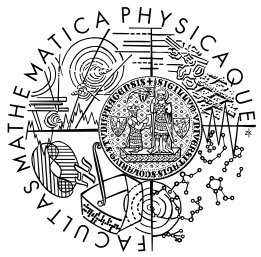

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Andrej Pečimúth

Optimization Decision Analysis for Graal

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	David Leopoldseder, Dr.
Study program:	Computer Science
Study branch:	Software Systems

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank David Leopoldseder, Petr Tůma, and the people from D3S and Oracle Labs for all their support and feedback.

Title: Optimization Decision Analysis for Graal

Author: Bc. Andrej Pečimúth

Department: Department of Distributed and Dependable Systems

Supervisor: David Leopoldseder, Dr., Oracle Labs

Consultant: prof. Ing. Petr Tůma, Dr., Department of Distributed and Dependable Systems

Abstract: Modern compilers apply a set of optimization passes aiming to speed up the generated code. The combined effect of individual optimizations is often unpredictable. Thus, changes to a compiler's code may hinder the performance of generated code as an unintended consequence. Due to the vast number of compilation units and applied optimizations, it is difficult to diagnose these regressions.

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 thesis introduces an approach utilizing tree edit distance (TED) to detect optimization differences in a semi-automated way. Since the same source code may be inlined in different contexts and optimized differently in each, we also present an approach to compare optimization decisions in differently inlined code. We employ these techniques to pinpoint the causes of performance problems in various benchmarks of the Graal compiler.

Keywords: compiler, optimization

Contents

Introduction

Compilers rely on optimizations to generate efficient machine code. Optimizations are transformations on the level of intermediate representation (IR) [1] aiming to speed up the code. Modern dynamic compilers use elaborate heuristics incorporating profiling feedback to determine which transformations are beneficial [2, 3]. Such compilers can make speculative decisions [4] to further improve compilation outcomes. The quality of optimization decisions is a crucial factor in determining the speed of a compiled program.

Compilers under active development, such as Graal [5], 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 thesis, 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] to identify performance-affecting commits. The computation-time costs of regression testing [7] are significant and thus cannot be conducted for each commit. Instead, compiler performance is measured [6] across one or more merge commits.

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) due to the inherent non-determinism of the environment.

Performance problems are often related to frequently executed code. To investigate a regression, a compiler engineer might profile the workload [8] 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 [9] 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 [3] 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 [10] tree edit distance (TED).

Due to inlining [3], 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.

There are several additional use cases for our approach. Besides the regression scenario, Graal workloads compiled just-in-time (JIT) often exhibit multimodal performance distribution. In order to stabilize such workloads, it is essential to identify which optimization decisions are indispensable to achieving peak performance. With profdiff, we can find the optimization decisions that differ between the well-performing and poorly-performing compilations. Another use case is bridging the gap between Graal's ahead-of-time (AOT) and JIT capabilities. Comparing an AOT compilation with a JIT compilation of the same application allows us to identify missed optimization opportunities in the AOT compilation.

We implemented the described methods for the Graal compiler [5]. The implementation[1] is available in the open-source compiler repository. 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 observing 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 thesis 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.

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

Chapter 1

Background

This chapter provides the necessary background related to performance tracking for dynamic compilers. We focus on the Graal compiler [5], which is an example of a modern compiler for the Java platform [11]. Graal compiles Java bytecode to machine code. It can operate either as a JIT compiler (part of *GraalVM* [12]) or an AOT compiler (part of *Native Image* [13]). We introduce Graal in the context performance tracking. For a broader introduction, we recommend reading "Simulation-Based Code Duplication in a Dynamic Compiler" by Leopoldseder [14] or "Partial Escape Analysis and Scalar Replacement for Java" by Stadler [15].

In contrast to native applications, dynamic environments pose a challenge for performance evaluation. Modern VMs often utilize an interpreter and multiple JIT compilers [16]. Due to this and also other factors, JIT-compiled workloads often exhibit performance fluctuations [17] during or between VM invocations. Section 1.1 introduces the relevant aspects of GraalVM. The VM's inherent non-determinism complicates performance tracking. Section 1.2 explains how Graal's performance is tracked and analyzes potential causes of performance regressions.

Native Image [13] is interesting in the context of this thesis, as it enables us to run identical workloads in much contrasting settings and analyze the differences. The technology makes it possible to create native executables from Java applications. The advantage of Native Image is minimal startup time of the compiled application and reduced resource usage. This is particularly beneficial for short-lived applications, such as serverless cloud applications. Moreover, we can expect fewer performance fluctuations. As Java is a dynamic language, there is an additional constraint: Native Image must know all reachable classes and uses of dynamic features at compile time. Neither the bytecode nor JIT compilers are available at runtime, which hinders speculative optimization.

