Understanding and Finding JIT Compiler Performance Bugs

ZIJIAN YI, The University of Texas at Austin, USA
CHENG DING, The University of Texas at Austin, USA
AUGUST SHI, The University of Texas at Austin, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

Just-in-time (JIT) compilers are key components for many popular programming languages with managed runtimes (e.g., Java and JavaScript). JIT compilers perform optimizations and generate native code at runtime based on dynamic profiling data, to improve the execution performance of the running application. Like other software systems, JIT compilers might have software bugs, and prior work has developed a number of automated techniques for detecting functional bugs (i.e., generated native code does not semantically match that of the original code). However, no prior work has targeted JIT compiler performance bugs, which can cause significant performance degradation while an application is running. These performance bugs are challenging to detect due to the complexity and dynamic nature of JIT compilers. In this paper, we present the first work on demystifying JIT performance bugs. First, we perform an empirical study across four popular JIT compilers for Java and JavaScript. Our manual analysis of 191 bug reports uncovers common triggers of performance bugs, patterns in which these bugs manifest, and their root causes. Second, informed by these insights, we propose layered differential performance testing, a lightweight technique to automatically detect JIT compiler performance bugs, and implement it in a tool called Jittery. We incorporate practical optimizations into Jittery such as test prioritization, which reduces testing time by 92.40% without compromising bug-detection capability, and automatic filtering of false-positives and duplicates, which substantially reduces manual inspection effort. Using Jittery, we discovered 12 previously unknown performance bugs in the Oracle HotSpot and Graal JIT compilers, with 11 confirmed and 6 fixed by developers.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Software performance**; **Software testing and debugging**.

Additional Key Words and Phrases: JIT Compilers, performance testing, compiler testing

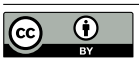
ACM Reference Format:
Zijian Yi, Cheng Ding, August Shi, and Milos Gligoric. 2026. Understanding and Finding JIT Compiler Performance Bugs. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 137 (April 2026), 29 pages. https://doi.org/10.1145/3798245

1 Introduction

Modern optimizing compilers are responsible for generating both *correct* and *efficient* code [2, 21]. An optimizing compiler takes a program as an input (p) and produces a semantically equivalent program in another form (e.g., intermediate representation or native code), i.e., $compiler(p) = p'$.

Just-in-time (JIT) compilation is a technique used by many programming language implementations to improve performance *during* program execution. It originates from Smalltalk [25], and has been widely adopted in many popular programming languages like Java [22], JavaScript [87], and Python [71, 79]. Unlike ahead-of-time (AOT) compilation, JIT compilation happens during the

Authors' Contact Information: Zijian Yi, The University of Texas at Austin, , USA, zijianyi@utexas.edu; Cheng Ding, The University of Texas at Austin, , USA, cheng.ding@utexas.edu; August Shi, The University of Texas at Austin, , USA, august@utexas.edu; Milos Gligoric, The University of Texas at Austin, , USA, gligoric@utexas.edu.



ACM 2475-1421/2026/4-ART137
https://doi.org/10.1145/3798245

execution of a program. An input to a JIT compiler is an intermediate representation of (or part of) a program (a) and the output is commonly native code, i.e., $jit(a) = o$. JIT compilers collect profiling data while the program is running and use this data to guide the compilation process. They selectively compile code that is frequently executed and worth optimizing for future execution, so that the performance gain from JIT compilation outweighs the compilation overhead [45]. Speculative optimizations (i.e., optimizations that are only applicable under certain assumptions) may also be performed based on the profiling data [3, 15, 99].

Like many software systems, a JIT compiler can itself have unintended software bugs. These bugs can be broadly categorized into functional bugs or performance bugs. A functional bug in the JIT compiler is when the compiler does not produce code that matches the semantics of the original code, i.e., we say that a JIT compiler has a functional bug if $semantics(a) \neq semantics(jit(a))$. Finding functional bugs in JIT compilers has been a popular topic in recent years with many notable results [19, 36, 39, 51, 69, 72, 88, 89, 95–97]. On the other hand, we say that a JIT compiler has a *performance bug* when it delivers unexpected performance behavior on certain input programs. We distinguish two cases. First, the JIT compiler itself takes an excessive amount of time to perform its task, i.e., $time(jit(a)) > threshold$, where the threshold denotes an upper bound on the expected compilation time as defined by users or developers. We refer to this kind of performance bug as *long compilation* in this paper (also referred to as "compile time hog" in some literature). Second, the JIT compiler produces code that takes too long to execute, i.e., $time(run(a)) < time(run(o))$. Namely, we observe an unexpected performance behavior of the JIT compiled program compared to an unoptimized version of that code. We refer to this as *high-order performance bugs*, where the root causes of performance degradation come not from the program a itself but from the compiler that transforms the program. Since JIT compilation happens at runtime, both *long compilation* and *high-order performance bugs* can lead to similar effects. There is little prior work on understanding and detecting performance bugs in JIT compilers.

Existing research on performance bugs [4, 37, 42, 74, 94] mostly focuses on performance bugs in application code that can be fixed by modifying the application source itself. In contrast, performance bugs in JIT compilers cannot be easily, sometimes not even possibly, resolved at the application level. While certain workarounds at the source-code level can make specific code patterns more amenable to JIT optimizations [32], it remains desirable to improve JIT compilers themselves to make them more flexible and robust, thereby reducing the burden on developers. There has also been prior work on identifying *long compilation* bugs in C [46] and Markdown [52] compilers and on reducing compilation time in C++ [1, 27], as well as studies on *high-order performance bugs* that primarily target missed optimizations in AOT compilers [6, 30, 55, 80, 81]. One study that explores JIT compilers [68] largely treats them as AOT compilers, focusing solely on optimization-related issues. However, JIT compilers exhibit several unique aspects like profiling, speculation, and interactions with other runtime components that are not present in AOT compilers. These aspects can also introduce performance bugs, yet they remain unexplored by prior work.

To bridge this gap in understanding JIT compiler performance bugs, we present the first in-depth empirical study of JIT compiler performance bugs from real-world Java and JavaScript issue trackers. Our analysis of 191 bug reports reveals common triggers, patterns in which these bugs manifest, and their root causes. Specifically, we find that (1) nearly half of the bugs can be exposed by small, focused micro-benchmarks rather than full benchmark suites; (2) they are most commonly detected through comparative signals such as performance regressions or differences across equivalent executions; and (3) beyond traditional optimization and code generation issues, JIT-specific components like speculation and runtime interaction are also significant sources of bugs. These characteristics provide insights on how to design better testing and development methods to detect and combat performance bugs in JIT compilers.

Based on these insights, we propose Jittery, a tool that implements *layered differential performance testing* to automatically detect JIT compiler performance bugs. Jittery generates a large number of small random programs, executes each under two differential JIT configurations (e.g., different compiler tiers or compiler versions), and flags programs whose execution times diverge significantly as potential bug candidates. Since rigorous benchmarking of every generated program is prohibitively expensive, we organize detection into a series of layers with increasing iteration counts: early layers cheaply discard programs that show no performance anomaly, while later layers apply more thorough measurement only to the surviving candidates. This lets us test many programs quickly (efficiency) while still obtaining reliable measurements for promising candidates (accuracy). Jittery further leverages runtime information from earlier layers to prioritize candidates for subsequent layers. Finally, it uses heuristics to remove potential false positives and duplicates, which greatly reduce manual inspection effort. Jittery leads to the discovery of 12 previously unknown performance bugs in the Oracle HotSpot and Graal JIT compilers, with 11 confirmed and 6 already fixed by the developers.

The main contributions of this paper include the following:

- ⋆ **Empirical study**. The first in-depth empirical study of real-world JIT compiler performance bugs. We analyze 191 JIT compiler performance bugs in 4 widely used JIT compilers and summarize their characteristics. Based on our empirical study, we provide insights on challenges and potential solutions for detecting and avoiding JIT compiler performance bugs.
- ⋆ **Dataset**. We make a dataset of JIT compiler performance bugs, which we built during our empirical study, publicly available. We expect that this dataset can be used to help future work to develop techniques for detecting and preventing JIT compiler performance bugs.
- ⋆ **Approach**. We developed Jittery, a lightweight tool based on layered differential performance testing to automatically detect JIT compiler performance bugs with practical optimizations to further improve efficiency and reduce manual effort.
- ⋆ **Results**. Jittery led to the discovery of 12 previously unknown performance bugs in the Oracle HotSpot and Graal JIT compilers, including some involving aspects unique to JIT compilers like speculation. 11 of these bugs have been confirmed, with some already fixed.

Our dataset and scripts are publicly available at https://github.com/EngineeringSoftware/jittery.

2 Background

This section provides background on just-in-time (JIT) compilers, especially focusing on unique features of JIT compilers that differentiate them from ahead-of-time (AOT) compilers. We discuss a brief overview of JIT compilers (Section 2.1), speculative optimization (Section 2.2), and tiered compilation (Section 2.3).

2.1 Overview of JIT Compilers

Figure 1 shows the overview of the JIT compilation process for general-purpose programming languages like Java and JavaScript. Program execution begins in the interpreter (①). While the interpreter runs the program, the profiler (②) collects profiling data, e.g., counters recording the number of times a method is invoked or a loop is entered. Once profiling thresholds are reached, the JIT compiler (③) is triggered to compile any "hot code" (Ⓐ), i.e., code regions that have been frequently executed and would likely continue to be frequently executed in the future. The JIT compiler then performs optimization and code generation for this hot code, following steps similar to those in AOT compilers, but often incorporating speculative optimizations guided by the collected profiling data (Section 2.2). Furthermore, modern runtimes often contain more than one

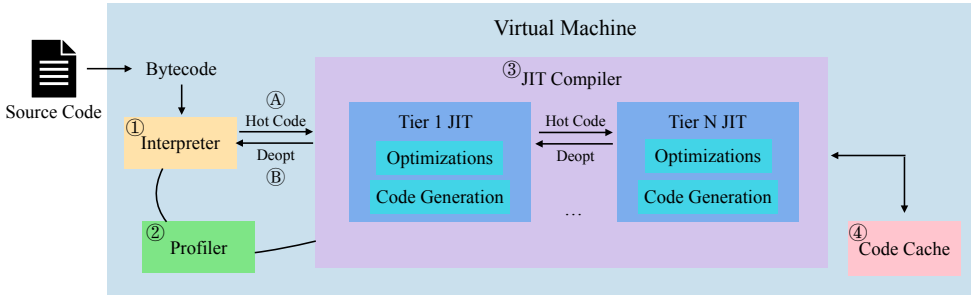


Fig. 1. High-level workflow of running a program through JIT compilation, which happens while the program is running. Commonly, JIT compilers optimize code snippets that are frequently executed.

```
1  if (obj == null) {
2      throw new NullPointerException();
3  }
4  val = obj.f;
```

(a)

```
1  val = obj.f;
2
3  // If 'obj' is null
4  // A signal is handled by JVM
```

(b)

Fig. 2. Example of a null-check elimination speculative optimization in HotSpot: (a) a common null check code pattern in Java source code; (b) the pseudocode generated by HotSpot with null-check eliminated. The optimization removes the null check and uses an implicit guard when speculation fails.

