Diagnosing Compiler Performance by Comparing Optimization Decisions

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

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

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

Abstract

Modern compilers apply a set of optimization passes aiming to speed up the generated code. The combined effect of individual optimizations is difficult to predict. Thus, changes to a compiler's code may hinder the performance of generated code as an unintended consequence.

Performance regressions in compiled code are often related to misapplied optimizations. The regressions are hard to investigate, considering the vast number of compilation units and applied optimizations. A compilation unit consists of a root method and inlined methods. Thus, a method may be part of several compilation units and may be optimized differently in each. Moreover, inlining decisions are not invariant across runs of the virtual machine (VM).

We propose to solve the problem of diagnosing performance regressions by capturing the compiler's optimization decisions. We do so by representing the applied optimization phases, optimization decisions, and inlining decisions in the form of trees. This paper introduces an approach utilizing tree edit distance (TED) to detect optimization differences in a semi-automated way. We present an approach to compare optimization decisions in differently inlined methods. We employ these techniques to pinpoint the causes of performance problems in various benchmarks of the Graal compiler.

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

*Partially funded by project SVV 260698/2023.

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10…$15.00
https://doi.org/10.1145/3617651.3622994

Keywords: Compiler Optimizations, Tree Matching, Just-In-Time Compilation, Virtual Machines

ACM Reference Format:
Andrej Pečimúth, David Leopoldseder, and Petr Tůma. 2023. Diagnosing Compiler Performance by Comparing Optimization Decisions. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617651.3622994

1 Introduction

Compilers rely on optimizations to generate efficient machine code. Optimizations are transformations on the level of intermediate representation (IR) [12]. Modern just-in-time (JIT) compilers use elaborate heuristics incorporating profiling feedback to determine which transformations are beneficial [23, 34]. The quality of optimization decisions is a crucial factor in determining the speed of a compiled program.

Compilers under active development, such as Graal [29], have several changes merged every day. Well-written commits contain atomic changes with clear intentions, e.g., to add a feature or to fix a bug. However, the actual effects of the changes may be unclear due to the interplay of individual optimizations and the system's overall complexity. Therefore, there may be additional unintended effects of each commit. As a result, the quality of generated code may be negatively impacted. In this paper, we refer to the quality of generated code as *compiler performance*. Thus, the unintended effects of changes pose the risk of compiler performance regressions. The Graal compiler undergoes automated regression testing [6, 14] to identify performance-affecting commits. The computation-time costs of regression testing [1] are significant and thus cannot be conducted for each commit.

When a performance regression is identified, it is necessary to determine its cause. However, it has been observed that the root cause is often unrelated [6] to the changed code. There may be too many code changes to inspect, and it is hard to predict their effects. Moreover, a regression may not manifest itself in each invocation of the virtual machine (VM).

Performance problems are often related to frequently executed code. To investigate a regression, a compiler engineer might profile the workload [32] to identify those native methods where most execution time is spent. A native method represents one compilation unit. A compilation unit in Graal consists of a root method and up to hundreds of inlined methods. The collected profiles sometimes uncover which compilation units take longer to execute after the regression. The cause of the regression is likely to be rooted in such a compilation unit.

Compiler performance regressions can often be traced down to individual optimizations. For example, the cause of a regression might be that a potentially beneficial optimization was not applied. Therefore, performance diagnosis consists of inspecting the optimizations in the affected compilation unit. However, the existing techniques to investigate the differences in optimization decisions are limited. Typical IR graphs contain thousands of nodes and undergo hundreds of transformations. The options include viewing and comparing the IR of individual compilation units [49] and miscellaneous logs produced by the optimizer.

Another source of complexity is that compilation units do not have a simple one-to-one mapping across VM invocations. The set of methods compiled as a compilation unit is not invariant. The methods are selected for compilation by non-deterministic execution counters, and inlining decisions [34] are not deterministic either. One method may be part of several compilation units and may be optimized differently in each. As a result, it is often infeasible to compare how a single method is optimized across VM invocations.

To diagnose these regressions, we propose capturing the compilation and execution of an application. During compilation, we track the optimization decisions, including the execution flow of the optimizer, represented as an *optimization tree*. Additionally, we build an *inlining tree*, which represents the structure of inlined code and associated inlining decisions. These two trees reflect the optimizations performed in a compilation unit.

Inlining often enables new optimization opportunities in a compilation unit. To represent these relationships, we propose linking optimization decisions to inlined code. To this end, we introduce the *optimization-context tree*, which shows optimization decisions in inlining contexts.

As a dynamic step, we profile the running application to estimate the execution time share of each compilation unit. We refer to the data from the compilation and execution steps collectively as an *experiment*.

We present profdiff, which is an approach to compare two experiments. We can leverage profdiff to compare two experiments compiled by different compiler versions, e.g., before and after a regression. Profdiff highlights the differences between optimization decisions in hot code. We identify these differences by semantically comparing inlining, optimization, and optimization-context trees using 1-degree [38] tree edit distance (TED).

Due to inlining [34], a single method may be part of several compilation units Conversely, several methods may be inlined into one compilation unit. Therefore, it is not sufficient to compare only pairs of compilation units. We introduce *compilation fragments* to compare optimization decisions in hot methods compiled in different contexts.

We implemented the described methods for the Graal compiler [29]. The implementation[1] is available in the open-source compiler repository. The concepts presented in this paper are applicable to any dynamic compiler that can correlate the IR with source code and is organized as a system of optimization passes.

We evaluated our tool with several engineers from the Graal compiler team and with industry-standard benchmark suites. We describe three workloads in which we pinpointed several suboptimal inlining decisions. The findings were validated by overriding the inlining decisions made by the compiler and resulted in a speed-up of about 8% to 30%.

In summary, we present a novel approach that automatically identifies optimization differences between two experiments in frequently executed code. This paper contributes the following:

- We propose capturing the dynamic execution flow of a compiler including the performed optimization decisions, the performed inlining decisions, and optimization decisions in inlining contexts in the form of trees.
- We propose comparing optimization decisions in frequently executed code by applying 1-degree TED to compute the differences between these trees.
- We propose a technique to compare optimization decisions performed in hot methods that are inlined in different contexts.
- We present an extensive evaluation of the tool with industry-standard benchmarks showing it can identify performance-affecting optimization decisions.

2 Compiler Performance Tracking

There are several changes merged to the Graal compiler daily. Compiler developers track [6] changes in metrics such as the wall clock time to execute a representative workload [5, 35]. Workloads from benchmark suites are repeated several times for selected compiler configurations and target platforms. The challenges of performance tracking include handling warm-up [3] of the JIT compiler and various sources of variance. For these reasons, performance tracking incurs a high cost [1] in terms of machine time. Statistical methods are employed [14] to detect performance changes in either direction.

[1] https://github.com/oracle/graal/blob/master/compiler/docs/Profdiff.md

Detecting a regression is the first part of the problem. Finding the cause of the regression is the task that follows. In this section, we analyze different kinds of possible regression causes. Performance tracking and regression cause analysis share a common set of challenges. These challenges are related to the architecture of GraalVM [30] and the non-determinism of the environment. We present a short overview of the architecture and explain the relevance to performance tracking and regression cause analysis.

2.1 Performance Regressions

Consider the commits merged to a compiler. In the simplest case, a commit may directly change the rules or heuristics that dictate when an optimization is applied. These changes may cause a regression such that an optimization is not applied when it should be or applied when it should not be. A compiler engineer might investigate this by identifying the optimization decisions that changed in a particular workload after the regression is detected. This is the primary use case for the automated detection of changed optimization decisions.

True reasons for performance changes are not always [6] directly related to the committed code. A change to a dynamic compiler may result in unexpected consequences. For example, changes to one optimization phase influence all successive phases. A manual search for indirect effects is difficult because it might be unclear what effects to look for. Therefore, automated optimization difference detection is a great fit for these scenarios.