1.1 GraalVM

GraalVM is a Java Virtual Machine (JVM) based on the Java HotSpot Virtual Machine [18]. As a result, much of the information presented here applies to both HotSpot and GraalVM. The primary difference is that in GraalVM, the Graal compiler [5] replaces HotSpot's server compiler [19].

GraalVM utilizes tiered compilation [16] comprising an interpreter and two just-in-time compilers. The execution of a method starts in the interpreter. The interpreter collects profiling information [20], 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 [21]. The client compiler generates instrumented code that also collects profiles. After the method-invocation counter passes another threshold, the method is compiled by the Graal compiler [5], striving for peak performance. The reverse step is also possible: execution may be transferred from compiled code to the interpreter (deoptimization). Figure 1.1 sums up the execution transfers between the interpreter and compiled code. Note that Figure 1.1 is a simplification, and the actual compilation policy [16] is more complex.

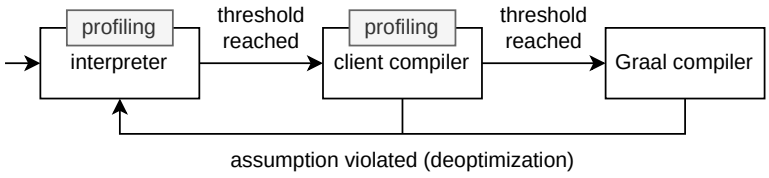

Figure 1.1 Tiered execution and execution transfers in GraalVM.

Tiered compilation leads to a warm-up phase [17] in some applications. It has also been shown that some workloads do not reach a steady state at all [17]. 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 [22]. To measure the performance of a workload, it is necessary to invoke the VM several times and also repeat the workload during a single VM invocation [7, 17].

The interpreter and code compiled by the client compiler collect profiles that guide optimization decisions [2, 3, 23] during subsequent compilation. For example, the profiles are used to estimate the relative probability of an instruction [2]. 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 [3] to devirtualize indirect call sites. Thus, the quality of generated code depends [23] 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 [23]. The window during which profiles are collected is also subject to factors such as the timing of compilation jobs or deoptimization [24].

Deoptimization [24] allows JIT compilers to optimize aggresively based on speculative assumptions. For example, consider the Java method in Listing 1.

```
1  int first(Object[] objects) {
2    return (Integer) objects[0];
3  }
```

Listing 1 Method `first` returns the first array element cast to an integer.

As mandated by the language specification [25], the above method performs an implicit bounds check, cast check, and two null checks. If a check fails, the method throws an exception. In case the VM does not record a failed check, the compiler speculatively assumes the check never fails. Thus, the compiled code only checks the condition and *deoptimizes* on failure. Listing 2 illustrates how a speculative compiler might translate the example (with explicitly marked deoptimization).

```
1  int first(Object[] objects) {
2    if (objects == null) {
3      deoptimize;
4      // The interpreter throws a NullPointerException.
5    }
6    if (objects.length == 0) {
7      deoptimize;
8      // The interpreter throws an ArrayIndexOutOfBoundsException.
9    }
10   Object temp1 = objects[0];
11   if (!(temp1 instanceof Integer)) {
12     deoptimize;
13     // The interpreter throws a ClassCastException.
14   }
15   Integer temp2 = (Integer) temp1
16   if (temp2 == null) {
17     deoptimize;
18     // The interpreter throws a NullPointerException.
19   }
20   return temp2.intValue();
21 }
```

Listing 2 Method `first` with explicit deoptimization.

When an assumption is violated, the compiled code transfers control (deoptimizes) back to the interpreter. The VM records the violated assumption, and the interpreter proceeds with execution. In our example, the interpreter throws the appropriate exception. Later, the compilation policy may decide to recompile the method. This time, however, the JIT compiler does not make the same speculative assumption.

In the context of performance evaluation, deoptimization is yet 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.

1.2 Performance Regressions

There are several changes merged to the Graal compiler daily. Compiler developers track changes in metrics such as wall clock time, compile time, code size, and resident set size [26]. In this thesis, we focus on the wall clock time to execute a representative workload [27]. Graal uses an automated pipeline to test for performance changes [6] regularly. Workloads from benchmark suites such as Renaissance [27] are repeated several times for selected compiler configurations and target platforms. The challenges of performance tracking include handling warm-up [17] of the JIT compiler and various sources of variance. For these reasons, performance tracking incurs a high cost in terms of machine time. Statistical methods are employed [7] to detect performance changes in either direction.

Detecting a regression is the first part of the problem. Finding the cause of the regression is the task that follows. However, this is complicated due to the large size and number of compilation units, the non-determinism of the environment, and the lack of suitable data from the compiler.