JIT compiler with different trade-offs (Section 2.3). Because speculations may occasionally fail to hold, execution must sometimes revert to the interpreter or a lower tier JIT compiler through a process known as deoptimization (Ⓑ) [99]. Finally, since native code is generated dynamically at runtime, the JIT compiler must carefully manage the memory region used to store compiled code, commonly referred to as the code cache, ensuring that other parts of the code invoke and use the correct compiled code.

There are two major types of decisions a JIT compiler needs to make based on the collected profiling data: 1) what code to compile, and 2) what speculative optimizations to apply. In practice, both decisions are typically guided by hand-crafted heuristics derived from empirical experience and human expertise [15, 45].

Let the time of compiling a given code region be denoted as C, time to execute that piece of code in the interpreter mode as T_i, time to execute the compiled code as T_c, and the expected number of future executions as N. For compilation to be profitable, the following condition should hold:

$$(T_i - T_c) \times N > C \tag{1}$$

2.2 Speculative Optimization

Speculative optimization is a technique where JIT compilers make educated guesses about the behavior of a program based on runtime profiling data. Unlike optimizations in AOT compilers, whose correctness is ensured by static analysis, speculative optimizations may not always hold true and they are guarded by runtime checks. When a speculation fails, the execution falls back to a safe state through *deoptimization*.

Figure 2 illustrates an example of speculative optimization for null check elimination in the HotSpot [44]. The source code in Figure 2a shows a common code pattern in Java, where field access is preceded by an explicit null check. In most programs, it is rare for the null check to fail, which can be confirmed based on the profiling data collected after executing that part of the code many times.

```
1  function add(a, b) {
2    return a + b;
3  }
4  for (let i = 0; i < 100000; i++) {
5    add(i, i + 1);
6  }
7  console.log(add("Hello ", "world"));
```

(a)

```
1  function add(a, b) {
2    if (!is_smi(a) || !is_smi(b)) {
3      deoptimize();
4    }
5    return raw_numeric_add(a, b);
6  }
```

(b)

Fig. 3. Example of type specialization for add operator speculative optimization in V8: (a) add operator is overloaded for many different types in JavaScript source code; (b) the pseudocode generated by V8 with type specialization based on profiling data. The optimization specializes the add operator for small integer addition and uses an explicit guard to ensure the assumption holds.

The JIT compiler then speculates that the check will always succeed (without formally proving it), generating code similar to the pseudocode shown in Figure 2b. It will directly access the field without any additional checks. If a null pointer dereference is ever encountered, the operating system will send a signal to execution runtime, whose handler performs deoptimization and resumes execution in a safe state. This form of speculative optimization can improve performance by eliminating redundant null checks, while also reducing compilation time and code size by avoiding the need to generate code for exception creation, throwing, and handling.

Speculative optimization is more extensive in dynamically typed languages like JavaScript. Figure 3 shows an example of speculative optimization in V8 [59]. The source code in Figure 3a shows a simple `add` function. Since the '+' operator is overloaded for multiple types, a general code generation strategy would be to generate code that checks the types of the operands at runtime. However, if the profiling data shows that the function is frequently called with small integers (`smi`), V8 can speculate that the operands are always of type `smi` and specialize code that only provides fast path for the case as shown in Figure 3b. When this speculation fails, e.g., if a call is made with a non-integer operand, the generated code triggers a deoptimization event. During deoptimization, the runtime system reconstructs the corresponding interpreter or baseline stack frame from metadata embedded in the optimized code, discards the invalid optimized frame, and transfers control back to the interpreter or a less-optimized version of the code. These steps ensure that execution can safely continue without violating the program's semantics.

2.3 Tiered Compilation

Compilation at runtime takes time and resources, so trade-offs have to be made between the compilation overhead and the efficiency of the generated code. For example, for simple functions, several basic optimizations are enough to generate efficient code. However, for complex and frequently invoked functions, more sophisticated optimizations are needed to gain better performance.

To balance these trade-offs, modern JIT compilers often rely on *tiered compilation*. The code under execution transitions between different levels of execution based on the profiling data and the runtime environment, and the JIT chooses the appropriate compilation and profiling strategies for different execution levels. For example, in HotSpot, there are 5 execution levels [64] and two JIT compilers: C1 and C2 [65]. C1 (a.k.a., client compiler) is a low overhead compiler that generates code quickly but with fewer optimizations. C2 (a.k.a., server compiler) is a high overhead compiler that generates highly optimized code. Figure 4 shows the transitions between different execution levels. Execution starts in the *interpreter* (L0). When a method becomes hot, policy heuristics first decide between L2 and L3, with C2 queue pressure as a key signal: under high C2 load, L2 is preferred; otherwise L3 is chosen to collect richer profiling. The C1 queue length also adjusts

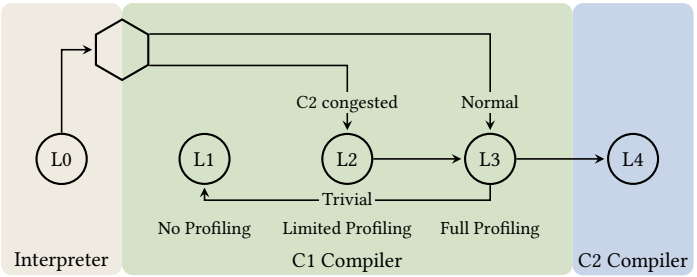


Fig. 4. Common transitions among execution levels (L0–L4) in HotSpot tiered compilation. Execution starts in the interpreter (L0), then typically moves to L2 or L3 based on queue/load heuristics; methods with sufficient profiling in L3 are promoted to L4, while trivial methods may end at L1.

thresholds to avoid overloading the compiler. Methods running at L2 can later move to L3 when C2 pressure recedes. Once enough profiling is collected at L3, methods are usually promoted to L4 (C2) for peak performance. For trivial methods, the policy may choose L1 (optimized C1 code without profiling) instead of promoting to L4. SpiderMonkey has two tiers of JIT compilers, namely Baseline Compiler and WarpMonkey [62]. V8 recently also introduced two new JIT compilers featuring faster compilation time named Maglev [86] and Sparkplug [85], besides the existing JIT compiler: TurboFan [87].

Consider speculative optimization and its potential benefits. Let $p \in [0, 1]$ denote the probability that a speculation holds at runtime. Let $G > 0$ represent the reduction in execution time (e.g., in milliseconds or cycles) achieved when the speculation is correct, and let $D > 0$ denote the additional execution-time overhead incurred due to deoptimization when the speculation fails. Under these definitions, the speculation is expected to be beneficial when the expected time saved outweighs the expected penalty:

$$p \times G > (1 - p) \times D. \tag{2}$$

3 Study Methodology

In this section, we describe our empirical study methodology. We describe how we collect bug reports of interest (Section 3.1), the way we analyze these reports (Section 3.2), the research questions we aim to answer (Section 3.3), and potential threats to validity (Section 3.4) of our study.

3.1 Bug Collection

We choose four subjects for our study: HotSpot, Graal, V8, and SpiderMonkey. The former two are Java JIT engines used in Oracle JDK, V8 is the JavaScript engine in the Chrome browser, and SpiderMonkey is the JavaScript engine in the Firefox browser. They all follow the basic architecture as described in Section 2.1. These subjects are widely-used and represent two kinds of programming languages where JIT compilers are commonly used, namely statically-typed Object-Oriented (e.g., Java) and dynamically-typed (e.g., JavaScript) languages. These subjects are representative of the state-of-the-art JIT compilers.

All of the subjects are open-sourced and have open issue tracking systems. Besides Graal, which uses GitHub issue tracker [63], the other three all have their own dedicated issue tracking systems [35, 61, 66]. We systematically collected issues in the issue tracking systems based on the following criteria:

- **Component**: We only consider performance issues that are related to JIT compilers. Other components like garbage collectors may also cause performance issues but they are out of scope

Table 1. Subjects used in our empirical study.

Subject	Language	Collected	Filtered				Studied
			Irrelevant	Enhancement	Duplicate	Not an Issue	
HotSpot	Java	111	11	8	5	0	87
Graal	Java	39	2	2	1	16	18
V8	JavaScript	75	4	8	0	12	51
SpiderMonkey	JavaScript	47	1	4	0	7	35
Total	-	272	18	22	6	35	191

of our study. Note that if JIT compilers cause performance issues in other components, we still consider them.

- **Type**: We only consider issues that are labeled as 'bug' or 'defect'. These issues usually represent problems that manifest in practice and are reported by users. There are also issues labeled as 'enhancement' or 'feature' which are not considered in our study. Besides, we only consider issues that are marked as 'fixed' by the developers as they are of higher priority and contain more details that help us understand the issues.
- **Date**: To narrow down the scope of our study, we only consider issues that are created after 2015 and fixed before 2025. The time range is chosen to ensure that the issues are recent so that they reflect the current state of the JIT design and fixes are merged into the codebase.

To be precise about our steps, we give an example. The JDK Bug System [66] provides a query language for filtering issues, and we use the following query to search for issues of interest in HotSpot:

```
labels = performance AND type = Bug AND Resolution = fixed AND
  created >= 2015-01-01 AND resolved <= 2025-01-01 AND
  component = hotspot AND subcomponent in (compiler, jit)
  ORDER BY created ASC
```

Other issue tracking systems provide similar filtering or query machinery to search for issues conforming with our criteria. We found in Graal [63] many performance issues that are not explicitly labeled as 'performance', and thus we use a set of keywords (i.e., 'perf', 'performance', 'slow', 'slower', 'opt', 'optimization', 'optimisation', 'latency', 'missed', 'missing', 'high', 'higher') to search for potential issues of interest.

3.2 Bug Filtering and Analysis

After the initial automated search, we collected a total of 272 issues. We then performed a manual review to further refine the dataset. During this process, we removed issues that were mis-labeled (*Irrelevant*), too vague or proposal-oriented (*Enhancement*), duplicates of other reports (*Duplicate*), or closed without a fix (*Not an issue*). Such cases were treated as feature requests rather than genuine performance bugs revealed in practice. After this manual filtering, 191 issues remained for our study.

Table 1 shows the final number of performance bugs we collected and studied for each subject. The first column lists the studied subjects, followed by the programming language, the initial number of collected bugs, the number of bugs filtered out by category, and the final count of studied bugs. The last row reports the total across all subjects.

```java
public static void test(long n) {
  while (n > 0) {
    for (int i = 0; i < size; i++) {
      // Can be vectorized
      oBuf[i] = (short)(iBuf[i] >> 3);
    }
    n--;
  }}
```

(a) test case

```
bool ciMethodData::load_data() {
   // ...
+  if (_invocation_counter == 0
+       && mdo->backedge_count() > 0) {
+      _invocation_counter = 1;
+  }
   _state = ... // ...
}
```

(b) fix

Fig. 5. Case study of JDK-8280320: loop optimizations are missing during On-Stack Replacement (OSR) compilation. (a) A test case that triggers OSR compilation but is not properly optimized; (b) A fix that avoids skewing counter data during OSR compilation. The bug is caused by incorrectly set profiling data, which prevents the vectorization optimization in HotSpot.