Another class of challenges is related to the non-determinism of the VM. A performance problem may manifest itself only in some VM invocations leading to significant performance deviations. Thus, it is often necessary to sample several different compilation outcomes [15]. Replay compilation [20, 27, 28, 31, 37] helps to diagnose such workloads by reproducing prior compiler behavior. Performance distribution may shift as changes are merged into the compiler, which is considered an overall performance regression.

2.2 GraalVM

To understand the factors contributing to the non-determinism of the environment, we give a short overview of GraalVM's architecture. GraalVM is a configuration of the HotSpot Java Virtual Machine (JVM) [42] that replaces the server compiler [33] with the Graal compiler [29]. GraalVM utilizes tiered compilation [46] comprising an interpreter and two just-in-time compilers. The execution of a method starts in the interpreter. The interpreter collects profiling information [36], such as execution counters and type profiles for indirect calls. When the number of method invocations exceeds a threshold, the method is typically compiled with a modest optimization level by the client compiler [22]. The client compiler generates instrumented code that also collects profiles. After the number of method invocations passes another threshold, the method is compiled by the top-tier Graal compiler [29], striving for peak performance. The reverse step (i.e., deoptimization [18]) is also possible: execution may be transferred from compiled code to the interpreter. Figure 1 sums up the execution transfers between the interpreter and compiled code. Note that Figure 1 is a simplification, and the actual compilation policy [46] is more complex.

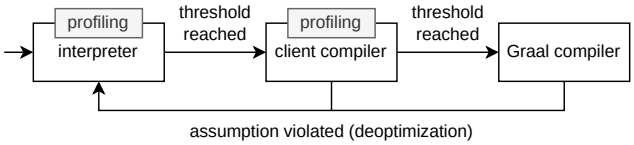

Figure 1. Execution transfers in GraalVM.

The collected profiles guide optimization decisions [23, 34, 47] during compilation. For example, the profiles are used to estimate the relative probability of an instruction [23]. This is useful to assess the benefit of an optimization given its cost (e.g., considering the code size increase). Type profiles allow the inliner [34] to devirtualize indirect call sites. Thus, the quality of generated code depends [47] on the quality of the profiles. However, the profiles are sampled from a limited time window. When the characteristics of the input data change, the profiles become inaccurate [47]. The window during which profiles are collected is also subject to factors such as the timing of compilation jobs or deoptimization [18].

Deoptimization [18] allows JIT compilers in GraalVM to optimize speculatively. The compiled code is aggressively optimized based on speculative assumptions. When an assumption is violated, the compiled code transfers control (deoptimizes) back to the interpreter. This is another source of non-determinism. The VM may resume profiling after deoptimization is triggered. Deoptimization occurs at a seemingly random time. The recompiled code may be optimized differently and exhibit different performance characteristics.

Tiered compilation leads to a warm-up phase [14] in some applications. However, it has been shown that some workloads do not reach a steady state at all [3]. The JIT compiler and the garbage collector, which run in parallel with application code, have been linked to inconsistency during a single VM run. The workloads are also subject to instability across multiple invocations of the VM. Therefore, it is necessary to invoke the VM several times and also repeat the workload during a single VM invocation [14].

3 Profdiff

This section introduces profdiff, an approach to capture and compare optimization decisions. We have extended the Graal compiler [29] with an option to collect and store optimization logs. The compiler collects an optimization tree (Section 3.1) and an inlining tree (Section 3.2) for each compilation unit.

We can associate optimization decisions with their inlining contexts using the optimization-context tree described in Section 3.3. The optimization-context tree is built from the collected optimization and inlining trees.

Our focus is the peak performance of a workload. Therefore, we capture logs only from the top-tier compiler, i.e., Graal. A top-tier compiler compiles only methods whose execution counters exceed predefined thresholds. In JIT terminology, these methods are considered hot. Our approach is based on the observation that a suboptimal optimization decision is likely to prolong the compilation unit's execution time. Additionally, the impact of suboptimal optimization is amplified in compilation units where a significant portion of time is spent. Thus, the transformation causing the performance degradation is likely to be found in a frequently executed compilation unit.

To identify the hottest compilation units, we profile the executing program using proftool [32], a profiler based on perf.[2] Proftool samples the execution time spent in the VM and in the generated code. Profdiff marks a configurable number of compilation units with the highest execution shares as *hot*.

Figure 2 shows how two runs (experiments) of the same workload are executed and compared using profdiff. We run the same workload twice on GraalVM. The workload may be executed with different VM or compiler versions. Profdiff compares the trees of hot compilation units to identify the different optimizations applied in two runs of the same workload. The comparison is restricted to hot compilation units to avoid reporting likely unimportant differences.

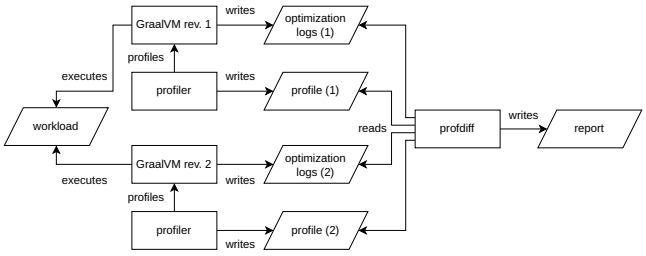

Figure 2. Executing and comparing two experiments.

Figure 3 illustrates how profdiff compares two experiments. Compilation units in each experiment are grouped by their root methods. The figure displays the sampled execution shares relative to the execution share of all compiled code. Compilation units marked as hot are highlighted in red.

The goal is to determine what optimization decisions differ in compilations of the same code. Several compilation units may be rooted in the same method due to speculative assumptions and consequent recompilations. Consider method

[2] https://perf.wiki.kernel.org/

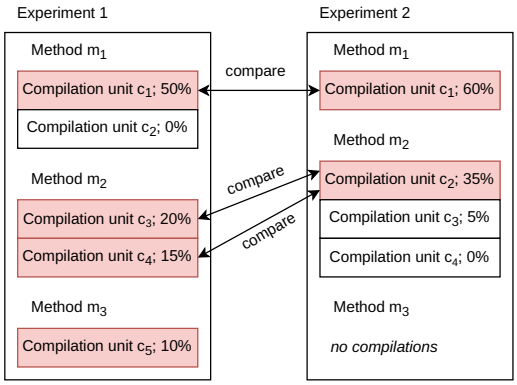

Figure 3. Comparing hot compilation units.

m_1 from Figure 3, compiled in both runs. Profdiff compares all hot compilation units of m_1 in experiment 1 with those of m_1 in experiment 2. In long-running workloads, later compilations that define the performance of the workload eclipse the initial compilation units.

Two compilation units are compared using their inlining, optimization, or optimization-context trees. We apply 1-degree TED [38] with some pre- and postprocessing to compare the trees. The result of the comparison is another tree that conveys which optimizations the compilations have in common and which optimizations are different. Section 3.4 explains this process in detail.

In the presence of inlining, comparing just pairs of compilation units is insufficient. For example, suppose that m_3 from Figure 3 is inlined in compilation c_1 in experiment 2. The problem is that we are not comparing the dedicated compilation of m_3 (compilation c_5 in experiment 1) with c_1 in experiment 2. The code of method m_3 is in both of these compilation units. Compilation units c_5 and c_1 might have optimized the code of method m_3 differently. Therefore, it is desirable to compare these optimization decisions. We solve this by creating compilation fragments, which we describe in Section 3.5.

3.1 Capturing Optimization Decisions

Compilers use an IR to represent the semantics of the compiled program. The Graal compiler uses a graph-based IR [12]. Throughout this text, we illustrate compiler transformations by listing Java code, although Graal performs these transformations on IR graphs.

This section talks about transformations, optimizations, and optimization phases. A transformation is an operation that changes the IR. An optimization is a kind of transformation aiming to speed up generated code. An optimization phase is a procedure that applies optimizations or invokes other optimization phases.

