the challenges and analyzes the approach. In Section 5, we devise an experiment to evaluate the impact on real-world workloads.

To illustrate our approach, consider the source code in Listing 5 and the method `heightWithMargin`, which computes the height of a user-interface element. Figure 1 contrasts how the compiler may optimize the code on the IR level during a regular compilation (top) compared to a compilation with deferred optimization (bottom). The figure shows Graal IR [9], which is a superposition of the control-flow and data-flow graphs: thick lines represent control-flow edges, and thin lines represent data-flow edges.

The top left IR graph in Figure 1 shows the state after parsing method `heightWithMargin` without any optimization. In a regular compilation (top of the figure), the compiler evaluates the load from the static final field. As a result, the `height` call becomes direct and inlinable. After inlining, the compiler evaluates the result of the integer addition. The final compiled code merely returns the result evaluated at compilation time.

The bottom of Figure 1 shows how the compiler optimizes the method using our approach. The IR graphs on the left of the dashed line are reusable. The compiler cannot fold the field load yet in order to avoid introducing an object reference into the IR and keep the IR reusable. Consequently, the receiver of the `height` call is not a constant, and therefore the receiving method cannot be determined at this time. However, the compiler can inline the call speculatively (as depicted in the diagram) if profiles show that the only recorded receiver type is `Button`. The nodes in the IR check that the receiver's type is `Button` — if it is not, `Deopt` transfers control to the interpreter (deoptimizes). After inlining the `height` call, the compiler evaluates the result of the integer addition. Finally, the field load is lowered to a read operation with a null check.

The IR graphs on the right of the dashed line show optimizations that leverage VM-specific information. The compiler evaluates the read operation and replaces the read node with the constant result. Since the constant is of the expected type, the null check and the type check are removed. The final IR maps to efficient machine code and exactly matches that of a regular compilation. Note that if the actual field's value were null or of a different type (impossible given the code in Listing 5), it would still be correct to reuse the IR, but the IR would not be useful as it would deoptimize immediately. The reusable IR speculates about the VM state based on profiling information, which is necessary for performance, but only in a way that maintains correctness across VM runs.

4.1 Reusing IR

The IR intended for reuse is pre-optimized only in such a way that is valid across VMs. To install the IR in a different VM, we must finish the compilation of the IR to obtain the machine code. We can do this by applying the optimizations specific to the target VM's state and then running a tail of the regular compilation pipeline.

The advantage of reusing the pre-optimized IR is that we reuse the work performed in a past compilation and can still leverage many optimization opportunities dependent on the VM-specific state. Compared to reusing compiled code directly, we still require the final compilation step, which takes some CPU and memory resources. To minimize the cost of the final compilation, we reuse the low-level IR from a late compilation stage.

The final compilation step takes less CPU time and memory the closer the cached IR is to the target machine code. However, the IR must be high-level enough so that we can still apply and benefit from the VM-specific optimizations. The concrete point in the compilation pipeline where we can store the IR depends on the compiler implementation and also on the VM-specific optimizations we defer to the final compilation step.

4.2 Deferring Optimizations

Our approach requires avoiding implicit assumptions that may not hold in a future VM run when compiling the IR intended for later reuse. In particular, we cannot embed addresses of VM objects directly in the IR because the addresses will likely change in another run. Moreover, we must defer optimizations based on the application state, such as those that depend on the contents of the heap. For example, the compiler cannot yet access fields of object constants or even the type information of the constants, as we cannot generally guarantee they will not vary between VM runs. The compiler can access the values of primitive constants from the constant pool (those not computed by a bootstrap method [12]), which do not vary unless the source code of the application changes.

Instead of embedding an address of a VM object in the IR, we can use a placeholder IR node in place of the address. Higher-level optimization passes do not typically perform optimizations based on the concrete value of an address. Therefore, this mechanism does not discard optimization opportunities. The placeholder node stores the information on how to obtain the actual value of the address so that we can relocate the IR at the time of reuse. In GraalVM, we can use the same approach to relocate method vtable entry offsets in the IR.

Optimizing JIT compilers may replace a field load by embedding the field's value in the IR if the value cannot change in the rest of the application's lifecycle. However, the field's value could differ in a future VM run. Therefore, we cannot perform this optimization while compiling the reusable IR. To capture this optimization opportunity, we defer this optimization to the final compilation step, where we can leverage the VM-specific state.

Optimization passes that comprise the tail of the regular compilation pipeline, which we apply at the time of reuse, do not require any special handling. For example, instruction selection or register allocation are part of a late compilation stage. At this stage, the compiler can make informed decisions as it can utilize the VM-specific data.