For each of the remaining issues, we conducted an in-depth manual inspection of all related information provided in each bug report, including the bug description, developer discussions, attached files, and associated fix commits or pull requests.

3.3 Research Questions

The goal of our study is to gain a deeper understanding of performance bugs in JIT compilers in practice and to provide actionable insights that can help developers detect and prevent such bugs more effectively. To this end, we aim to answer the following research questions:

- **RQ1:** What input artifacts are used to trigger performance bugs?
- **RQ2:** What are common symptoms of performance bugs?
- **RQ3:** What are common root causes of performance bugs?

Answering RQ1 helps us understand how to design effective test inputs for detecting performance bugs. Addressing RQ2 reveals the observable symptoms that can guide the construction of reliable test oracles. Finally, exploring RQ3 sheds light on which components of JIT compilers are more error-prone and how similar issues can be mitigated in future development.

3.4 Threats to Validity

Like other empirical studies, our work is subject to both external and internal threats.

External threats concern the representativeness and generalizability of our subjects and studied issues. We selected the most widely used JIT compilers across two representative programming languages and collected issues reported over the past ten years to ensure recency and relevance. We believe these criteria make our dataset broadly representative of real-world JIT compiler behavior.

Internal threats primarily stem from manual issue analysis. To mitigate bias, we defined clear classification criteria with illustrative examples and considered all available contextual information. Each issue was independently analyzed by at least two authors, with disagreements resolved through discussion. We also make our dataset publicly available to facilitate independent verification. These measures help reduce internal threats and enhance the reliability of our findings.

4 Case Study

In this section, we present motivating examples of real-world JIT compiler performance bugs to illustrate their unique characteristics. Section 5 then provides a more systematic analysis of these characteristics.

Case 1: Inaccurate profiling data. **JDK-8280320**[1] (Figure 5). This bug shows a missed loop optimization during On-Stack Replacement (OSR) compilation [24] in HotSpot. It was discovered while benchmarking AVX-related tests on x86 server machines: the release build exhibited approximately 35× lower throughput than the fastdebug build, because the compiled code fell back to scalar instructions instead of the expected AVX instructions. Figure 5a shows a reduced test case. The issue arises when misleading profiling data causes OSR compilations to incorrectly skip loop optimizations. Specifically, the invocation counter remains at zero even when the loop's backedge counter indicates active execution, leading the compiler to underestimate loop hotness and disable key optimization passes. The fix, shown in Figure 5b, correctly initializes the profiling counter to ensure accurate scaling of execution counts and restores the expected optimization behavior.

Observation 1: Inaccurate profiling data in JIT compilers can cause missed optimizations even when the optimization pipeline itself is correctly implemented.

Case 2: Long compilation. **Graal-2548**. This bug occurs in the Spring Framework when the Graal JIT compiler attempts to compile the `BCrypt::key` method. Due to a defect in the compiler's optimization logic, the compilation process enters an infinite loop, causing the compilation thread to run indefinitely with full CPU utilization. In contrast to AOT compilers–where such a problem would surface during the build process–this issue only manifests at runtime, making it significantly harder to detect and diagnose without inspecting the JIT compiler's internal logs. The problem was observed in the GraalVM EE 20.1.0 release, and at the time of discovery, no publicly available fix was provided.

Observation 2: Similar types of performance bugs can be substantially harder to detect in JIT compilers than in AOT compilers, because they manifest dynamically during program execution and may cause severe runtime performance degradation.

Case 3: Performance difference. **V8-42210048** (Figure 6). This bug exposes a severe performance cliff in V8's TurboFan compiler when subclassing the `Object` constructor in JavaScript. Specifically, instance creation becomes 3×–4× slower when a class explicitly extends `Object` compared to a class without an `extends` clause. The issue was identified using a microbenchmark that compares three variants: a base class without inheritance, a class directly extending `Object`, and a class extending `Object` through a factory function (Figure 6a). The slowdown is caused by inefficient handling of class inheritance during object construction when the superclass is `Object`. The fix enhances `JSCallReducer` to detect `JSConstruct` nodes targeting the `Object` constructor and to optimize them into more efficient `JSCreate` nodes under specific conditions, as shown in Figure 6b.

Observation 3: Semantically equivalent/similar programs can exhibit huge performance differences, and such differences may indicate performance bugs in JIT compilers. Checking for such differences can act as a useful test oracle for detecting performance bugs.

Case 4: Speculation and recompilation loop. **JDK-8257594**. This issue demonstrates a performance bug observed through compiler log outputs, where the JIT compiler repeatedly emits `uncommontrap` events, indicating it is stuck in a cycle of deoptimization and recompilation. In this scenario, the compiler repeatedly invalidates optimized code when a speculative assumption fails (e.g., a failed `checkcast`), falls back to interpreter execution, and then recompiles the same method again under the same incorrect assumptions. This process–known as a recompilation loop–continues until the recompilation limit is reached, wasting CPU cycles and preventing stable optimized code generation. The root cause is an overly aggressive speculative decision in handling invalid type checks, leading to persistent deoptimizations. Such loops can severely impact application throughput and overall runtime efficiency.

[1]All the bug labels are clickable and redirect to related pages.

```
var baseNoExtends = (function() {
    const A = class A {};
    return function baseNoExtends() { return new A;} })();
var baseExtendsObject = (function() {
    const A = class A extends Object {};
    return function baseExtendsObject() { return new A;} })();
var baseExtendsViaFactory = (function() {
    const A = (BaseClass => class A extends BaseClass {})(Object);
    return function baseExtendsViaFactory() { return new A;} })();
var TESTS = [baseNoExtends, baseExtendsObject, baseExtendsViaFactory];
function test(fn) { ... }
for (var j = 0; j < TESTS.length; ++j) { test(TESTS[j]); }
```

(a) test case

```
+ if (*function == function->native_context()->object_function()) {
+     if (arity == 0) {
+         NodeProperties::ChangeOp(node, javascript()->Create());
+         return Changed(node);
+     }
+     HeapObjectMatcher mnew_target(new_target);
+     if (mnew_target.HasValue() && *mnew_target.Value() != *function) {
+         for (int i = arity; i > 0; --i) node->RemoveInput(i);
+         NodeProperties::ChangeOp(node, javascript()->Create());
+         return Changed(node);
+     }
+ }
```

(b) fix

Fig. 6. Case Study of V8-42210048: performance degradation in JavaScript Object constructor subclassing. (a) A test case that evaluates the performance impact of three alternative class initialization strategies, revealing observable performance differences. (b) A fix that adds support to the JSCallReducer to recognize JSConstruct nodes where the target is the Object constructor, and reduce them to JSCreate nodes. The bug is caused by the lack of optimization for Object constructor calls in subclassing scenarios.

Observation 4: JIT compilers can become trapped in self-reinforcing recompilation loops caused by flawed speculative assumptions, consuming significant resources without the ability to converge to a stable optimized state.

Case 5: Interaction with garbage collector. **V8-42209849**. This issue reveals a performance regression of up to 10× in V8 version 6.1, observed through benchmarks of a router library. V8 maintained a per-context weak list of all optimized JavaScript functions, used primarily for code unlinking during deoptimization. Although individual entries were weak references (i.e., they did not prevent the referenced functions from being collected), the garbage collector still had to traverse the *entire* list on every scavenge (minor GC) cycle to update pointers and remove dead entries. In applications that created many closures—such as router libraries that generate closures for each registered route—this list grew large, making the per-scavenge traversal cost increasingly expensive and dominating overall GC time. The fix removed the weak list entirely and replaced the eager code unlinking with a lazy scheme: a deoptimization mark is now checked in the prologue of each code object upon its next activation, eliminating the need for the costly per-GC-cycle list traversal.

Observation 5: The intricate interaction between JIT compilers and runtime subsystems such as garbage collection can magnify minor design oversights into severe performance regressions.

5 Characteristics of Bugs

In this section, we study the characteristics of various aspects of real-world JIT compiler performance bugs to answer our research questions (Section 3.3). Specifically, we study input artifacts used to trigger performance bugs (Section 5.1), symptoms of performance bugs (Section 5.2), and root causes of performance bugs (Section 5.3). Table 2 shows the summary characteristics.

5.1 RQ1: Input Artifacts

JIT compilers, which are parts of virtual machines, take programs as input and optimize them. To study what kind of input artifacts are used in real-world performance bug reports, we analyzed the provided materials in bug reports that we selected in Section 3.2.

We identify the following common input artifacts provided in bug reports (the top part of Table 2):

- **Micro-benchmarks** (48.69%): Small, self-contained programs designed to expose specific performance issues in JIT compilers. These tests simplify diagnosis.
- **Benchmark suites** (24.61%): Standardized performance benchmarks such as DaCapo [11] and Renaissance [70] for JVM, as well as Octane [20] and JetStream [90] for JavaScript. These suites are often integrated into CI/CD pipelines to automatically detect performance regressions [33].
- **Applications** (13.61%): A performance issue is detected in real-world application programs and then further investigated to originate in the JIT compiler.
- **Other** (13.09%): Reports that do not specify the reproducing artifact but directly identify performance problems in the JIT compiler, often through code inspection or developer self-reporting.

While benchmark suites are widely used to evaluate the overall performance of JIT compilers, we find that they detect only about one-fourth of all performance bugs. In contrast, nearly half of the reported bugs are demonstrated by micro-benchmarks–small, focused programs that test specific compiler features. These micro-benchmarks are either explicitly designed to exercise particular language features or distilled from real-world examples. Because benchmark suites consist of a large fixed set of programs, they are not well-suited for revealing previously unknown performance issues and take a long time to run. By comparison, although micro-benchmarks target narrower and sometimes less common scenarios, their focused nature enables them to uncover unknown issues and subtle corner cases, while remaining lightweight, fast to execute, and easier to diagnose.

Moreover, during our dataset construction, we observed that the V8 issue tracker contains numerous false positives (i.e., reports later marked as “Won’t Fix” or “Obsolete”) generated by periodic benchmark runs [34]. This observation highlights the limitations of existing benchmark suites, which may fail to isolate compiler-specific regressions from performance noise.

Answer to RQ1. Micro-benchmarks play a crucial role in exposing JIT performance bugs, either by testing specific compiler optimizations or runtime behaviors, or by isolating the minimal program patterns from real-world examples. For the automated detection of performance bugs in JIT compilers, future work should focus on generating small, feature-specific programs that exercise distinct compiler optimizations or runtime behaviors, such as inlining, loop unrolling, speculative checks, etc.

5.2 RQ2: Symptoms

Performance bugs are typically identified through abnormal execution behaviors. We summarize the common symptoms observed in the collected bug reports (the middle part of Table 2):

Table 2. Summary of characteristics of studied performance bugs in JIT compilers.