We capture the decisions to perform optimizations (i.e., optimization decisions) because changes in these decisions

are often linked to changes in code quality. Changes in optimization decisions are frequent between compiler versions or even successive VM invocations. Moreover, some of these decisions are based on estimates or inaccurate data. Thus, suboptimal decisions are expected and may lead to performance regressions.

Complex optimization phases, e.g., duplication [23] or inlining [34], decide whether a transformation is worth applying by estimating its cost and benefit. These estimates are prone to instability as described in Section 2. The cost is linked to the increased code size. The benefit comprises direct effects (e.g., removing call overhead) and enabled optimization opportunities (e.g., conditional elimination). The benefit also depends on the execution frequency of the affected code, and the execution frequency is in turn estimated from the collected profiles.

The optimization phases in Graal follow a *phase plan*, which comprises a list of optimization passes that run in a preset order. Selected phases run iteratively and apply other phases. A phase may be applied more than once in a phase plan. The performance of the generated program is sensitive to which phases are applied in what order. Thus, compiler developers may tune the phase plan between compiler revisions. This motivates capturing the dynamic phase plan for each compilation. The dynamic phase plan reflects the execution flow of the optimizer. Additionally, we associate optimization decisions with the phases that performed them.

Optimization phases are composable, i.e., an optimization phase may invoke another optimization phase. Consider the example in Listing 1. The second if-statement (lines 7–9) is duplicated to the branches of the preceding if-else statement (lines 2–6). After that, the duplication phase applies a dedicated conditional elimination phase [40], which identifies one of the duplicated conditionals to be `false`. Finally, a dedicated phase performing local optimizations simplifies the control-flow graph. The optimized code is illustrated in Listing 2.

Listing 1. Unoptimized code.

```
1  int foo(int i) {
2    if (i > 0) {
3      i += 1;
4    } else {
5      i = 0;
6    }
7    if (i > 7) {
8      i += 1;
9    }
10   return i;
11 }
```

Listing 2. Optimized code.

```
int foo(int i) {
  if (i > 0) {
    i += 1;
    if (i > 7) {
      i += 1;
    }
  } else {
    i = 0;
  }
  return i;
}
```

Figure 4 shows the relationship between the duplication phase and the subsequent optimization phases. The duplication phase modifies some input graph IR_1, and the result is graph IR_4. Each arrow in Figure 4 is a graph transformation applied by a particular optimization phase. The first duplication (leftmost arrow) is directly applied by the duplication phase. Then, the duplication phase invokes the conditional elimination phase, which applies a conditional elimination (middle arrow). In the end, the duplication phase invokes the canonicalizer phase, which performs a local IR simplification (rightmost arrow).

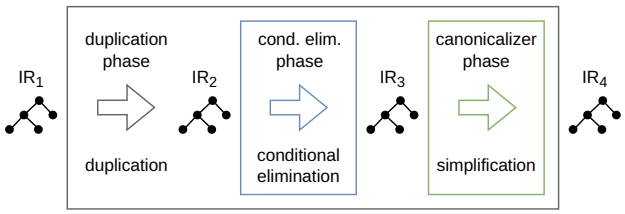

Figure 4. Composition of optimization phases.

An applied transformation often enables new optimization opportunities. An example of this is duplication enabling conditional elimination, as we have shown in Listing 1 and Listing 2. Local IR simplifications may enable additional IR simplifications. Such self-enabling phases may be applied iteratively until there are no more optimization opportunities. To improve interpretability, we capture the order of the applied optimizations and also associate them with the dynamic phase plan. The position of an optimization (phase) in the phase plan may explain its purpose, e.g., to clean up after duplication.

Optimization Tree. In order to preserve optimization decisions, their order, and the phases that applied them, we represent them as an optimization tree. The optimization tree is an ordered tree, where each node corresponds to either a phase or an optimization decision. The children of a phase are the phases and optimizations that the phase applied.

We illustrate the optimization tree using a running example, shown in Listing 3. The code reads lines from the

Listing 3. Running example: prints whether all JSON literals are equal.

```
1  class Example {
2    public static void main(String[] args) {
3      int limit = 0;
4      if (args.length > 0) {
5        limit = Integer.parseInt(args[0]);
6      }
7      System.out.println(literalsEqual(limit));
8    }
9    static boolean literalsEqual(int limit) {
10     Scanner scanner = new Scanner(System.in);
11     Object first = null;
12     for (int i = 0; i < limit; i++) {
13       String line = scanner.nextLine();
14       Object literal = JSONParser.parse(line);
15       if (i == 0) {
16         first = literal;
17       } else if (!literal.equals(first)) {
18         return false;
19       }
20     }
21     return true;
22   }
23 }
```

standard input and interprets each line as a JSON literal. The program prints `true` if all JSON literals are equal and `false` otherwise. The first program argument determines the number of lines read. If no arguments are provided, the program does not read any line. The example is slightly contrived to demonstrate various optimization opportunities.

For example, it might be worth peeling [40] the loop at line 12. Loop peeling involves pulling the first loop iteration in front of the loop. Listing 4 illustrates the result of loop peeling (omitting the rest of the method). The line numbers in Listing 4 represent the lines from which the code originates. Loop peeling opens an additional optimization opportunity: the condition `i == 0` always holds in the peeled iteration.

Listing 4. After peeling the loop at line 12 from the running example (Listing 3).

```
12  int i = 0;
12  if (i < limit) {
13    String line = scanner.nextLine();
14    Object literal = JSONParser.parse(line);
15    if (i == 0) {
16      first = literal;
17    } else if (!literal.equals(first)) {
18      return false;
19    }
12    i++;
12    for (; i < limit; i++) {
13      line = scanner.nextLine();
14      literal = JSONParser.parse(line);
15      if (i == 0) {
16        first = literal;
17      } else if (!literal.equals(first)) {
18        return false;
19      }
20    }
20  }
```

We store a descriptive name of the transformation for each optimization decision. For some decisions, we store additional key-value properties. After performing the loop peeling shown in Listing 4, the compiler records the following information.

```
LoopPeeling line 12 with {peelings: 1}
```

The line above illustrates the content of the logs, which are stored in a structured format. The key-value property `peelings: 1` informs that this is the first peeling of the loop. These properties further disambiguate the kind of performed transformation.

Optimization decisions are associated with positions in the source code. In Listing 4, the line numbers serve as the positions. The positions and properties not only improve interpretability but also establish whether profdiff considers two optimization decisions equivalent.

We obtain the position of an optimization by using the position of a node affected by the optimization. Compilers usually have mechanisms to track these positions. For simplicity, the positions in the example are line numbers. In our implementation, we use the offset of the bytecode instruction, which is more fine-grained than line numbers. For optimizations that affect more than one node, such as loop transformations, we record the position of one of the affected nodes (e.g., the node modeling the beginning of a loop).

Listing 5 shows a snippet of an optimization tree produced by compiling `literalsEqual` from the running example. The root of the tree is the root phase. The root phase applied the loop-peeling phase, and the loop-peeling phase peeled the loop at line 12. After that, the loop-peeling phase invoked the canonicalizer phase, which performs local IR simplifications. The canonicalizer phase replaced the increment of the induction variable `i + 1` with the constant 1 in the peeled iteration. Similarly, the condition `i == 0` in line 15 is trivially satisfied in the first iteration. Thus, the equals (==) node was replaced with the constant `true`.