We allow speculative assumptions [10] in the reusable IR in order to not hinder peak performance. At the time of reuse, we only require an extra step that verifies the IR assumptions with the VM state. The verification is typically also part of a regular JIT compilation, so existing VMs and compilers already have the infrastructure to track and verify assumptions. In GraalVM, we must also track and verify the initialization status of the classes referenced in a compilation.

4.3 Performance of Reused IR

The deferring of optimizations may affect the optimization decisions in all optimization passes on the reusable IR. Due to the lack of field-load folding and related optimizations at the beginning of a compilation, the IR may be more complex, and the heuristics of some optimization passes may be less likely to apply a transformation. As a result, the code compiled from a cached IR may reach only reduced peak performance. The actual effect on real-world workloads is difficult to predict.

For example, folding field loads usually enables other transformations, such as constant folding or inlining. We defer field-load folding to the final compilation step in our approach. Although we can still rerun constant folding at the final stage, inlining is typically performed early in the compilation pipeline. For this reason, we miss inlining opportunities dependent on the optimizations we deferred. Moreover, the inliner considers the size of IR of a potential inlinee. Due to the deferred optimizations, the IR may be larger at this stage, which may misguide the inliner's heuristics.

5 Experimental Evaluation

In this section, we evaluate the impact of making the IR reusable by deferring optimizations to a later stage. We implemented the approach in the GraalVM compiler so that we can compare the code quality of compilations with deferred optimizations with the unmodified GraalVM compiler. This section presents an estimate of the peak-performance impact on the workloads from the Renaissance [23] benchmark suite. Moreover, we analyze the cause of the performance difference in the most negatively affected workload.

To put the measured peak-performance impact in perspective, we compare it with the client compiler [15]. The client compiler is the lower-tier compiler [27] in GraalVM, and it emits moderately optimized code. The advantage of the client compiler is that it completes the compilations much faster than the top-tier GraalVM compiler. We show that the performance of the code compiled with deferred optimization is closer to that of the unmodified GraalVM compiler than that of the lower-tier compiler.

5.1 Experimental Implementation

We created a modified version of the GraalVM compiler, which prevents the compiler from utilizing some VM-specific state until a point where we can cache the IR. At this point, instead of caching the IR, we make the VM state available to the compiler and apply optimizations that may benefit from it. Then, the rest of the regular compilation pipeline runs, and we install the code in the running VM. As a result, we obtain the code as if it were compiled from an IR cache. This way, we can estimate the peak-performance impact of reusing cached IR.

The point at which we could cache the IR is relatively late in the compilation pipeline, so there could be compilation-time savings for compilations from the IR cache. However, we do not measure the impact on the warmup, as there are a few missing pieces that would enable actual IR reuse. To achieve IR reuse, we would need to relocate and verify the equivalence of class, method, and constant references, relocate addresses of other VM objects (e.g., related to the GC), patch virtual method table offsets and similar values in the IR, verify the equivalence of the VM configuration and the target platform, and verify the speculative optimizations and assumptions in the cached IR.

The key parts of the implementation are disabling field-load folding and wrapping object constants obtained from the runtime constant pool. We disable field-load folding because values of static final fields are computed by static initializers, which could compute different values in different runs. This way, we also ensure that all object-constant references appearing in the cached IR originate in the constant pool. As a result, the relocation of constants is straightforward because we can identify each constant in the IR using a constant-pool index in the respective constant pool.

Wrapping object constants ensures that the compiler cannot use any information about them that could be different in another VM run. The wrapper for the object constants is a new kind of IR node that we added to ensure the existing optimization passes cannot use any information about the wrapped constant. After the point where we would cache the IR, we replace the wrapper nodes with regular constant nodes. Then, the rest of the compilation pipeline can leverage the VM-specific information as usual.

5.2 Measurement Setup

We measure the peak performance of GraalVM in three setups: (1) the baseline setup without any compiler modifications, (2) the GraalVM compiler modified to defer optimizations, (3) with the GraalVM compiler disabled (leaving the client compiler [15] enabled). We evaluate the peak performance based on the average iteration time (lower is better) of warmed-up workloads from the Renaissance [23] benchmark suite. For each workload, we report the average iteration time of the second and third variants relative to the baseline.

In the measured workloads, most JIT compilation activity occurs in the first few minutes of run time. Therefore, we run each workload for 30 minutes and remove the first half of the measured iteration times as warmup. We filter out the samples outside three standard deviations from the mean and compute each run's arithmetic mean of the iteration time. For each workload and setup, we repeat 40 runs and compute the arithmetic mean of the per-run means. The number of runs is selected based on our available machine time. We report 99% confidence intervals computed using hierarchical bootstrap resampling — we first sample from the runs and then from the iterations of the sampled runs.