	HotSpot	Graal	V8	SM	Total
Number of bugs studied	87	18	51	35	191
Artifacts Used to Trigger Performance Bugs					
Micro-benchmarks: A simple test case or micro-benchmark *Example: JDK-8300256, Graal-6564*	38	9	36	10	93
Benchmark Suites: Standardized benchmark suites *Example: V8-40782714, JDK-8333583*	24	2	9	12	47
Applications: Real-world application performance issues that are tracked down to JIT *Example: Mozilla-1131523, Graal-2548*	3	6	5	12	26
Other: All the bugs not included in the above categories *Example: V8-42208696, Mozilla-1331350*	22	1	1	1	25
Symptoms of Performance Bugs					
Performance Regression: Performance degradation with new versions *Example: JDK-8274983, V8-41480684*	28	3	12	17	60
Performance Difference: Similar settings deliver different performance *Example: V8-42208216, Graal-831*	10	6	18	13	47
Log: Abnormal output observed from logs of JIT compilers or other tools *Example: JDK-8280473, Mozilla-1526315*	19	2	12	2	35
Native Code: Suboptimal native code generated by JIT compilers *Example: JDK-8176513, Graal-2010*	10	4	1	1	16
Crash/Hang: Program crashes or hangs *Example: JDK-8256368, V8-40832269*	8	1	4	1	14
Other: All the bugs not included in the above categories *Example: JDK-8253923, V8-42207398*	12	2	4	1	19
Root Causes of Performance Bugs					
Optimization: Missed optimization opportunities or inappropriate optimizations *Example: JDK-8279888, Graal-701*	36	8	17	6	67
Speculation: Suboptimal speculative decisions *Example: V8-42209544, Mozilla-1644425*	14	2	22	17	55
Code Generation: Suboptimal native code sequence *Example: JDK-8172434, Mozilla-1503116*	21	6	6	5	38
Interaction with Runtime: Interaction between JIT and other components *Example: JDK-8155638, V8-42209849*	12	1	2	2	17
Long Compilation: JIT compilers spend a significant amount of time compiling code *Example: Mozilla-1318926, Graal-2548*	3	1	2	1	7
Other: All the bugs not included in the above categories *Example: JDK-8302736, V8-40782714*	1	0	2	4	7

- **Performance regression** (31.41%): A significant degradation in performance after JIT compiler version updates.
- **Performance difference** (24.61%): Noticeable performance gaps between equivalent executions–such as across platforms, vendors, or semantically identical program variants–often indicate compiler inefficiencies or missed optimizations for certain platforms or code patterns.

- **Log** (18.32%): Abnormal entries in compiler or runtime logs, or anomalies revealed by monitoring and profiling tools, signaling unexpected performance behaviors.
- **Native code** (8.38%): (Manually) inspecting the generated native code reveals inefficient instruction sequences or missing optimizations.
- **Crash/Hang** (7.33%): Severe performance issues may lead to crashes, hangs, or assertion failures triggered by incorrect compiler assumptions.
- **Other** (9.95%): Other irregular behaviors, such as performance fluctuations, that indicate unstable or inconsistent optimization decisions.

Unlike functional bugs, whose oracles are well-defined (i.e., program crashes or incorrect outputs), performance bugs lack a clear ground truth. The diversity observed in the list above underscores the difficulty of defining reliable oracles for performance validation.

The strong dependence on comparative signals suggests that *differential testing* [58] is particularly suitable for uncovering performance bugs in JIT compilers. Effective oracle design should try to combine quantitative metrics (e.g., execution time, compilation cost) with qualitative indicators (e.g., anomalous log entries or inefficient native code) to detect subtle performance degradations more systematically.

Answer to RQ2. Performance bugs in JIT compilers lack simple oracles and are often detected through various comparative evidence. Test oracles often employ a differential approach–comparing execution across compiler versions, configurations, or semantically equivalent programs–while integrating lightweight checks such as profiling counters, log anomalies, or code pattern analysis to pinpoint unexpected slowdowns and missed optimizations.

5.3 RQ3: Root Causes

To understand commonly buggy parts in JIT compilers and how to prevent or detect issues in these parts, we inspect the root causes in the studied bug reports. Some common root causes include (the bottom part of Table 2):

- **Optimization** (35.08%): Bugs caused by missed or incorrect optimizations, such as redundant computations, unoptimized loops, or overly aggressive transformations that miss certain edge cases. These bugs typically result from flawed heuristics or incomplete cost models.
- **Speculation** (28.80%): Bugs caused by invalid or misleading speculative assumptions derived from profiling data, or overly aggressive speculation that fails too often with high deoptimization and recompilation costs.
- **Code generation** (19.90%): Inefficiencies in instruction selection, register allocation, or scheduling that produce suboptimal native code. Such issues are similar to those in AOT compilers but are complicated by additional runtime guards or stubs inserted by JIT compilers.
- **Interaction with runtime** (8.90%): Performance issues caused by inefficient coordination between the JIT compiler and other runtime components, such as code cache management, dynamic compilation triggers, or garbage collectors.
- **Long compilation** (3.66%): Bugs due to excessive compilation time or resource usage, which delay code execution and reduce overall responsiveness.
- **Other** (3.66%): Issues related to internal design decisions, such as suboptimal data structures or algorithms, or cross-component interactions.

While optimizations, speculations, and code generation are closely intertwined and often influence each other, we distinguish between them using the following criteria. We classify speculation bugs as those related to the collection or use of profiling data, including setting profiling data, generating intermediate representation (IR) based on such data, and the deoptimization process. In contrast, we classify optimization bugs as those arising from IR transformations that are independent of profiling data, and code generation bugs as those related to the backend of a compiler—transforming IR to native code.

For example, in JDK-8280320, certain profiling data is incorrectly set, causing the compiler to miss certain loop optimizations even though the loop optimizations themselves are correctly implemented, so we classify it as a speculation bug. On the other hand, consider the case of JDK-8279888, there is an implementation flaw in the loop optimization itself, so we classify it as an optimization bug, since the problem will arise whenever the optimization is performed.

Traditionally, the optimization and code generation phases have been recognized as error-prone in AOT compilers. Our study reveals that JIT-specific components, particularly speculation and runtime interaction, are also significant sources of performance bugs, especially for dynamically-typed languages like JavaScript.

These components are difficult to validate using conventional compiler testing, because their correctness depends on dynamic runtime behavior, profiling feedback, and adaptive decisions made during execution. To improve robustness, JIT compiler testing should focus on systematically validating speculative and adaptive mechanisms under diverse runtime conditions. This includes stress-testing profiling-guided optimizations, evaluating deoptimization and recompilation triggers, and ensuring stable performance across varying workload characteristics.

Answer to RQ3. Beyond traditional optimization and code generation testing, JIT compilers require focused validation of dynamic behaviors. Testing frameworks should systematically emulate diverse profiling patterns, workload phases, and execution environments to capture the impact of dynamic feedback on compilation decisions. In particular, they should stress-test speculation heuristics, deoptimization triggers, and tiered compilation transitions to reveal unstable or overly aggressive adaptive behaviors that lead to performance degradation.

5.4 Fixes

We also examined the fixes proposed in each bug report to understand how performance issues are addressed. Unlike general software performance bugs, where fixes often follow recognizable patterns such as adjusting condition checks or adding early exits [42], we did not observe clear or recurring fix patterns in JIT compilers.

Two main factors contribute to this observation. (1) JIT compilers consist of tightly coupled components and phases, so fixes frequently involve coordinated edits across multiple modules. (2) The performance impact of source-level changes is difficult to predict, as even small modifications can alter the generated machine code in complex ways. Consequently, resolving these issues often requires deep, domain-specific understanding of compiler internals and optimization interactions.

These findings highlight the importance of modular design in JIT compilers to improve maintainability and isolate performance-critical logic. For instance, SpiderMonkey recently introduced CacheIR [23], a structured intermediate representation for inline caches, an essential speculative mechanism for optimizing dynamic property access [25], to make this component more maintainable and extensible.

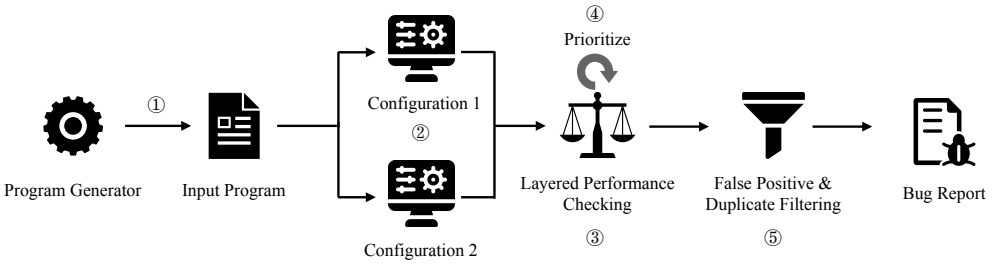


Fig. 7. Overview of layered differential performance testing.

6 JITTERY: Layered Differential Performance Testing

Based on our empirical study on input artifacts (Section 5.1), 48.69% of performance bugs can be demonstrated through micro-benchmarks, which are simple test cases that target specific features of JIT compilers. Thus, the goal of our approach is to generate a large number of small programs that can potentially reveal performance issues. We use various program generators, which have been previously shown effective at finding functional bugs in both AOT and JIT compilers [36, 51, 53, 93, 95]. Their potential for finding performance bugs has never been studied. Moreover, based on the common symptoms observed in real-world performance bugs (Section 5.2) such as performance difference (24.61%) and performance regression (31.41%), differential testing proved to be an effective approach for detecting performance bugs. To balance the trade-off between test efficiency and accuracy from a large number of generated programs, we designed and developed JITTERY, a tool that implements *layered differential performance testing* with test oracles for detecting performance bugs in JIT compilers.

We applied JITTERY to the HotSpot and Graal JIT compilers and found 12 previously unknown performance bugs with 11 confirmed or fixed by developers.

6.1 Overview

Figure 7 shows the overview of JITTERY. We next describe the key parts of JITTERY.

Program generators (①). The first step is to generate input micro-benchmarks that are likely to trigger certain JIT features. Due to the abundance of existing program generators for finding functional bugs, we leverage existing tools to generate programs. We use three generators in our experiments: Artemis [51], Java* Fuzzer [72], and LeJit [96].

Artemis is a mutation-based generator that mutates given programs in a semantics-preserving way to trigger different JIT compilation decisions. Java* Fuzzer is a random program generator, which is also used by Artemis to generate seed programs. LeJit is a template-based generator that generates programs by filling in holes in templates extracted from existing Java projects. To ensure JIT compilation is triggered, we invoke generated code a large number of times (by wrapping the code in a loop).

Differential configurations (②). The next step is to run the generated programs with a pair of configurations. There are many possible choices of such configurations as described in Section 5.2 (e.g., the original program and a mutated but semantically equivalent version of it). We explore two kinds of configurations in this paper: (1) Regression Pair (RP) and (2) Level Pair (LP). RP uses two different versions of the JIT compiler under test. LP leverages the tiered compilation feature of modern JIT compilers (Section 2.3). In this paper, we use levels L1 and L4 for HotSpot and Graal, which exercises C1 and C2 compilers. This way we enable different sets of optimizations to be applied, as well as different use of profiling and speculative optimizations.

Layered performance checking (③). We gradually filter out programs that are not likely to trigger performance bugs to balance test efficiency and accuracy, as doing rigorous benchmarking for every program is costly. Recall that we execute the entry method in a loop with a set number of iterations to trigger JIT compilation. We choose an ascending list of iteration counts (Ns). We refer to each iteration count as a *performance checking layer*. We run each program against these layers and when a program does not show an interesting performance difference, we skip the remaining layers. When we increase the iteration count for programs of interest, the benefit is two-fold: (1) the increased iteration count allows the warmup overhead to be amortized over more executions, reducing its impact on the measured performance; (2) the increased iteration count can augment potential performance issues and make them more observable.