In a typical Renaissance [27] workload, there may be hundreds to thousands of compilation units. Usually, only few of these compilation unit characterize the performance of the workload. Compiler engineers may determine which compilation units take most of the execution time using a profiler. These characterics may change between VM invocations due to the instability of inlining decisions. Additionally, performance anomalies may manifest randomly only in some VM invocations. As a consequence, all information that may be needed must be collected from a single VM invocation and analyzed after the fact.

The options to collect relevant information from the compiler include general-purpose log messages, structured inlining decisions, and IR graphs. There are no tools to integrate profiles with either of these options to high-

light essential compilation units. Although the log messages are organized in trees, they are purely textual and do not follow a common pattern. The inlining decisions are structured as call trees, where each call site is associated with relevant inlining decisions. Such trees are suitable for comparison based on tree-matching algorithms.

Lastly, the compiler has an option to serialize the IR graphs of compilation units. These graphs are viewable in Ideal Graph Visualizer [9], which can also compare them. The drawbacks are the size of the graphs, a relatively low level of abstraction, and method inlining [3]. This is because a compilation unit may contain thousands of IR nodes and some high-level optimizations produce many IR-level changes. For example, consider the IR-level difference resulting from loop unrolling [28]: the graph difference comprises all nodes in the loop's body. Moreover, method inlining involes replacing a call node with the body of the target method. Thus, it may be difficult to compare compilation units that inlined different methods.

Consider how a commit merged to the compiler affects the performance of compiled code. 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.

Due to the non-determinism of the VM, a performance problem may not manifest itself in each invocation of the VM. We observe this as performance fluctuations between VM invocations. Every time a workload is executed, compilation outcomes are different and performance deviations may be significant. Performance distribution could shift as changes are merged into the compiler, i.e., the workload would regress on average. A compiler engineer can try to diagnose such problems by repeating the workload several times and classifying each run as *fast* or *slow*. The optimization differences that are consistent between fast and slow runs are likely related to the cause of the performance instability. However, pinpointing consistent differences using the available tools is challenging.

Chapter 2

Profdiff

This chapter introduces profdiff, an approach to capture and compare optimization decisions. We provide an implementation for the Graal compiler, which is available[1] in the open-source Graal repository.

We have extended the Graal compiler with an option to collect and store optimization logs. The optimization logs contain an inlining tree (Section 2.2) and an optimization tree (Section 2.1) for each compilation unit. The inlining tree is a call tree that describes inlining decisions in a compilation unit. The optimization tree shows optimization decisions structured according to the dynamic execution flow of the optimizer. We can also associate optimization decisions with the application code they affect using the optimization-context tree (Section 2.3). 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 from Graal, which is the top-tier compiler. 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 [8], 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*.

[1] `https://github.com/oracle/graal`

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

Figure 2.1 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.

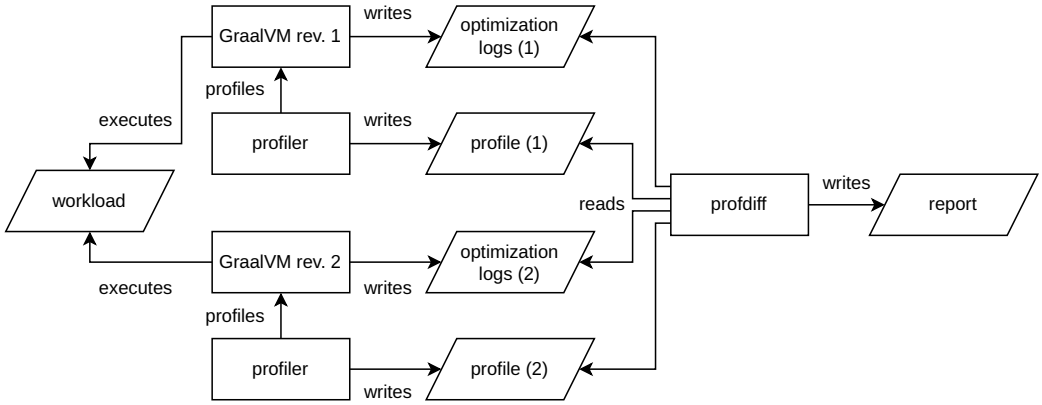

Figure 2.1 Executing and comparing two experiments.

The scenario in Figure 2.1 depicts a comparison of two JIT experiments. Optimization logs in an AOT scenario are collected similarly, except we profile the built executable instead of the VM. The structure of JIT and AOT logs is equivalent. As a result, profdiff can compare an AOT experiment with a JIT experiment or two AOT experiments.