We conducted the experiments on 20 identical machines with the Intel Xeon E3-1230 v6 CPU, 32 GB of RAM, Fedora 35 with Linux kernel version 5.16.11, and Renaissance version 0.15.0. We used a developer build of GraalVM Enterprise based on OpenJDK 22.

Table 1. Slowdown ratios of the GraalVM compiler with deferred optimization and the client compiler compared to the GraalVM baseline with 99% confidence intervals.

benchmark	deferred optimization		client compiler	
scala-stm-bench7	1.00	0.99–1.01	7.25	7.19–7.31
reactors	1.04	1.03–1.05	10.09	10.02–10.16
future-genetic	1.05	1.03–1.07	12.69	12.53–12.87
akka-uct	1.05	1.04–1.05	20.95	20.69–21.21
fj-kmeans	1.06	1.06–1.06	51.67	51.37–51.98
philosophers	1.07	1.04–1.10	18.82	18.50–19.16
page-rank	1.08	1.06–1.09	18.11	17.93–18.30
movie-lens	1.11	1.10–1.11	17.30	17.24–17.36
als	1.11	1.11–1.12	54.07	53.46–54.69
scala-doku	1.14	1.07–1.22	12.68	12.04–13.44
rx-scrabble	1.15	1.14–1.16	6.77	6.71–6.83
neo4j-analytics	1.16	1.15–1.16	22.80	22.67–22.92
finagle-http	1.17	1.16–1.18	12.28	12.22–12.35
dotty	1.18	1.18–1.18	16.44	16.30–16.59
log-regression	1.23	1.22–1.24	32.02	31.75–32.30
dec-tree	1.23	1.23–1.24	19.55	19.48–19.63
scrabble	1.28	1.25–1.30	31.31	30.93–31.74
chi-square	1.28	1.25–1.30	50.78	50.18–51.27
finagle-chirper	1.30	1.29–1.32	17.71	17.58–17.83
scala-kmeans	1.32	1.32–1.33	14.04	13.87–14.23
par-mnemonics	1.63	1.50–1.78	17.64	16.51–18.97
mnemonics	1.68	1.55–1.82	12.49	11.64–13.41
naive-bayes	1.86	1.81–1.90	58.01	56.65–58.84
gauss-mix	5.62	5.00–6.17	181.88	161.17–201.73

5.3 Measurement Results

Table 1 reports the slowdown ratios of each workload. In the middle, we report the mean iteration time of the GraalVM compiler with deferred optimization divided by the mean iteration time of the unmodified GraalVM compiler baseline, including 99% confidence intervals. On the right, we show the mean iteration time of the client compiler relative to the unmodified GraalVM compiler baseline with respective 99% confidence intervals.

The effect of deferring optimizations in the GraalVM compiler ranges from no impact in scala-stm-bench7 to a 5.62x slowdown in gauss-mix. For all workloads but gauss-mix, the slowdown ratio is below 2. We analyze the cause of the regression in gauss-mix in Section 5.4.

Although deferring optimizations causes an overall performance regression compared to the unmodified GraalVM compiler, the performance of the compiled code is still much better than that of the client compiler. The workloads compiled with the client compiler exhibit significant slowdowns — they are from about 7 to 182 times slower. This data clearly shows that the quality of the code compiled with deferred optimizations is much closer to the code quality of the highly optimizing GraalVM compiler than that of the lower-tier compiler.

5.4 Code Quality

The regression in the most negatively affected workload, gauss-mix, is due to a failure to optimize method handles. A method handle [21] is an invokable method reference in Java. In a typical compilation, the GraalVM compiler attempts to transform (intrinsify) method-handle invocations to regular method invocations. The necessary condition is that the target of the method handle is an object constant, which the compiler can resolve to an invokable method. This procedure only partially succeeds with deferred optimization: although the compiler resolves the target method of the method handle after field-load folding is allowed, it is too late to inline the target method.

In the IR during compilation, there is a constant holding an instance of a method handle with the final field `member`, which identifies the target method. The compiler can intrinsify the method-handle invocation only if the target method is constant, i.e., the compiler must first fold the load of the `member` field to a constant node. With deferred optimization, this happens later in the compilation pipeline. However, the GraalVM compiler inlines very early, so it is too late to inline the target method. We analyzed the optimization decisions in the workload combined with a sampling profiler; the hottest compilation units were those the compiler failed to inline. This is a clear sign of a performance regression caused by inlining.