Test inputs prioritization (④). Across different performance checking layers, we can use runtime information from previous layer(s) to prioritize which test inputs (namely programs) to run first in the next layer. This step can help find bugs faster with limited resources.

False positive and duplicate filtering (⑤). After all the performance testing is done, although the remaining programs are very likely to be true positives, there could potentially exist false positives due to inherent noise in performance measurement and duplicate bugs with the same root cause. To mitigate this problem, we use a few simple heuristics to automatically filter out false positives or duplicates. We find these heuristics to be effective and can greatly reduce manual effort. Finally, we do more thorough and rigorous performance measurement for remaining candidates before reporting them to compiler developers.

6.2 Algorithm

We formally describe the layered differential performance testing (Algorithm 1). The algorithm takes as input a set of programs Ps, a configuration pair $CP = (C_1, C_2)$, a list of iteration counts Ns (one per layer), thresholds THs (one per layer), and a list of top-k limits TOP_Ks (with $|Ns| = |THs| = n$ and $|TOP\_Ks| = n - 1$).

Initially (line 1), the algorithm determines the number of layers n, which equals the length of Ns. It then initializes an empty history structure H to store past performance measurements (line 2) and prepares a result list Rs, where each entry corresponds to a program and is initialized to `true` (line 3).

For each iteration, the algorithm reorders the program list using the prioritization function $\text{Prioritize}(Ps, H)$ (line 5), which leverages previous execution history to determine the order in which programs should be tested. Starting from the second iteration ($i > 1$), only the top-k programs after prioritization are selected, where $k = TOP\_Ks[i-1]$, and we skip the top-k selection if $TOP\_Ks[i-1] < 0$ (line 6). If fewer than k programs remain active (i.e., still marked as `true` in Rs), all remaining programs are executed (line 9).

Within each iteration, the algorithm evaluates every selected program P (line 11). If a program has already been filtered out in a previous iteration, it is skipped (line 13). Otherwise, the program is executed twice–once under configuration C_1 and once under configuration C_2 (lines 15–16)–to obtain two performance measurements m_1 and m_2. The function $\text{Check}(m_1, m_2, THs[i])$ (line 17) compares these measurements to determine whether the ratio between them exceeds the current threshold $THs[i]$. If this condition is not satisfied, the corresponding entry $Rs[P]$ is updated to `false` (line 18), indicating that the program is unlikely to reveal a performance bug and will not be tested in future iterations. Finally, the historical record $H[P]$ is updated with the new measurement pair (m_1, m_2) (line 20).

Algorithm 1 Layered Differential Performance Testing with Prioritization

Input: Programs Ps,
Configuration Pair $CP = (C_1, C_2)$,
Iterations Ns, Thresholds THs, Top-K limits TOP_Ks
Output: List of booleans Rs indicating if a potential performance bug exists for each program

$n \leftarrow |Ns|$
Initialize historical measurements $H \leftarrow [\,]$
Initialize result list $Rs[P] \leftarrow$ true for all $P \in Ps$
for $i \leftarrow 1$ **to** n **do**
  $Ps' \leftarrow \text{Prioritize}(Ps, H)$
  if $i > 1$ **and** $TOP\_Ks[i-1] > 0$ **then**
    $k \leftarrow TOP\_Ks[i-1]$
    $T \leftarrow \{P \in Ps' \mid Rs[P] = \text{true}\}$
    $Ps' \leftarrow Ps'[0 : \min(k, |T|)]$
  end if
  for all P in Ps' **do**
    if $Rs[P] = \text{false}$ **then**
      continue
    end if
    $m_1 \leftarrow \text{Execute}(P, C_1, Ns[i])$
    $m_2 \leftarrow \text{Execute}(P, C_2, Ns[i])$
    if $\neg\text{Check}(m_1, m_2, THs[i])$ **then**
      $Rs[P] \leftarrow \text{false}$
    end if
    $H[P] \leftarrow \text{UpdateHistory}(H[P], (m_1, m_2))$
  end for
end for
return Rs

There are some customizable functions in the algorithm: $\text{Execute}(P, C, N)$, $\text{Check}(m_1, m_2, TH)$, $\text{UpdateHistory}(H[P], (m_1, m_2))$, and $\text{Prioritize}(Ps, H)$. In this paper, we define them in a straightforward way, but future work should explore more sophisticated approaches:

- **Execute:** We run entry method in the program P under the given configuration C for N iterations and measure the execution time using hardware performance counters [91]. Other information (e.g., logs and native code as we show in Section 5.2), besides the execution time, could also be collected and used.
- **Check:** We assume there is a configuration (i.e., a newer compiler version or a higher compilation level) which runs faster than another configuration (an older compiler or a lower compilation level). We check if the ratio (m_2/m_1) of the two measurements exceeds the threshold TH. If so, we consider this program is likely to reveal a performance bug and return true.
- **UpdateHistory:** We do not consider all previous history in this paper, but only the latest measurement pair.
- **Prioritize:** We prioritize programs based on their previous performance measurements. We compute the ratio of the two measurements from the latest history and sort programs in descending order based on this ratio. The intuition is that a larger ratio indicates a more significant performance difference between the two configurations, making it more likely to reveal a performance bug. A more sophisticated prioritization method can also exploit features from the program under test itself and combine the static and dynamic information.

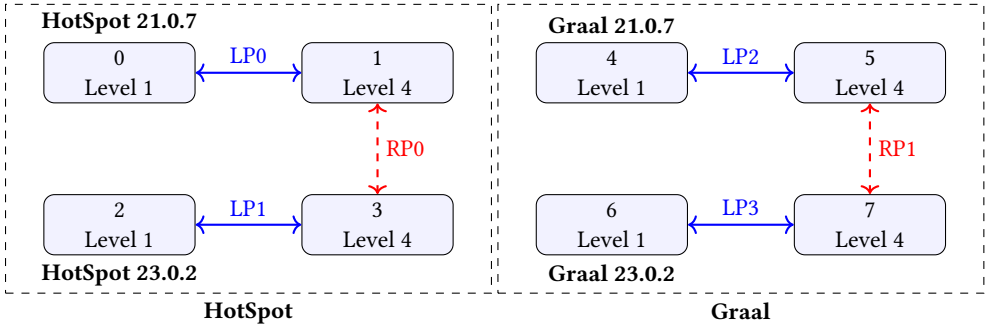


Fig. 8. Visualization of JDK configurations setup. Blue lines (LP0-LP3) connect optimization levels within each version; red dashed lines (RP0-RP1) connect regression pairs across versions.

6.3 Filtering False Positives and Duplicates

We now describe how we use simple heuristics to filter out false positives and duplicate bugs when the algorithm finishes.

For false positive filtering, consider the case where the performance difference for executing the entry method once is denoted as $m_d = |m_1 - m_2|$. As the iteration count increases, the accumulated difference $m_d \times Ns[i]$ is expected to grow proportionally. If a program's accumulated performance difference across multiple performance-checking layers does not exhibit such an increasing trend, it is likely a false positive caused by constant compilation overheads or transient environmental noise. This heuristic effectively eliminates cases where the observed performance gap is not a true reflection of runtime behavior differences for some short running programs.

We apply duplicate filtering to programs generated using LeJit. Different programs that are generated from the same template (i.e., they are structurally identical and only differ in certain values or expressions) often expose bugs that have identical root cause. To avoid redundant reporting, we record the template in a known-bug set and exclude subsequent findings with identical templates. In addition, we observe that exceptions frequently correlate with performance bugs. Therefore, we also group programs (for each program generator) by their thrown exception types (if any) and filter out new bugs that share the same exception. This combined strategy efficiently removes redundant bugs while retaining distinct performance issues for manual inspection.

6.4 Evaluation

In this section, we describe the steps we took to assess the benefits of using JITTERY, and we report our results and findings.

Experiment setup. We use two JDK versions 21.0.7 and 23.0.2, with each version corresponding to two JIT engines—HotSpot and Graal. This setup gives us four JIT compilers, eight configurations, four pairs of level configurations, and two pairs of regression pairs as depicted in Figure 8. We run using an Ubuntu 24.04.2 LTS machine with an Intel(R) Xeon(R) w5-3433 CPU and 128GB memory. We use the option `-XX:TieredStopAtLevel` to limit the reachable tiered execution levels, and the option `-Xbatch` to turn off background compilation. We report detailed results for LeJit as we did more extensive experiments with it.

Evaluation results. We show the evaluation results when using LeJit as the input generator for 8 Java projects, which were used to evaluate LeJit itself [96]. We choose (empirically) the following parameters for our algorithm (Algorithm 1): $n = 4$, $Ns = [100000, 500000, 1000000, 3000000]$, $THs = [1.2, 1.2, 1.3, 1.4]$ for LP and $THs = [1.1, 1.1, 1.2, 1.3]$ for RP. We disable the TOP_Ks limit in this experiment, and report results when using TOP_Ks later in this section.

Table 3. Evaluation results of Jittery using LeJit. Columns show the project name; numbers of extracted templates and generated programs; programs filtered at each layer; remaining programs after each filtering step; and total execution time (excluding the first layer).

Project	Input		Pair	Layered Filtering				Pass	False Positive Filtering	Duplicate Filtering	Unique	Time [min]
	#Template	#Gen		TH_0	TH_1	TH_2	TH_3					
codec	1157	6355	LP0	1824	3504	756	191	80	5	2	2	2167.55
			LP1	1804	3480	808	172	91	18	2	2	
			LP2	1417	2989	1209	467	273	14	3	2	
			LP3	1403	3035	1204	441	272	13	2	2	
			RP0	4926	1220	201	8	0	0	0	0	
			RP1	5202	971	175	7	0	0	0	0	
compress	792	6582	LP0	860	3102	1745	564	311	22	2	1	2276.14
			LP1	848	2950	1809	611	364	9	1	1	
			LP2	586	2000	1975	1186	835	13	1	1	
			LP3	560	2128	1870	1163	861	21	2	1	
			RP0	4578	1451	519	33	1	0	0	0	
			RP1	5249	1067	253	13	0	0	0	0	
lang	1472	12368	LP0	2319	6412	3141	408	88	1	1	1	3180.85
			LP1	2317	6495	3325	142	89	1	1	0	
			LP2	1619	4096	4213	1794	646	0	0	0	
			LP3	1634	3937	4325	1723	749	1	1	1	
			RP0	9357	2450	541	2	18	16	6	2	
			RP1	10141	1974	253	0	0	0	0	0	
math	656	5387	LP0	600	3150	1320	162	155	42	6	3	1901.12
			LP1	550	3169	1390	166	112	27	6	3	
			LP2	456	2447	1700	610	174	16	2	2	
			LP3	455	2398	1723	614	197	17	5	2	
			RP0	3994	1063	303	21	6	5	3	1	
			RP1	4346	834	197	9	1	0	0	0	
statistics	1062	8567	LP0	657	5928	1959	23	0	0	0	0	2296.09
			LP1	630	6100	1813	24	0	0	0	0	
			LP2	477	4291	3202	597	0	0	0	0	
			LP3	473	4278	3277	539	0	0	0	0	
			RP0	6191	1787	566	23	0	0	0	0	
			RP1	6781	1452	324	10	0	0	0	0	
text	1212	10592	LP0	3957	3870	2141	559	65	0	0	0	2330.62
			LP1	3842	3978	2004	711	57	0	0	0	
			LP2	3107	3260	1972	1031	1222	1	1	0	
			LP3	3132	3203	1936	1225	1096	0	0	0	
			RP0	8430	1662	427	20	53	28	16	1	
			RP1	8723	1621	248	0	0	0	0	0	
vectorz	785	6255	LP0	1462	3229	1208	269	87	20	2	2	2123.96
			LP1	1393	3344	1203	229	86	13	2	2	
			LP2	1108	2653	1300	772	422	11	2	2	
			LP3	1102	2652	1327	742	432	16	2	2	
			RP0	4786	1149	307	13	0	0	0	0	
			RP1	5123	945	174	13	0	0	0	0	
zxing	1182	10326	LP0	2316	4861	2278	570	301	86	1	1	3208.59
			LP1	2489	4727	2247	550	313	116	1	1	
			LP2	1520	3559	2743	1614	890	138	1	1	
			LP3	1512	3639	2673	1458	1044	132	1	1	
			RP0	7936	1866	508	16	0	0	0	0	
			RP1	8271	1642	327	86	0	0	0	0	

Table 4. Comparison of execution time with and without test prioritization across evaluation projects.