Listing 5. Optimization tree of `literalsEqual` from the running example (Listing 3).

```
RootPhase
  LoopPeelingPhase
    LoopPeeling line 12 with {peelings: 1}
    IncrementalCanonicalizerPhase
      CanonicalReplacement line 12
        with {replacedNodeClass: +, canonicalNodeClass: Constant}
      CanonicalReplacement line 15
        with {replacedNodeClass: ==,
              canonicalNodeClass: LogicConstant}
```

When only one method is compiled, the line number (or an instruction offset) might sufficiently represent the position. In the presence of method inlining, it is necessary to capture the inline call stack relative to the root method. As an example, assume we compile method `main` from Listing 3 and inline the call to `literalsEqual`. After peeling the inlined loop, the compiler logs the following information.

```
LoopPeeling line {Example.literalsEqual(int): 12,
  Example.main(String[]): 7} with {peelings: 1}
```

The interpretation of the above example is that the loop originates in method `literalsEqual` at line 12, which is inlined in method `main` at line 7. If `main` additionally invoked `literalsEqual` at a different line, different positions would distinguish the optimizations in the inlined code.

Graal may parse a method, optimize it, and then inline it in a different method. To ensure that the optimization decisions performed in the inlined callee are preserved, we build an optimization tree for each IR graph. Whenever a callee is inlined, we copy the callee's optimization tree to the optimization tree of the caller. The callee's tree is attached as a subtree of the optimization phase that performed the inlining. It is necessary to update the positions of optimization decisions in the copied tree so that they reflect the new context.

3.2 Capturing Inlining Decisions

In modern compilers, inlining is essential for the performance of many programs. Inlining not only eliminates call

overhead but also introduces new optimization opportunities. Improved inlining policies can significantly boost performance [34]. Capturing inlining information is a necessity in the context of comparing optimization decisions.

Listing 6 shows method `main` from the running example (Listing 3) after duplication. The duplication creates an opportunity to inline the call to `literalsEqual` in the else-block. The constant argument allows the compiler to remove the loop in the inlined method. The loop removal is realized as a simple local IR optimization after loop peeling (Listing 4) because the peeled condition is `false` when `limit` equals 0.

Listing 6. After duplication in method `main` from the running example (Listing 3).

```
2   public static void main(String[] args) {
4     if (args.length > 0) {
5       int limit = Integer.parseInt(args[0]);
7       System.out.println(literalsEqual(limit));
6     } else {
7       System.out.println(literalsEqual(0));
6     }
8   }
```

The inlining tree is a tree of call sites. Each node is associated with a target method. The root node corresponds to the compiled root method. The children of each node correspond to method calls. For each node except the root, we store the position of the instruction which invokes the method in the parent's method body. We assign a call-site category to each. The category reflects the state of the call site at the end of the compilation. For example, Listing 7 shows an inlining tree created by compiling method `main` from the running example (Listing 3). Call-site categories are displayed in parentheses (explained in detail later). The tree shows that `literalsEqual` is inlined.

Listing 7. Inlining tree of method `main` from the running example (Listing 3).

```
(root) Example.main(String[])
  (direct) Integer.parseInt(String) line 5
  (inlined) Example.literalsEqual(int) line 7
    (direct) Scanner.init(InputStream) line 10
    (direct) Scanner.nextLine() line 13
    (direct) JSONParser.parse(String) line 14
    (indirect) Object.equals(Object) line 17
  (direct) PrintStream.println(boolean) line 7
```

We store a list of inlining decisions for each node (omitted in Listing 7). An inlining decision from the inliner is either positive (the callee was inlined) or negative, and the decision is linked to a message explaining the reasoning.

At the beginning of a compilation, we start with a tree consisting of only the root method. The root is assigned the special category `root`. Then, we create a node for each call-site in the compiled method body. Non-inlined callsites are leaf nodes and categorized as `indirect` if the call involves dynamic dispatch or `direct` otherwise. Whenever a callsite is inlined, we create the corresponding nodes for the invocations in the inlined callee's body. Inlined calls are marked as `inlined`. The collected inlining tree captures the final state at the end of the compilation.

Each non-inlined callsite is linked to a method-invocation node in the IR. The compiler might delete such a node from the IR, e.g., when the node is in an unreachable branch. The callsites linked to deleted nodes are classified as `deleted` in the inlining tree.

Indirect Calls. The call target of a callsite may be indirect, i.e., the target of the call is designated at runtime. For this reason, the call cannot be directly inlined. Thus, for each callsite, we record whether it is direct or indirect.

Consider the program from the running example (Listing 3). The JSON parser returns an object representing a literal, e.g., an `Integer`, `String`, `List`, or a `Map`. These types override the `equals` method. Therefore, the call to `equals` is marked as `indirect` in the inlining tree (Listing 7).

GraalVM [30] records the frequencies of receiver types for indirect call sites. The receiver type determines the concrete method to call. Profile accuracy is a possible source of suboptimal inlining decisions. Therefore, we record receiver-type profiles for indirect callsites, and profdiff displays them in the inlining tree.

The compiler may inline an indirect call site through devirtualization. If there is only one recorded receiver type for an invocation, the compiler can relink the call to the recorded receiver. In JIT, this may involve speculation [13]. Relinking the call makes it effectively direct and inlinable. Note that the inlining tree captures the state at the end of the compilation.

Suppose the input to the JSON parser from the running example (Listing 3) comprised only integers. The compiler could speculatively insert a type check and inline the call to `Integer.equals`. Listing 8 shows the inlining tree after the transformation.

Listing 8. Inlining tree of method `literalsEqual` from the running example (Listing 3) after type-guarded inlining.

```
(root) Example.literalsEqual(int)
  (direct) Scanner.init(InputStream) line 10
  (direct) Scanner.nextLine() line 13
  (direct) JSONParser.parse(String) line 14
  (inlined) Integer.equals(Object) line 17
```

A polymorphic call is devirtualized by replacing it with a type switch (an if-cascade with type checks) for the receiver type [19, 34]. Each branch of the switch leads to a direct inlinable call and possibly to a virtual call or deoptimization as a fallback. When these direct invocations are created, we attach them to the inlining tree as the children of the indirect callsite. Listing 9 shows the result of inlining some receivers from the JSON parser example. By attaching the new nodes as children, we convey that they were created by devirtualization.

Listing 9. Inlining-tree snippet for a devirtualized call site.

```
(devirtualized) Object.equals(Object) line 17
  (inlined) Integer.equals(Object) line 17
  (inlined) String.equals(Object) line 17
  (inlined) List.equals(Object) line 17
```

3.3 Optimizations in Context

Every transformation performed by the compiler affects a set of IR nodes. As explained in Section 3.1, we assign *positions* to the captured optimization decisions. Consequently, we can link optimization decisions to inlined code. The optimization-context tree is the inlining tree extended with optimization decisions. In this tree, each optimization decision is a child node of the method whose code the optimization transformed — this link is given by the assigned position. The optimization tree shows optimization decisions in the dynamic context of the compiler, whereas the optimization-context tree shows them in the context of the application code. Thus, both trees show the same set of optimization decisions, except their structure conveys complementary information.

The optimization-context tree shows what optimizations were applied to each compiled method. Linking optimization decisions to methods is also useful when two compilation units (or fragments) are compared. The difference between the two optimization-context trees shows what optimization decisions were applied to methods compiled in both compilation units. Moreover, if one of the compilation units inlines a method that the other does not, the representation discerns what optimizations were performed in such differently inlined code.

Profdiff builds the optimization-context tree by extending an inlining tree with optimization decisions from an optimization tree. As an illustration, the optimization-context tree in Figure 5c is built from the trees in Figure 5a and Figure 5b. The process starts by copying the inlining tree. Then, all optimization decisions from the optimization tree are attached as leaves. The place where an optimization decision is attached is determined by the position of the optimization decision.

3.4 Comparing Optimization and Inlining Decisions

Consider two compilation units with optimization and inlining decisions. The decisions are captured as optimization, inlining, or optimization-context trees. In this section, we examine potential differences between two compilations regarding optimization and inlining decisions. In order to identify these differences, we apply a tree-matching algorithm to compare the presented trees. We introduce the delta tree [7], which is a tree representation of optimization and inlining decisions that are either different or identical.