6 Conclusion

This paper investigated the challenges of reusing compiled code in a modern VM. The problems are related to the non-determinism of the running application, non-determinism of the runtime environment, and aggressive optimizations based on the state of the VM. Intending to reuse previous compilation results with minimal peak-performance impact, we introduced the approach of reusing IR with deferred optimization. Unlike previous work, the compiled code leverages VM-specific optimizations performed on the reusable IR. While deferred optimization in a high-performance compiler can reduce performance (leading to an up to 6x slowdown in the most affected workload), it still significantly outperforms the lower-tier compiler. These results suggest the approach could be valuable for improving warmup times in specific workloads.

Acknowledgments

This research was supported by projects SVV 260 698 and GAUK 74824.

References

[1] Azul. 2024. *Cloud Native Compiler.* Retrieved May 20, 2024 from https://docs.azul.com/optimizer-hub/about/cloud-native-compiler

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (2017), 27 pages. https://doi.org/10.1145/3133876

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

[4] OpenJDK Community. 2024. *Project CRaC.* Retrieved May 20, 2024 from https://openjdk.org/projects/crac

[5] OpenJDK Community. 2024. *Project Leyden.* Retrieved May 20, 2024 from https://openjdk.org/projects/leyden

[6] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. 2002. Code Sharing among Virtual Machines. In *ECOOP 2002 — Object-Oriented Programming*, Boris Magnusson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–177. https://doi.org/10.1007/3-540-47993-7_7

[7] Irwin D'Souza. 2018. *Ahead Of Time Compilation: Relocation.* Retrieved May 21, 2024 from https://blog.openj9.org/2018/10/26/ahead-of-time-compilation-relocation/

[8] Irwin D'Souza. 2018. *Ahead Of Time Compilation: Validation.* Retrieved May 21, 2024 from https://blog.openj9.org/2018/11/08/ahead-of-time-compilation-validation/

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

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

[11] Ben Evans. 2020. *Behind the scenes: How do lambda expressions really work in Java?* Retrieved June 24, 2024 from https://blogs.oracle.com/javamagazine/post/behind-the-scenes-how-do-lambda-expressions-really-work-in-java

[12] Brian Goetz. 2018. *JEP 309: Dynamic Class-File Constants.* Retrieved July 15, 2024 from https://openjdk.org/jeps/309

[13] IBM. 2024. *Introduction to class data sharing.* Retrieved May 21, 2024 from https://eclipse.dev/openj9/docs/shrc/

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

[15] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[16] Han B. Lee, Amer Diwan, and J. Eliot B. Moss. 2007. Design, implementation, and evaluation of a compilation server. *ACM Trans. Program. Lang. Syst.* 29, 4 (August 2007), 18–es. https://doi.org/10.1145/1255450.1255451

[17] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (2023), 22 pages. https://doi.org/10.1145/3622839

[18] Oracle. 2022. *Garbage-First (G1) Garbage Collector.* Retrieved June 25, 2024 from https://docs.oracle.com/en/java/javase/18/gctuning/garbage-first-g1-garbage-collector1.html

[19] Oracle. 2024. *GraalVM Compiler.* Retrieved June 27, 2024 from https://www.graalvm.org/latest/reference-manual/java/compiler/

[20] Oracle. 2024. *Native Image.* Retrieved May 20, 2024 from https://www.graalvm.org/latest/reference-manual/native-image/

[21] Ed Ort. 2009. *New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine.* Retrieved June 25, 2024 from https://www.oracle.com/technical-resources/articles/javase/dyntypelang.html

[22] Andrej Pečimúth. 2023. Remote Just-in-Time Compilation for Dynamic Languages. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Cascais, Portugal) *(SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3618305.3623593

[23] 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. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 17. https://doi.org/10.1145/3314221.3314637

[24] John Rose. 2013. *Virtual Calls (HotSpot).* Retrieved June 25, 2024 from https://wiki.openjdk.org/display/HotSpot/VirtualCalls

[25] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[26] Sun Microsystems. 2006. *The Java HotSpot Performance Engine Architecture.* Retrieved July 6, 2024 from https://www.oracle.com/java/technologies/whitepaper.html

[27] Igor Veresov. 2013. *Tiered Compilation in Hotspot JVM.* Retrieved July 6, 2024 from https://www.slideshare.net/maddocig/tiered

[28] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[29] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. 2, OOPSLA, Article 124 (2018), 23 pages. https://doi.org/10.1145/3276494