Project	Time [min] w/o prioritization	Time [min] w/ prioritization	Time reduction (%)
codec	2167.55	264.09	87.82
compress	2276.14	187.46	91.76
lang	3180.85	172.14	94.59
math	1901.12	212.78	88.81
statistics	2296.09	133.10	94.20
text	2330.62	171.57	92.64
vectorz	2123.96	226.77	89.32
zxing	3208.59	113.78	96.45
Total	19484.92	1481.69	92.40

Table 3[2] shows the evaluation results. The first column lists the project names, followed by two columns showing the number of templates extracted and the number of programs generated by LeJit (each template may correspond to multiple programs). The next column shows the used configuration pair. The next four columns present the number of programs filtered at each performance-checking layer. The following three columns show, respectively, the number of programs that remain after all layers ("Pass"), after false-positive filtering ("False Positive Filtering"), and after duplicate filtering ("Duplicate Filtering"). The last two columns report the number of unique true positive performance bugs ("Unique") and the total time spent on the entire process ("Time"), excluding the first performance layer in which all programs are executed. We exclude time for the first layer because Jittery is designed to *augment* existing program generators, which typically already execute all generated programs to check for functional bugs. The first layer can therefore be reused both to detect functional bugs and collect runtime information, upon which our layered differential performance testing proceeds to identify performance bugs.

We can observe from Table 3 that the first two layers (TH_0 and TH_1) filter out the majority of programs, significantly improving the efficiency of the testing process since the later layers involve more iterations and are more computationally expensive. The false positive and duplicate filtering stages further reduce the number of programs requiring manual inspection, leaving only a few cases to be examined manually. Although these two automated filtering steps can eliminate a large number of programs, they are not perfect, which explains the discrepancy between the counts in the second-to-last and third-to-last columns.

Effects of prioritization. Table 4 presents the execution time of Jittery with and without top-k prioritization (configured with $TOP\_Ks = [500, 100, 50]$). As shown in the table, enabling prioritization substantially reduces the overall testing time, achieving an average time reduction of 92.40%. This improvement stems from the fact that the prioritization strategy guides the testing process toward programs that are more likely to expose performance anomalies, thereby reducing the number of less promising candidates that require costly evaluation in later layers. Importantly, we observe that applying top-k prioritization does not lead to any loss of true positive bugs in our experiments. This finding indicates that our prioritization effectively accelerates the testing process while preserving its effectiveness in identifying genuine performance issues. These results demonstrate that prioritization can serve as a practical and scalable enhancement to Jittery.

[2]Because there are duplicate bugs across projects and JDK versions, and some bugs are not reproduced in this set of experiments, the counts in the second-to-last column do not sum up to the total number of bugs reported in this paper.

Table 5. Comparison of execution time (in minutes) between JITTERY using a single layer and JITTERY using multiple layers.

Project	Multiple Layers w/o prioritization	Single Layer w/o prioritization	Multiple Layers w/ prioritization	Single Layer w/ prioritization
codec	2167.55	5038.53 (+132.45%)	264.09	570.48 (+116.02%)
compress	2276.14	2762.75 (+21.38%)	187.46	339.52 (+81.12%)
lang	3180.85	3692.30 (+16.08%)	172.14	318.02 (+84.74%)
math	1901.12	3423.68 (+80.09%)	212.78	378.22 (+77.75%)
statistics	2296.09	4221.85 (+83.87%)	133.10	179.25 (+34.67%)
text	2330.62	4055.27 (+74.00%)	171.57	392.75 (+128.92%)
vectorz	2123.96	3568.98 (+68.03%)	226.77	438.52 (+93.38%)
zxing	3208.59	3624.03 (+12.95%)	113.78	216.17 (+89.99%)
Total	19484.92	30387.39 (+55.95%)	1481.69	2832.93 (+91.20%)

Effects of multiple layers. Using multiple layers enables gradual filtering of programs: with only a single layer, there is limited flexibility in selecting iteration counts (Ns) and thresholds (THs). For example, if the iteration count is set too low, many false positives may remain (as shown with TH_0 in Table 3), and if it is set too high, the approach becomes expensive, since many programs must be executed with a large number of iterations. Thus, we opt for a multi-layer approach.

Table 5 compares the execution time of single-layer and multi-layer settings. The settings for multi-layer remain the same as in Table 3. For the single-layer setting, we only keep the last layer besides the first layer, and thus we set the following parameters: $Ns = [100000, 3000000]$, $THs = [1.2, 1.4]$ for LP and $THs = [1.1, 1.3]$ for RP, and $TOP\_Ks = [500]$ (columns "w/ prioritization"). As shown in the table, the single-layer configuration incurs substantially higher execution time than the multi-layer approach, both with and without prioritization, since many more programs are executed in the final layer. Note that the single-layer setting can avoid missing true bugs as it skips the intermediate filtering layers. We did find a new regression bug between the two versions of Graal compilers used in our experiments (but the issue is not present in the latest version and is therefore not reported). Besides, we also observe that the execution-time gap varies across projects, e.g., in codec we observe +132.45% difference and in lang we observe +16.08% difference. This is because the runtime of some simple programs is relatively insensitive to high iteration counts, while the multi-layer approach introduces additional overhead from repeated instrumentation of different iteration counts and compilation of programs into bytecode.

6.5 Discovered Bugs

JITTERY discovered 12 previously unknown performance bugs in HotSpot and Graal JIT compilers, with 11 of them confirmed or fixed by developers.

Table 6 summarizes the bugs we found. For each bug, we show the corresponding generators and pairs used to find the bugs (if multiple choices are selected, it means the bug was found by several generators or pairs)[3]. We also include corresponding projects if the bug is found by LeJit (using the first source project of the bug-reporting template, with the number of projects shown in parentheses when multiple projects find the bug), as well as the root causes when such information has been confirmed by developers for our reports.

[3]The goal of our work is *not* to compare generators, but to showcase that JITTERY is able to detect performance bugs with programs obtained from various sources.

Table 6. Summarization of bugs found by JITTERY.

ID	Generator			Oracle		Project(s)	Root Cause	Status
	LeJit	Artemis	Java* Fuzzer	Level	Regression			
JDK-8345766	✓	-	-	✓	-	math (2)	Optimization	Fixed
JDK-8346989	✓	✓		✓	-	math (1)	Speculation	Fixed
JDK-8349401	✓	-	-	-	✓	math (2)	Codegen	Fixed
JDK-8349452	✓	-	-	✓	✓	zxing (1)	Codegen	Confirmed
JDK-8349364	✓	-	-	✓	-	vectorz (2)	Optimization	Confirmed
JDK-8350494	-	✓	-	✓	-	N/A	Optimization	Duplicate
JDK-8350720	-	✓	-	✓	-	N/A	Optimization	Confirmed
JDK-8353638	✓	✓	✓	✓	-	compress (6)	Speculation	Fixed
Graal-10565	✓	✓	✓	✓	-	lang (6)	Speculation	Fixed
Graal-10609	✓	-	✓	✓	-	math (2)	Unknown	Confirmed
Graal-10776	✓	-	-	-	✓	codec (1)	Speculation	Fixed

As shown in the table, JITTERY is capable of uncovering performance bugs spanning multiple compiler phases–not only those arising from optimization and code generation, but also speculation-related issues that are unique to JIT compilers. Besides, two of the bugs are severe performance regressions in standard libraries, and several are related to basic arithmetic calculations, which could be widely used by many programs. These findings demonstrate the generality and practical relevance of JITTERY in detecting diverse and impactful performance issues. We briefly describe four representative bugs detected by JITTERY, which spread across different phases of compilers.

Figure 9a shows a speculation bug where the HotSpot compiler continuously hits a deoptimization point and fails to self-correct the speculation. The `multiplyExact` method will throw an exception when the result overflows. When problematic arguments are continuously passed to the method, it will always hit the deoptimization point and move to the slow path. Instead, it should change its speculative assumption with updated profiling data.

Figure 9b shows a code generation bug where newly added optimizations may cause performance regression for other cases where the optimization may not be beneficial. A previous commit optimizes the `Arrays.fill()` stub for AVX512 targets, providing 2-5 times gains for large fill sizes. However, for small arrays, the call overhead to the optimized stub dominates the runtime, making the new path substantially slower.

Figure 9c shows a missed optimization bug in HotSpot C2 compiler. C2 currently lacks native lowering rules for floating-point remainder on most platforms. As a result, it always emits a runtime call instead of generating machine code, except on the obsolete x86-32 backend. This design prevents constant folding and other optimizations in the C2 pipeline–operations that C1 can perform successfully. The proposed fix converts `ModFNode` and `ModDNode` into macro nodes that are created during parsing, enabling standard C2 optimizations (e.g., constant folding and idealization) before being lowered to runtime calls during macro expansion.