3.4.1 Comparing Optimization Trees. Recall that the optimization tree captures the applied optimization decisions, phases, and their relative order. If an optimization

```
(root) Example.main(String[])
  (direct) Integer.parseInt(String) line 5
  (inlined) Example.literalsEqual(int) line 7
    (direct) Scanner.init(InputStream) line 10
    (direct) Scanner.nextLine() line 13
    (direct) Scanner.nextLine() line 13
    (direct) JSONParser.parse(String) line 14
    (direct) JSONParser.parse(String) line 14
    (deleted) Object.equals(Object) line 17
    (indirect) Object.equals(Object) line 17
  (direct) PrintStream.println(boolean) line 7
```

(a) Inlining tree.

```
RootPhase
  LoopPeelingPhase
    LoopPeeling line {Example.literalsEqual(int): 12,
        Example.main(String[]): 7} with {peelings: 1}
    IncrementalCanonicalizerPhase
      CanonicalReplacement line {Example.literalsEqual(
          int): 12, Example.main(String[]): 7}
      CanonicalReplacement line {Example.literalsEqual(
          int): 15, Example.main(String[]): 7}
```

(b) Optimization tree.

```
(root) Example.main(String[])
  (direct) Integer.parseInt(String) line 5
  (inlined) Example.literalsEqual(int) line 7
    LoopPeeling line 12 with {peelings: 1}
    CanonicalReplacement line 12
    CanonicalReplacement line 15
    (direct) Scanner.init(InputStream) line 10
    (direct) Scanner.nextLine() line 13
    (direct) Scanner.nextLine() line 13
    (direct) JSONParser.parse(String) line 14
    (direct) JSONParser.parse(String) line 14
    (deleted) Object.equals(Object) line 17
    (indirect) Object.equals(Object) line 17
  (direct) PrintStream.println(boolean) line 7
```

(c) Optimization-context tree.

Figure 5. Inlining, optimization, and optimization-context trees of method `main` from the running example (Listing 3).

decision does not have a matching decision in the other compilation unit, the difference should be reported. Moreover, any change in whether or when a phase is applied should be reported as well. Optimization phases may be applied depending on dynamic conditions. The order of optimization phases matters because each transformation potentially influences subsequent transformations.

To illustrate this, Figure 6a shows a possible optimization tree of method `literalsEqual` from the running example (Listing 3). Figure 6b captures a regression scenario (i.e., the second experiment): the order of the canonicalizer and loop-peeling phases is reversed. Figure 6c represents the differences as a delta tree [8]. The tree contains nodes from the optimization trees, and each node is prefixed with a symbol. The interpretation of "`.`" is that the node was unchanged, "`-`" means that the node was deleted, and "`+`" means that the node was inserted.

3.4.2 Comparing Inlining Trees. Two inlining trees built by compiling the same method may have non-identical shapes. For example, code duplication [23] multiplies the number of

```
RootPhase
  LoopPeelingPhase
    LoopPeeling line 12
    IncrementalCanonicalizerPhase
      CanonicalReplacement line 12
      CanonicalReplacement line 15
```

(a) Baseline optimization tree.

```
RootPhase
  LoopPeelingPhase
    IncrementalCanonicalizerPhase
    LoopPeeling line 12
```

(b) Regressed optimization tree.

```
. RootPhase
  . LoopPeelingPhase
    + IncrementalCanonicalizerPhase
    . LoopPeeling line 12
    - IncrementalCanonicalizerPhase
      - CanonicalReplacement line 12
      - CanonicalReplacement line 15
```

(c) Differences in the form of a delta tree.

Figure 6. Optimization tree of method `literalsEqual` from the running example (Listing 3) compared with a regressed optimization tree and their delta tree.

call sites. Moreover, when a method is inlined, inlining-tree nodes are created for the callees of the inlinee.

Another kind of difference is different transformations applied to the same call site. We say that two inlining-tree nodes represent *the same* call site if their paths from the root match. The path from the root to a node consists of the target methods and call-site positions on the path.

As an example, the compiler might inline the same call in only one of the compared compilations. The applied transformation is reflected in the call-site category we described earlier. A possible explanation for such a difference might come from the reasoning of the inliner or the collected profiles. Profdiff displays this information when a difference is detected.

To illustrate this, Figure 7a is an inlining tree obtained by parsing method `main` from the running example (Listing 3). Figure 7b lists the inlining tree of the same method after inlining the call to `literalsEqual`. This inlining tree also contains nodes for the callees of `literalsEqual`. Figure 7c shows the differences between the trees in the form of a delta tree [8]. The delta tree highlights that `literalsEqual` is a direct call in the first tree, but the call is inlined in the second tree. The delta tree also shows the call sites that are present only in the second inlining tree.

3.4.3 Comparing Optimization-Context Trees.

We can compare optimization-context trees to identify different inlining and optimization decisions. Recall that the tree places optimization decisions in their inlining contexts. Thus, the comparison highlights optimization differences in each inlined method separately.

```
(root) Example.main(String[])
  (direct) Integer.parseInt(String) line 5
  (direct) Example.literalsEqual(int) line 7
  (direct) PrintStream.println(boolean) line 7
```

(a) Initial inlining tree of method `main`.

```
(root) Example.main(String[])
  (direct) Integer.parseInt(String) line 5
  (inlined) Example.literalsEqual(int) line 7
    (direct) Scanner.init(InputStream) line 10
    (direct) Scanner.nextLine() line 13
    (direct) JSONParser.parse(String) line 14
    (indirect) Object.equals(Object) line 17
  (direct) PrintStream.println(boolean) line 7
```

(b) Inlining tree with `literalsEqual` inlined.

```
. (root) Example.main(String[])
  . (direct) Integer.parseInt(String) line 5
  * (direct -> inlined) Example.literalsEqual(int) line 7
    + (direct) Scanner.init(InputStream) line 10
    + (direct) Scanner.nextLine() line 13
    + (direct) JSONParser.parse(String) line 14
    + (indirect) Object.equals(Object) line 17
  . (direct) PrintStream.println(boolean) line 7
```

(c) Differences in the form of a delta tree.

Figure 7. Two possible inlining trees of method `main` from the running example (Listing 3) and their delta tree.

As an illustration, consider two possible compilations of method `main` from the running example (Listing 3). Suppose that the first compilation unit duplicates the call to `literalsEqual` (shown in Listing 6). The second compilation unit does not perform a duplication. Instead, it inlines the call to `literalsEqual`. Listing 10 compares the optimization-context trees of these compilation units.

Listing 10. Delta tree of two optimization-context trees.

```
. (root) Example.main(String[])
  - Duplication line 7
  . (direct) Integer.parseInt(String) line 5
  * (direct -> inlined) Example.literalsEqual(int) line 7
    + LoopPeeling line 12 with {peelings: 1}
    + CanonicalReplacement line 12
    + CanonicalReplacement line 15
    + (direct) Scanner.init(InputStream) line 10
    + (direct) Scanner.nextLine() line 13
    + (direct) Scanner.nextLine() line 13
    + (direct) JSONParser.parse(String) line 14
    + (direct) JSONParser.parse(String) line 14
    + (deleted) Object.equals(Object) line 17
    + (indirect) Object.equals(Object) line 17
  - (direct) Example.literalsEqual(int) line 7
  - (direct) PrintStream.println(boolean) line 7
  . (direct) PrintStream.println(boolean) line 7
```

We can see a duplication (line 7) performed only in the first compilation unit. The calls to `literalsEqual` are matched, and the listing explicitly states that the first compilation did not inline but the second did. We can also see the optimization decisions and call sites in the inlined method. The tree clearly shows that the loop peeling was performed in differently inlined code. Finally, the call sites created by the duplication are present only in the first compilation unit.

3.4.4 Tree Edit Distance (TED).

We aim to semantically compare two optimization, inlining, or optimization-context trees in order to pinpoint the changes in compiler behavior. Due to aggressive inlining policies [34], compilation units in modern-day Java programs [5, 35] are often large, and thus they undergo hundreds of transformations. For this reason, comparing the trees using text-based tools that ignore the tree structure is insufficient. Instead, we employ TED [4] and define the tree operations we seek.

TED [4] solves the following problem. Let us have two labeled ordered trees T_1 and T_2. What is the minimum cost of operations to transform T_1 into T_2? The allowed operations are node deletion, node insertion, and node relabeling. The cost of the operations is given by a cost function.

In the 1-degree variant TED [38], only subtrees (rather than internal nodes) may be inserted or deleted. This variant matches our problem setting. The semantic differences between the trees we enumerated in earlier sections may be formulated as subtree operations or relabeling. We extend the original algorithm proposed for 1-degree TED [38] to compute a delta tree [8].

To compare two trees, we must define the cost function. Instead of defining the function in terms of labels, we define a function that returns whether two tree nodes are equal. If they are not equal, we define a function that returns the cost of "relabeling" the first node to the second. We set the cost of inserting or deleting a subtree with n nodes to the value n.

In the optimization tree, two nodes are equal if they are either optimization phases with the same identifier or optimization decisions with the same names, properties, and positions. In the inlining tree, two nodes are equal if their target methods, offsets, and call-site categories are equal. We use the relabeling operations on inlining tree nodes with matching target methods and offsets but differing call-site categories. In all other cases, the cost of relabeling is set to infinity. The cost function for the optimization-context tree is obtained by merging these definitions.

1-degree TED compares two ordered trees. However, the inlining tree is unordered, and some optimization phases perform transformations whose relative order is insignificant. Therefore, before computing TED, we sort all nodes in the inlining and optimization-context trees. For the optimization tree, we have a hand-picked list of phases whose children we sort. The sorting criterion is based on source-level positions (i.e., line numbers in this paper).

Profdiff can post-process the delta tree when the compiler engineer is interested only in the differences. This is done by iteratively removing unchanged leaf nodes from the delta tree so that only changes and their contexts are left. If the input trees are equivalent, the post-processed delta tree is empty.

3.5 Compilation Fragments

Inlining [34] enables many other optimizations by broadening the scope of the code that the compiler observes. For example, object allocations can be placed on the stack [41] provided the object does not escape the inlined scope. However, the same transformation might have been possible even if the inlining decisions differed.

Suppose that we have a hot method (with a dedicated hot compilation unit) in one of the experiments. In the other experiment, this method is inlined in another method. We want to compare the optimization decisions in the inlinee versus those in the dedicated compilation unit. However, the techniques introduced up to this point compare only two compilation units. To solve this problem, we present compilation fragments.

As an example, we show method `literalsEqual` from the running example (Listing 3) first compiled separately and then inlined in its caller, method `main`. We illustrate what optimization decisions the compiler might perform in each case and how we can use compilation fragments to compare these optimizations. Listing 11 shows the dedicated compilation unit of method `literalsEqual`. The compiler peeled the method's loop once, as illustrated in Listing 4.

Listing 11. Optimization-context tree for a dedicated compilation of method `literalsEqual`.

```
(root) Example.literalsEqual(int) line 7
  LoopPeeling line 12 with {peelings: 1}
  CanonicalReplacement line 12
  CanonicalReplacement line 15
  (direct) Scanner.init(InputStream) line 10
  (direct) Scanner.nextLine() line 13
  (direct) Scanner.nextLine() line 13
  (direct) JSONParser.parse(String) line 14
  (direct) JSONParser.parse(String) line 14
  (deleted) Object.equals(Object) line 17
  (indirect) Object.equals(Object) line 17
```

Now, consider the compilation unit of `main` shown in Listing 12. The compiler duplicates and inlines the call to method `literalsEqual` with the constant argument 0, which limits the number of loop iterations. The compiler removes the loop by peeling it and evaluating the condition to `false`.

Listing 12. Optimization-context tree after duplication, inlining, and deleting the loop in `literalsEqual`.

```
(root) Example.main(String[])
  Duplication line 7
  (direct) Integer.parseInt(String) line 5
  (inlined) Example.literalsEqual(int) line 7
    LoopPeeling line 12 with {peelings: 1}
    CanonicalReplacement line 12
    CanonicalReplacement line 12
    CanonicalReplacement line 15
    (direct) Scanner.init(InputStream) line 10
    (deleted) Scanner.nextLine() line 13
    (deleted) Scanner.nextLine() line 13
    (deleted) JSONParser.parse(String) line 14
    (deleted) JSONParser.parse(String) line 14
    (deleted) Object.equals(Object) line 17
    (deleted) Object.equals(Object) line 17
  (direct) Example.literalsEqual(int) line 7
  (direct) PrintStream.println(boolean) line 7
  (direct) PrintStream.println(boolean) line 7
```

We construct a *compilation fragment* from Listing 12 to compare the inlined method with the dedicated compilation unit. The compilation fragment is obtained by copying the subtree rooted in the inlined `literalsEqual` node. The subtree forms an optimization-context tree. As a result, we can compare the tree from Listing 11 with the just-constructed compilation fragment.

Listing 13. Delta tree of the dedicated compilation unit from Listing 11 and a fragment from Listing 12.

```
. (root) Example.literalsEqual(int)
  . LoopPeeling line 12 with {peelings: 1}
  + CanonicalReplacement line 12
  . CanonicalReplacement line 12
  . CanonicalReplacement line 15
  . (direct) Scanner.init(InputStream) line 10
  * (direct -> deleted) Scanner.nextLine() line 13
  * (direct -> deleted) Scanner.nextLine() line 13
  * (direct -> deleted) JSONParser.parse(String) line 14
  * (direct -> deleted) JSONParser.parse(String) line 14
  . (deleted) Object.equals(Object) line 17
  * (indirect -> deleted) Object.equals(Object) line 17
```

The delta tree in Listing 13 compares the dedicated compilation unit with the compilation fragment. The tree highlights the replacement of the loop condition with a constant. Thanks to the constant argument, the transformation was performed only in the fragment from Listing 12. We can see that all call sites inside the loop body were deleted in the compilation fragment.

Creating Fragments for Inlinees. Every inlinee is a potential compilation fragment. Creating fragments for all inlinees is infeasible. This section proposes a simple condition that determines when profdiff should create compilation fragments. We prove that, under certain assumptions, this condition is sufficient to compare all pairs of relevant method compilations.

Recall that profdiff marks frequently executed compilation units as hot. We say that a *method* is hot if there exists a hot compilation unit of that method in either experiment. We propose to leverage the hotness information to designate for which inlinees we should create fragments. Thus, the assumption is that a method is important only if the program spends a significant fraction of time executing it.

We suggest the following algorithm to create compilation fragments. For each hot compilation unit, we iterate over all nodes in its inlining tree that are marked as `inlined`. If the target method of this call site is hot, we create a compilation fragment rooted in this node. Then, compilation fragments whose root method is m shall be compared with all dedicated compilation units of method m in the other experiment. In summary, we compare all pairs of hot compilation units and all pairs of fragments and compilation units.

To show why this is sufficient, we must define what pairs of *method compilations* should be compared. A method compilation of method m is either a compilation unit rooted in m or a compilation unit that inlined m. Now, consider a program's global *call tree*. The call tree is a tree that is rooted in the entry point of a program. For each method invocation in the source code, we insert a child node representing the concrete invoked method to the node that calls the method. For indirect invocations, we insert nodes for all possible targets. For simplicity, assume that our workloads are deterministic.

A compilation unit contains method compilations, i.e., the compiled root method and the inlined methods. We can represent its method compilations as a call tree. The compilation unit's call tree is a subgraph of the global call tree. We say that a compilation unit *covers* some part of the global call tree. Note that a compilation unit may cover several parts of the call tree.