Figure 9d shows a performance regression in `ByteArrayOutputStream.write()` under Graal, which is caused by conservative speculations. Calling `write()` in a loop repeatedly enters a small synchronized block on a newly allocated object. In older Graal versions, escape analysis correctly identified that the object did not escape the thread, replaced the monitor with a virtual lock, and completely eliminated synchronization. After the regression, however, Graal conservatively treated these locks as having side effects and emitted real `monitorenter` and `monitorexit` calls, forcing every iteration down the slow runtime locking path. The fix restores the original performance by reintroducing

```java
public static int square(int a) {
  return Math.multiplyExact(a, a);
}
public static void main(String[] a) {
  for (int i = 0; i < 5000000; i++) {
    try { square(i);}
    catch (Throwable e) {}
  }
}
```

(a) JDK-8346989

```java
public static void main(String[] a) {
  byte[][] bytes = new byte[90][1];

  for (int i = 0; i < 100000; ++i) {
    for (byte[] aByte : bytes) {
      Arrays.fill(aByte, i);
    }
  }
}
```

(b) JDK-8349452

```java
public static double f(final double x) {
  double w = 0.1;
  double p = 0;
  p = (3.10E307 % (w % Z));
  p = (7.61E307 / (x % x));
  return (x * p);
}
public static void main(String[] a) {
  for (int i = 0; i < 100000; i++) {
    f(1.0);
  }
}
```

(c) JDK-8345766

```java
public static void write() {
  final ByteArrayOutputStream buffer = \
        new ByteArrayOutputStream();
  for (int i = 0; i < 100; i++) {
    buffer.write(1);
  }
}
public static void main(String[] a) {
  for (int i = 0; i < 100000; i++) {
    write();
  }
}
```

(d) Graal-10776

Fig. 9. Examples of performance bugs found by JITTERY in HotSpot and Graal.

speculative virtual-lock support, allowing the compiler to assume locks are side-effect-free and safely virtualize them unless a deoptimization proves otherwise.

7 Limitations and Future Work

We discuss the limitations of JITTERY and outline directions for future work. Key challenges in detecting JIT compiler performance bugs lie in test input generation and test oracle design.

Test input generation. Detecting performance bugs effectively requires generating inputs that can trigger diverse JIT behaviors. Existing generators designed for finding functional bugs already explore many compiler paths, but how to tailor them toward performance-related behaviors remains an open question. For example, optimizations such as vectorization or inlining depend on specific program patterns (e.g., certain loop structures). Moreover, generating programs that systematically expose speculative optimizations and deoptimizations–behaviors unique to JIT compilers–presents an intriguing avenue for future research.

Test oracle. Designing test oracles for performance bugs is harder than for functional testing. The first challenge is choosing appropriate metrics. In addition to timing, logs and program states can also serve as performance indicators. Another challenge is measurement accuracy, which can be influenced by hardware, OS, and runtime factors. Accurately measuring the performance of JIT compilers is particularly challenging [7]. Although existing methods improve accuracy [31, 84, 98], they often reduce throughput. JITTERY therefore uses coarse-grained measurements for scalability, and future work could explore the use of fine-grained measurements.

8 Related Work

We discuss several related works in this section: (a) detecting performance bugs, (b) compiler testing, (c) finding missed optimizations in compilers, (d) JIT compiler benchmarking, and (e) formal verification of compilers.

Detecting performance bugs. Jin et al. [42] conducted one of the earliest empirical studies on performance bugs in real-world applications and proposed a rule-based approach for their detection. Subsequent empirical studies have also investigated performance bugs from various perspectives [4, 37, 74, 94], but all primarily focus on application-level software rather than compiler infrastructures.

In addition to empirical analyses, several automated techniques have been proposed to detect performance bugs. Fuzzing-based approaches such as JITProf [32], PerfFuzz [48], and others [38, 50, 83] generate diverse test inputs to expose pathological performance behaviors. Li et al. [52] studied *long compilation* bugs in markdown compilers, emphasizing inefficiencies during compilation rather than runtime execution. Other domain-specific tools target specialized systems, such as for database engines [5, 43] and for machine learning libraries [82]. Our work differs from these in that we focus on detecting performance bugs in JIT compilers themselves, which are not caused by the application source code but by the compiler's optimization and code generation phases.

Compiler testing. Extensive research has focused on testing functional correctness of both AOT and JIT compilers [18], spanning a wide range of languages such as C/C++ [28, 46, 47, 53, 56, 93], Java [19, 39, 51, 72, 95–97], JavaScript [36, 88, 89, 92], and Smalltalk [69].

Zhao et al. [97] synthesized realistic test programs by mining code snippets from real-world repositories to better expose functional bugs. Zang et al. [95, 96] proposed a template-based generation approach that automatically extracts program templates from real programs. Li et al. [51] developed a systematic method named compilation space exploration to trigger different compilation decisions in JIT compilers, improving coverage of dynamic behaviors.

Typing-related bugs in compilers have also been explored [13, 14, 26], alongside complementary efforts on test case prioritization [16, 17] and reduction [73, 78]. Several empirical studies have further analyzed compiler bug characteristics across ecosystems, including GCC/LLVM [77] and Rustc [54]. Marcozzi et al. [57] examined how compiler bugs found by fuzzers affect real-world applications, bridging the gap between research results and practical software reliability. Related compiler testing approaches have also been recently extended to other semantics-preserving code transformation tools such as JavaScript obfuscators and deobfuscators [40, 41].

Unlike this prior work, we are the first to focus on understanding and detecting performance bugs in JIT compilers.

Finding missed optimizations in compilers. Moseley et al. [60] investigated performance accountability in optimizing compilers, aiming to better attribute performance effects to specific optimization decisions. Barany [6] applied differential testing to identify missed optimizations by analyzing discrepancies in the generated binaries. Theodoridis et al. [80, 81] introduced techniques using dead-code markers and refined information to assess how effectively compilers eliminate redundant computations or exploit precise data-flow facts. Liu et al. [55] analyzed missed optimizations in WebAssembly compilers, while Gao et al. [30] proposed a source-level transformation framework to detect compiler-induced performance differences. The techniques used in prior works to derive semantically equivalent programs could potentially be extended to testing of JIT compilers. Jittery can naturally incorporate such similar program pairs by treating them as two distinct configurations.

Pečimúth et al. [68] studied Java JIT compilers by representing optimizations as trees and comparing them via tree edit distance to identify behavioral differences across compiler versions.

However, their work mainly focuses on optimization phases, leaving other JIT-specific components–such as speculation, tiered compilation, and runtime interaction–largely unexplored. Profiling-based approaches have also been used to support compiler-level performance debugging [10, 12] by analyzing internal compiler events to identify potential missed optimizations, though care is needed to avoid instrumentation interfering with compiler optimizations.

JIT compiler benchmarking. Benchmarking JIT compilers is more challenging than benchmarking AOT compilers since it involves warm-up stage and other runtime components [7, 84]. Several benchmarks have been proposed to represent real-world workloads [11, 70]. Some methods are proposed to more rigorously evaluate JIT compilers [31, 98]. Zheng et al. [98] studied deoptimizations in Graal JIT compiler and compared alternative deoptimization strategies. Southern et al. and Parravicini et al. [67, 76] studied deoptimization and speculation overhead in V8 JIT compiler. These benchmarks and methodologies help characterize JIT performance and can inform test oracle design. We focus on understanding performance testing of JIT compilers and designing the first approach for finding such bugs.

Formal verification of compilers. Formal verification is a promising approach to ensure the correctness of compilers. CompCert is the pioneering work in this area [49]. Recently, several other works have also focused on the formal verification of JIT compilers, including formal verification of speculative optimizations with dynamic deoptimization [8, 29], formal verification of native code generation [9]. Icarus [75] uses symbolic meta-execution to formally verify parts of real-world JIT compilers. All prior works focus on functional correctness of (JIT) compilers; verifying absence of performance bugs in generated code remains an open direction.

9 Conclusion

This paper provides the first systematic investigation into performance bugs in just-in-time (JIT) compilers. Through an empirical study of real-world issues across four widely used JIT compilers for Java and JavaScript, we identify recurring patterns in how these bugs arise and the compiler components most often affected. Guided by these observations, we develop Jittery, a lightweight tool based on layered differential performance testing that effectively uncovers JIT compiler performance anomalies while minimizing testing overhead and manual effort through prioritization and filtering. Jittery not only revealed multiple previously unknown performance bugs, several of which have been confirmed and fixed by developers, but also demonstrated the feasibility of automated approaches for diagnosing efficiency regressions in complex runtime systems. We hope our findings, Jittery, and dataset will inspire future research toward ensuring more robust, high-performance JIT compilers.

Acknowledgements

We thank Ivan Grigorik, Tong-Nong Lin, Aditya Thimmaiah, Zhiqiang Zang, Linghan Zhong, Jiyang Zhang, and anonymous reviewers for their feedback on this work. We are also grateful to compiler developers for investigating and addressing our bug reports. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036.

References

[1] Nader Al Awar, Zijian Yi, George Biros, and Milos Gligoric. 2025. Speeding up the local C++ development cycle with header substitution. In *International Symposium on Code Generation and Optimization*. 704–717. doi:10.1145/3696443.3708942

[2] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools.* pearson Education.

[3] Matthew Arnold, Stephen J Fink, David Grove, Michael Hind, and Peter F Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 93, 2 (2005), 449–466. doi:10.1109/JPROC.2004.840305

[4] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An empirical study of high performance computing (HPC) performance bugs. In *International Conference on Mining Software Repositories*. 194–206. doi:10.1109/MSR59073.2023.00037

[5] Jinsheng Ba and Manuel Rigger. 2024. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639076

[6] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *International Conference on Compiler Construction*. 82–92. doi:10.1145/3178372.3179521

[7] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27. doi:10.1145/3133876

[8] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally verified speculation and deoptimization in a JIT compiler. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–26. doi:10.1145/3434327

[9] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally verified native code generation in an effectful JIT: turning the CompCert backend into a formally verified JIT compiler. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 249–277. doi:10.1145/3571202

[10] Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2023. Optimization-aware compiler-level event profiling. *ACM Transactions on Programming Languages and Systems* 45, 2 (2023), 1–50. doi:10.1145/3591473

[11] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 169–190. doi:10.1145/1167515.1167488

[12] Humphrey Burchell, Octave Larose, and Stefan Marr. 2024. Towards realistic results for instrumentation-based profilers for JIT-compiled systems. In *International Conference on Managed Programming Languages and Runtimes*. 82–89. doi:10.1145/3679007.3685058

[13] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: A study of typing-related bugs in JVM compilers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. doi:10.1145/3485500

[14] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Programming Language Design and Implementation*. 183–198. doi:10.1145/3519939.3523427

[15] Craig Chambers and David Ungar. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Programming Language Design and Implementation*. 146–160. doi:10.1145/73141.74831

[16] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *International Conference on Software Engineering*. 700–711. doi:10.1109/ICSE.2017.70

[17] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *International Conference on Software Testing, Verification and Validation*. 266–277. doi:10.1109/ICST.2016.19

[18] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Comput. Surveys* 53, 1 (2020), 1–36. doi:10.1145/3363562

[19] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Programming Language Design and Implementation*. 85–99. doi:10.1145/2908080.2908095

[20] Chromium. 2025. Octane 2.0 Benchmark Suite. Accessed Oct 9, 2025 from https://chromium.github.io/octane/.

[21] Keith D Cooper and Linda Torczon. 2022. *Engineering a compiler*. Morgan Kaufmann.

[22] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. 1997. Compiling Java just in time. *IEEE Micro* 17, 3 (1997), 36–43. doi:10.1109/40.591653

[23] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J Nelson Amaral. 2023. CacheIR: The benefits of a structured representation for inline caches. In *International Conference on Managed Programming Languages and Runtimes*. 34–46. doi:10.1145/3617651.3622979

[24] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-stack replacement, distilled. In *Programming Language Design and Implementation*. 166–180. doi:10.1145/3296979.3192396

[25] L Peter Deutsch and Allan M Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Symposium on Principles of Programming Languages*. 297–302. doi:10.1145/800017.800542

[26] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *International Conference on Automated Software Engineering*. 482–493. doi:10.1109/ASE.2015.65
[27] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. 2017. {cHash}: Detection of redundant compilations via {AST} hashing. In *USENIX Annual Technical Conference*. 527–538.
[28] Yuyou Fan and John Regehr. 2024. High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM. In *International Symposium on Code Generation and Optimization*. 349–358. doi:10.1109/CGO57630.2024.10444854
[29] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of speculative optimizations with dynamic deoptimization. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28. doi:10.1145/3158137
[30] Fengjuan Gao, Hongyu Chen, Yuewei Zhou, and Ke Wang. 2024. Shoot yourself in the foot–efficient code causes inefficiency in compiler optimizations. In *International Conference on Automated Software Engineering*. 1846–1857. doi:10.1145/3691620.3695548
[31] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76. doi:10.1145/1297027.1297033
[32] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *International Symposium on the Foundations of Software Engineering*. 357–368. doi:10.1145/2786805.2786831
[33] Google Corporation and/or its affiliates. 2025. Addressing Performance Regressions. Accessed March 11, 2025 from https://chromium.googlesource.com/chromium/src/+/master/docs/speed/addressing_performance_regressions.md.
[34] Google Corporation and/or its affiliates. 2025. Chrome Issues for V8 Perf Regression. Accessed Oct 3, 2025 from https://issues.chromium.org/issues?q=%22%5BV8%20JavaScript%20Perf%5D%22.
[35] Google Corporation and/or its affiliates. 2025. Chromium Issue Tracker. Accessed March 10, 2025 from https://issues.chromium.org/issues.
[36] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities. In *The Symposium on Network and Distributed System Security*. doi:10.14722/ndss.2023.24290
[37] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *International Symposium on Empirical Software Engineering and Measurement*. 1–10. doi:10.1145/2961111.2962602
[38] Xue Han, Tingting Yu, and David Lo. 2018. Perflearner: Learning from bug reports to understand and generate performance test frames. In *International Conference on Automated Software Engineering*. 17–28. doi:10.1145/3238147.3238204
[39] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In *International Conference on Software Engineering*. 43–55. doi:10.1109/ICSE48619.2023.00016
[40] Shan Jiang, Pranoy Kovuri, David Tao, and Zhixun Tan. 2026. CASCADE: LLM-Powered JavaScript deobfuscator at Google. In *International Conference on Software Engineering, Software Engineering in Practice*. doi:10.1145/3786583.3786873
[41] Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2026. OBsmith: LLM-Powered JavaScript obfuscator testing. In *OOPSLA*. doi:10.1145/3798204
[42] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88. doi:10.1145/2345156.2254075
[43] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70. doi:10.14778/3357377.3357382
[44] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2000. Effective null pointer check elimination utilizing hardware trap. *ACM SIGPLAN Notices* 35, 11 (2000), 139–149. doi:10.1145/356989.357002
[45] Prasad A Kulkarni. 2011. JIT compilation policy for modern machines. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 773–788. doi:10.1145/2048066.2048126
[46] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. doi:10.1145/2666356.2594334
[47] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *Acm Sigplan Notices* 50, 10 (2015), 386–399. doi:10.1145/2858965.2814319
[48] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis*. 254–265. doi:10.1145/3213846.3213874
[49] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814
[50] Ao Li, Rohan Padhye, and Vyas Sekar. 2025. SPIDER: Fuzzing for Stateful Performance Issues in the ONOS Software-Defined Network Controller. In *International Conference on Software Testing, Verification and Validation*. 1–12. doi:10.1109/ICST62969.2025.10988999

[51] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT compilers via compilation space exploration. In *Symposium on Operating Systems Principles*. 66–79. doi:10.1145/3600006.3613140

[52] Penghui Li, Yinxi Liu, and Wei Meng. 2021. Understanding and detecting performance bugs in markdown compilers. In *International Conference on Automated Software Engineering*. 892–904. doi:10.1109/ASE51524.2021.9678611

[53] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting compiler testing by injecting real-world code. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 223–245. doi:10.1145/3656386

[54] Zixi Liu, Yang Feng, Yunbo Ni, Shaohua Li, Xizhe Yin, Qingkai Shi, Baowen Xu, and Zhendong Su. 2025. An empirical study of Rust-specific bugs in the rustc compiler. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3869–3896. doi:10.1145/3763800

[55] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *International Symposium on Software Testing and Analysis*. 436–448. doi:10.1145/3597926.3598068

[56] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1826–1847. doi:10.1145/3591295

[57] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. doi:10.1145/3360581

[58] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[59] Benedikt Meurer. 2024. An Introduction to Speculative Optimization in V8. Accessed March 2, 2025 from https://benediktmeurer.de/2017/12/13/an-introduction-to-speculative-optimization-in-v8/.

[60] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. 2009. Optiscope: Performance accountability for optimizing compilers. In *International Symposium on Code Generation and Optimization*. 254–264. doi:10.1109/CGO.2009.26

[61] Mozilla Corporation and/or its affiliates. 2025. Bugzilla. Accessed March 10, 2025 from https://bugzilla.mozilla.org/home.

[62] Mozilla Corporation and/or its affiliates. 2025. SpiderMonkey. Accessed Oct 6, 2025 from https://firefox-source-docs.mozilla.org/js/index.html.

[63] Oracle Corporation and/or its affiliates. 2025. Graal Issue Tracker. Accessed March 10, 2025 from https://github.com/oracle/graal/issues.

[64] Oracle Corporation and/or its affiliates. 2025. HotSpot Compilation Policy. Accessed March 11, 2025 from https://github.com/openjdk/jdk/blob/master/src/hotspot/share/compiler/compilationPolicy.hpp.

[65] Oracle Corporation and/or its affiliates. 2025. HotSpot Glossary of Terms. Accessed March 10, 2025 from https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html.

[66] Oracle Corporation and/or its affiliates. 2025. JDK Bug System. Accessed March 10, 2025 from https://bugs.openjdk.java.net/browse/JDK.

[67] Alberto Parravicini and Rene Mueller. 2021. The cost of speculation: Revisiting overheads in the V8 JavaScript engine. In *IEEE International Symposium on Workload Characterization*. 13–23. doi:10.1109/IISWC53511.2021.00013

[68] Andrej Pečimúth, David Leopoldseder, and Petr Tůma. 2023. Diagnosing Compiler Performance by Comparing Optimization Decisions. In *International Conference on Managed Programming Languages and Runtimes*. 47–61. doi:10.1145/3617651.3622994

[69] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-guided differential JIT compiler unit testing. In *Programming Language Design and Implementation*. 981–992. doi:10.1145/3519939.3523457

[70] 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 *Programming Language Design and Implementation*. 31–47. doi:10.1145/3314221.3314637

[71] Python Enhancement Proposals. 2024. PEP 744 - JIT Compilation. Accessed March 2, 2025 from https://peps.python.org/pep-0744/.

[72] Mohammad R. Haghighat. 2018. Java Fuzzer. Accessed March 11, 2025 from https://github.com/AzulSystems/JavaFuzzer.

[73] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*. 335–346. doi:10.1145/2345156.2254104

[74] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *International Conference on Software Engineering*. 61–72. doi:10.1145/2884781.2884829

[75] Naomi Smith, Abhishek Sharma, John Renner, David Thien, Fraser Brown, Hovav Shacham, Ranjit Jhala, and Deian Stefan. 2024. Icarus: Trustworthy just-in-time compilers with symbolic meta-execution. In *Symposium on Operating Systems Principles*. 473–487. doi:10.1145/3694715.3695949

[76] Gabriel Southern and Jose Renau. 2016. Overhead of deoptimization checks in the V8 JavaScript engine. In *IEEE International Symposium on Workload Characterization*. 1–10. doi:10.1109/IISWC.2016.7581268

[77] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *International Symposium on Software Testing and Analysis*. 294–305. doi:10.1145/2931037.2931074
[78] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *International Conference on Software Engineering*. 361–371. doi:10.1145/3180155.3180236
[79] The PyPy Team. 2025. PyPy. Accessed Oct 3, 2025 from https://pypy.org/.
[80] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709. doi:10.1145/3503222.3507764
[81] Theodoros Theodoridis and Zhendong Su. 2024. Refined input, degraded output: The counterintuitive world of compiler behavior. (2024), 671–691. doi:10.1145/3656404
[82] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *International Symposium on Software Testing and Analysis*. 189–199. doi:10.1145/3395363.3404540
[83] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2018. Synthesizing programs that expose performance bottlenecks. In *International Symposium on Code Generation and Optimization*. 314–326. doi:10.1145/3168830
[84] Luca Traini, Federico Di Menna, and Vittorio Cortellessa. 2024. AI-driven Java performance testing: Balancing result quality with testing time. In *International Conference on Automated Software Engineering*. 443–454. doi:10.1145/3691620.3695017
[85] V8. 2021. Sparkplug — a non-optimizing JavaScript compiler. Accessed March 10, 2025 from https://v8.dev/blog/sparkplug.
[86] V8. 2023. Maglev - V8's Fastest Optimizing JIT. Accessed March 10, 2025 from https://v8.dev/blog/maglev.
[87] V8. 2025. TurboFan. Accessed March 10, 2025 from https://v8.dev/docs/turbofan.
[88] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. 2025. DUMPLING: Fine-grained differential JavaScript engine fuzzing. (2025). doi:10.14722/ndss.2025.241411
[89] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced fuzzing for JavaScript engine JIT compiler. In *USENIX Security Symposium*. 1865–1882.
[90] Webkit. 2025. JetStream 2 Benchmark Suite. Accessed Oct 9, 2025 from https://browserbench.org/JetStream2.0/.
[91] Wikipedia. 2025. Hardware performance counter. Accessed Oct 9, 2025 from https://en.wikipedia.org/wiki/Hardware_performance_counter.
[92] Wai Kin Wong, Dongwei Xiao, Cheuk Tung Lai, Yiteng Peng, Daoyuan Wu, and Shuai Wang. 2025. Extraction and mutation at a high level: Template-based fuzzing for JavaScript engines. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 2898–2926. doi:10.1145/3763154
[93] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*. 283–294. doi:10.1145/1993316.1993532
[94] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *International Conference on Mining Software Repositories*. 199–208. doi:10.1109/MSR.2012.6224281
[95] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *International Conference on Automated Software Engineering*. 23:1–23:13. doi:10.1145/3551349.3556958
[96] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT Testing with Template Extraction. In *International Symposium on the Foundations of Software Engineering*. 1129–1151. doi:10.1145/3643777
[97] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *International Conference on Software Engineering*. 1133–1144. doi:10.1145/3510003.3510059
[98] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate profiling in the presence of dynamic compilation. *ACM SIGPLAN Notices* 50, 10 (2015), 433–450. doi:10.1145/2858965.2814281
[99] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2017. An empirical study on deoptimization in the Graal compiler. In *European Conference on Object-Oriented Programming*. 30:1–30:30. doi:10.4230/LIPIcs.ECOOP.2017.30

Received 2025-10-10; accepted 2026-02-17