Let us have a call-tree node m, which represents some method. Let c_1 be any hot compilation unit from experiment 1 that covers m, and let c_2 be any hot compilation unit from experiment 2 that covers m. Their root methods are m_1 and m_2, respectively. We will use m, m_1, and m_2 to refer to the nodes or methods they represent interchangeably. Both c_1 and c_2 contain the code of method m. Therefore, it is desirable to compare the optimization decisions in method m. The situation is depicted in Figure 8: there are two copies of the global call tree, and the subtrees covered by c_1 and c_2 are highlighted using dashes.

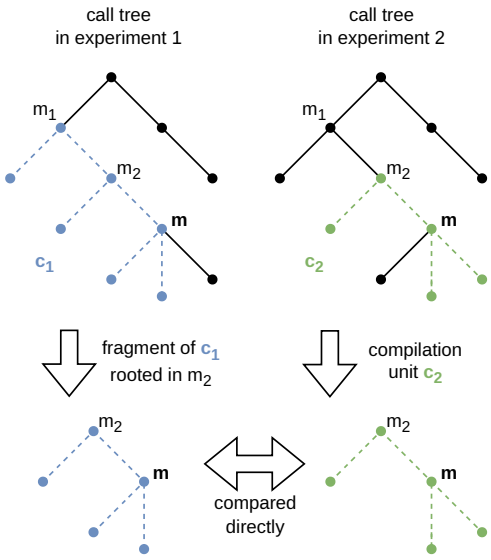

Figure 8. Method m and two compilations units, c_1 and c_2, that cover m. A compilation fragment is created from c_1 so that the optimization decisions in m are comparable.

We claim that using the rules we defined above, the optimization decisions in m will always be compared. We prove the claim by enumerating all possible cases. If $m_1 = m_2$, the hot compilation units c_1, c_2 are compared directly. If $m_1 \neq m_2$, it holds that either c_1 contains m_2 or c_2 contains m_1. This is because both call trees contain m. We assume w.l.o.g. that c_1 contains m_2, as depicted in Figure 8. We know that m_1 is a

hot method and m_2, which is inlined in the compilation unit of m_1, is also hot. Therefore, a compilation fragment rooted in m_2 is created from c_1. The compilation fragment will be compared with c_2.

4 Case Studies

This section presents several case studies where we applied profdiff. We focused on workloads from the Renaissance benchmark suite [35] because it is used to track the performance metrics of the Graal compiler. Graal compiler developers reported six workloads that exhibit performance fluctuations between VM invocations. Such workloads are non-deterministically faster or slower on each invocation, which is considered a bug. These issues have been open for more than a year at the time of our investigation. During that time, the performance distributions of some of these workloads shifted toward the slower state, leading to an overall regression. We may execute Graal with different Java Development Kit (JDK) versions, i.e., different library and VM revisions. The performance distribution of a workload might vary between JDK versions.

Profdiff is a good fit for such fluctuating workloads. First, we execute the VM with the workload several times to sample various compilation outcomes. Second, we cluster the runs into *slow* and *fast* based on the collected performance metrics. Then, we inspect several pairs of the runs using profdiff. Some variability regarding optimization decisions between hot compilations is expected. However, the goal is to identify which optimization and inlining differences are consistent between the slow and fast runs. These decisions are likely responsible for the performance gap. Finally, if possible, we override these decisions using compiler options and measure if this leads to a performance improvement.

We present three out of six fluctuating workloads from the Renassaince benchmark suite [35] where we could pinpoint and confirm a problem. We confirmed the findings by overriding the optimization decisions of the compiler, which led to stable improvements of 8% to 30%. In another workload, we identified the likely cause but could not override the suspected decisions to confirm the findings. We reported all these problems to the Graal team so that they may be fixed in a future release of the compiler.

4.1 Gauss Mix

The workload `gauss-mix` from Renaissance [35] fluctuated with slow and fast states approximately 30% apart. Over time, the workload got stuck in a slower state. We repeated the benchmark 30 times, and we found a single run that was about 30% faster than the rest. Profdiff uncovered that in the fast run, the compiler inlined several hot methods into a single compilation unit. In contrast, the slower runs spent time in several dedicated compilation units.

Profdiff created compilation fragments from the compilation unit in the fast run and compared them with the dedicated compilation units from the slower runs. Some of these dedicated compilation units did not inline more than the respective compilation fragments. We found that the sum of time fractions spent in such compilation units was higher than the time spent in the single compilation unit from the fast run. This is a clue that the compiler should always inline these calls into a single compilation unit.

We verified the findings by forcing the compiler to inline the identified methods using a command-line option. Simply forcing the compiler to inline these methods appeared insufficient because the compiler would halt compilation due to excessive graph size. Therefore, using another command-line option, we also forbade the compiler from inlining the root method of the single compilation unit in the fast run. This led to a consistent speed-up of about 30%.

4.2 Scala K-Means

The workload `scala-kmeans` from Renaissance [35] exhibited fluctuations on both JDK 11 and JDK 17. It regressed to a slower state in a later compiler version with JDK 17. We repeated the workload 30 times on JDK 11 and found four runs that were about 8% faster than the rest. Using profdiff, we identified five methods that were consistently inlined in the fast runs but not inlined in the slow runs.

We verified the findings by forcing the compiler to inline the identified methods using a command-line option. Force-inlining the particular method was sufficient to achieve the fast state consistently. Although we performed the experiments on JDK 11, we confirmed that overriding the inlining decision also speeds up the workload on JDK 17, where we achieve a speed-up of about 8%.

4.3 Scala Doku

The workload `scala-doku` from Renaissance [35] fluctuates between VM runs, with a possible speed-up of about 30% relative to the slow runs. We repeated the workload 30 times and found an apparent inlining difference between the fast and slow runs. The fast runs always inlined two methods related to iterators, but the slow runs never inlined them. Both calls were indirect through the iterator interface.

A likely reason for the different inlining decisions is the type profiles at the indirect call sites. The type profiles guide the decisions related to devirtualization. The profiles for the indirect call sites shown by profdiff differed significantly. The estimated probability of the method that should have been inlined was about 35% in one of the fast runs but only about 2% in a slow run, which did not inline the method. Thus, the workload's performance is likely linked to these inlining decisions, and their instability is, in turn, related to the type profiles. We confirmed the findings by forcing the compiler to inline the two target methods of the indirect call. We achieved this by extending the compiler with a new

option to force the devirtualization of selected calls. This led to a consistent speed-up of about 30%.

5 Related Work

The goal of performance bug detection in compilers is to discover poorly performing compilations. We give an overview of existing methods in Section 5.1. Performance diagnosis in general [16] aims to analyze and fix already discovered performance bugs. We talk about these methods in Section 5.2. Our work can be categorized as performance diagnosis in compilers. In contrast to performance diagnosis in general software, we focus on the performance of the compiled program rather than the compiler's performance. To the best of our knowledge, this is the first work that diagnoses compiler performance by capturing and comparing its behavior.

Replay compilation [15, 20, 27, 28, 31, 37] facilitates performance debugging by making it possible to reproduce prior compiler behavior. This is done by recording nondeterministic runtime information, such as profiles, during a VM invocation. The recorded information is reusable in another VM invocation to produce equivalent compilations. Compiler engineers might first use profdiff to narrow down a performance problem, then replay the affected compilation with a debugger or an IR visualizer [49]. Performance may fluctuate between VM invocations, so we must sample different compilation outcomes [15], whereas replay compilation reproduces the same outcome. Therefore, the issue addressed by replay compilation is distinct from the one solved by profdiff.

Mosaner et al. [24] present an approach to improve optimization decisions in a dynamic compiler. They compile and run methods with different optimization decisions. The extracted execution statistics are used to train or fine-tune a machine-learning model.

5.1 Performance Bug Detection in Compilers

There are several ways to detect performance bugs in compilers. Black-box approaches make it possible to compare different compilers. NULLSTONE [9] is a test suite covering individual compiler optimizations. In random testing, small programs are randomly generated, compiled, and checked for potential issues. Differential testing is a type of random testing. Test cases are generated and compiled in two different settings (e.g., by two different compilers or compiler versions). Various methods are employed to compare the compiled executables. If there is a difference, the test case is automatically reduced to trigger the bug in fewer lines of code. Barany [2] statically compares binaries by performance-related criteria such as instruction count, the number of arithmetic operations, or memory accesses. Kitaura and Ishiura [21] statically detect dissimilar code sections to detect potential performance differences. Then, they execute the code to verify the findings. Theodoridis et al. [45] instrument the source code with markers and check whether the compiler eliminates the markers as dead code. An optimization opportunity is reported if two compilers remove different sets of markers.

Hashimoto and Ishiura [17] employ an equivalence-based method, which is a type of random testing. They prepare an optimized and unoptimized version of the same C program and compare the generated code. If the compiler fails to optimize the unoptimized version of the program, a problem is reported. If an issue is detected, the source code is automatically reduced to isolate the problem.

Moseley et al. [25] collect profiles from executables compiled by different compilers, compiler versions, and configurations. The profiles comprise the instruction mix, control-flow edge counts, and data from hardware performance counters. They detect anomalies in these profiles to uncover performance problems. The technique is limited to compilation units with equivalent inlining.

Taneja et al. [44] employ a white-box approach, computing static analyses on code fragments and comparing the results to the analyses computed by LLVM. This way, they uncover soundness issues or missed optimization opportunities.

5.2 Performance Diagnosis in General Software

Tools aimed at application developers may also detect performance problems in programs and potentially suggest fixes. Yu and Pradel [50] present an approach based on profiling to pinpoint root causes of synchronization bottlenecks in concurrent applications. Nistor et al. [26] introduce a method to detect loops that can be exited early and suggest possible source-code fixes. Curtsinger and Berger [10] show a method to evaluate the potential impact of speeding up particular lines of code in multi-threaded applications. Song and Lu [39] present a tool to detect inefficient loops and suggest fix strategies. Della Toffola et al. [11] present a tool that suggests memoization for Java methods that repeat computations. Tan et al. [43] introduce a tool to mark useless memory operations. Wen et al. [48] present a technique to discover redundant computations.

6 Conclusion

We introduce the problem of identifying the causes of performance regressions in modern compilers. We present a solution based on tracking optimization decisions, where the decisions are represented as trees capturing the structure of either the optimizer or the inlined code, and we propose techniques to compare differently inlined code. The tooling is implemented as an open-source part of the Graal compiler. We evaluate the techniques with industry-standard benchmarks and describe three instances where we pinpointed the decisions causing performance problems. The workloads are about 8% to 30% faster when these decisions are overridden.

References

[1] Milad Abdullah, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Vojtěch Horký, and Petr Tůma. 2022. Reducing Experiment Costs in Automated Software Performance Regression Detection. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 56–59. https://doi.org/10.1109/SEAA56994.2022.00017

[2] Gergö Barany. 2018. Finding Missed Compiler Optimizations by Differential Testing *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3178372.3179521

[3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (10 2017), 27 pages. https://doi.org/10.1145/3133876

[4] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1 (2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

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

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

[7] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.* 25, 2 (6 1996), 493–504. https://doi.org/10.1145/235968.233366

[8] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 493–504. https://doi.org/10.1145/233269.233366

[9] Nullstone Corporation. 2012. *NULLSTONE for Java.* http://www.nullstone.com/htmls/ns-java.htm

[10] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/2815400.2815409

[11] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. *SIGPLAN Not.* 50, 10 (10 2015), 607–622. https://doi.org/10.1145/2858965.2814290

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

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

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[15] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java Performance Evaluation through Rigorous Replay Compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 367–384. https://doi.org/10.1145/1449764.1449794

[16] Xue Han, Tingting Yu, and Gongjun Yan. 2023. A systematic mapping study of software performance research. *Software: Practice and Experience* 53, 5 (2023), 1249–1270. https://doi.org/10.1002/spe.3185

[17] Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting Arithmetic Optimization Opportunities for C Compilers by Randomly Generated Equivalent Programs. *IPSJ Transactions on System and LSI Design Methodology* 9 (2016), 21–29. https://doi.org/10.2197/ipsjtsldm.9.21

[18] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[19] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Not.* 29, 6 (6 1994), 326–336. https://doi.org/10.1145/773473.178478

[20] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Program Locality. *SIGPLAN Not.* 39, 10 (10 2004), 69–80. https://doi.org/10.1145/1035292.1028983

[21] Kota Kitaura and Nagisa Ishiura. 2018. Random Testing of Compilers' Performance Based on Mixed Static and Dynamic Code Comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 38–44. https://doi.org/10.1145/3278186.3278192

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

[23] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[24] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/3546918.3546921

[25] Tipp Moseley, Dirk Grunwald, and Ramesh Peri. 2009. OptiScope: Performance Accountability for Optimizing Compilers. In *2009 International Symposium on Code Generation and Optimization*. 254–264. https://doi.org/10.1109/CGO.2009.26

[26] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 902–912.

[27] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. 2006. Replay Compilation: Improving

Debuggability of a Just-in-Time Compiler. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 241–252. https://doi.org/10.1145/1167473.1167493

[28] Oracle. 2019. *Replay Compilation in HotSpot JVM.* https://github.com/openjdk/jdk/blob/master/src/hotspot/share/ci/ciReplay.hpp

[29] Oracle. 2023. *Graal Compiler.* https://www.graalvm.org/latest/reference-manual/java/compiler/

[30] Oracle. 2023. *GraalVM.* https://www.graalvm.org/

[31] Oracle. 2023. *Profile Replay in GraalVM.* https://github.com/oracle/graal/blob/master/compiler/src/jdk.internal.vm.compiler/src/org/graalvm/compiler/hotspot/ProfileReplaySupport.java

[32] Oracle. 2023. *proftool.* https://github.com/graalvm/mx/blob/master/README-proftool.md

[33] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1.

[34] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 164–179. https://doi.org/10.1109/CGO.2019.8661171

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

[36] John Rose. 2013. *MethodData.* https://wiki.openjdk.org/display/HotSpot/MethodData

[37] Narendran Sachindran and J. Eliot B. Moss. 2003. Mark-Copy: Fast Copying GC with Less Space Overhead. *SIGPLAN Not.* 38, 11 (10 2003), 326–343. https://doi.org/10.1145/949343.949335

[38] Stanley M. Selkow. 1977. The tree-to-tree editing problem. *Inform. Process. Lett.* 6, 6 (1977), 184–186. https://doi.org/10.1016/0020-0190(77)90064-3

[39] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 370–380. https://doi.org/10.1109/ICSE.2017.41

[40] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[41] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[42] Sun Microsystems, Inc. 2006. *The Java HotSpot™ Performance Engine Architecture.* https://www.oracle.com/java/technologies/whitepaper.html

[43] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2020. What Every Scientific Programmer Should Know about Compiler Optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 42, 12 pages. https://doi.org/10.1145/3392717.3392754

[44] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3368826.3377927

[45] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3503222.3507764

[46] Igor Veresov. 2013. . https://www.slideshare.net/maddocig/tiered

[47] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2020. Exploring Impact of Profile Data on Code Quality in the HotSpot JVM. *ACM Transactions on Embedded Computing Systems* 19, 6, Article 48 (10 2020), 26 pages. https://doi.org/10.1145/3391894

[48] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 254–265. https://doi.org/10.1109/PACT.2015.29

[49] Thomas Würthinger. 2007. *Visualization of Program Dependence Graphs.* Master's Thesis. Johannes Kepler University Linz. https://ssw.jku.at/Research/Papers/Wuerthinger07Master/Wuerthinger07Master.pdf

[50] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering* 23, 5 (01 10 2018), 3034–3071. https://doi.org/10.1007/s10664-017-9578-1

Received 2023-06-29; accepted 2023-07-31