Figure 2.2 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 m_1 from Figure 2.2, 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 [10] with some pre- and postprocessing to compare the trees. The result of the comparison is another

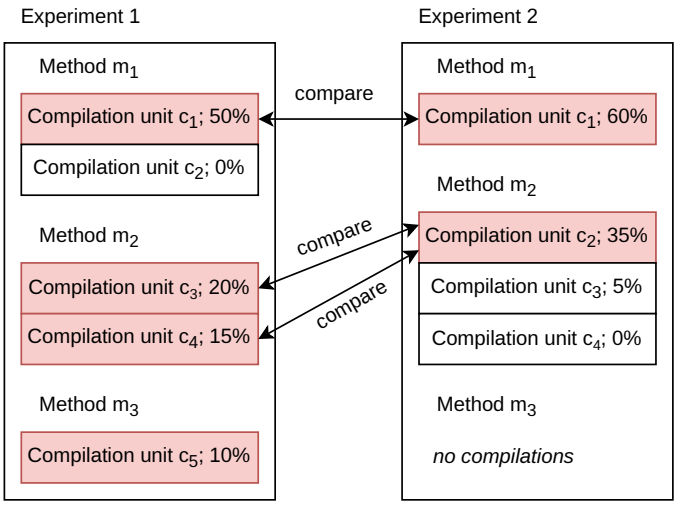

Figure 2.2 Comparing hot compilation units.

tree that conveys which optimizations the compilations have in common and which optimizations are different. Section 2.4 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 2.2 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 2.5.

2.1 Capturing Optimization Decisions

Compilers use an IR to represent the semantics of the compiled program. The Graal compiler uses a graph-based IR [1]. 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 [2] or inlining [3], decide whether a transformation is worth applying by estimating its cost and benefit. These estimates are prone to instability as described in Section 1.1. 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 Figure 2.3a. 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 [28], 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 Figure 2.3b.

Figure 2.4 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 2.4 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).

An applied transformation often enables new optimization opportunities. An example of this is duplication enabling conditional elimination, as we have shown in Figure 2.3a and Figure 2.3b. 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,

```
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 }
```

(a) Unoptimized code.

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

(b) Optimized code.

Figure 2.3 Example of a duplication and subsequent optimizations.

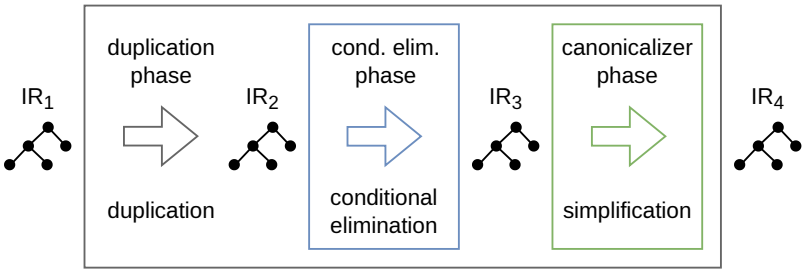

Figure 2.4 Composition of optimization phases.

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.

2.1.1 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 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.

```
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 }
```

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

For example, it might be worth peeling [28] 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.

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

```
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  }
```

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

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). The positions must be assigned consistently, which will become important later when we compare two optimization trees.

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`.

When only one method is compiled, the line number (or an instruction offset)

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

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

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 `main` at line 7. If `main` additionally invoked `literalsEqual` at a different line, different positions would distinguish the optimizations in the inlined code.

2.1.2 Inlining and Optimization Trees

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. It is necessary to update the positions of optimization decisions in the copied tree so that they reflect the new context.

For example, suppose that the compiler first peels the loop in Listing 3 and then inlines the method into `main`. A snippet of the possible optimization tree for method `main` is shown in Listing 6. The listing shows that the compiler attached the optimization tree of `literalsEqual` to the `InliningPhase`, which performed the inlining.

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

Listing 6 Optimization tree after inlining an optimized IR graph from the running example (Listing 3).

2.2 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 [3].

Listing 7 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.

```
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  }
```

Listing 7 After duplication in method `main` from the running example (Listing 3).

Capturing inlining information is a necessity in the context of comparing optimization decisions. Inlining may enable optimization opportunities that would not be otherwise possible. Moreover, for each method, there is also a set of optimizations that the compiler performs regardless of the inlining context. When we compare two compilation units with different inlining decisions, we may be presented with several optimization differences. It may not be immediately clear which optimizations were enabled by inlining and which optimizations are merely results of compiling more code.

```
RootPhase
  InliningPhase
    RootPhase
  LoopPeelingPhase
    LoopPeeling line 12 with
      {peelings: 1}
  ...
```

(a) `literalsEqual` inlined.

```
RootPhase
  InliningPhase
  LoopPeelingPhase
  ...
```

(b) `literalsEqual` not inlined.

Figure 2.5 Optimization trees of method `main` from the running example (Listing 3) with different inlining decisions.

For example, consider two compilations of method `main` from the running example (Listing 3). Their optimization trees are shown in Figure 2.5. The compilation unit in Figure 2.5a inlined `literalsEqual` but the compilation unit in Figure 2.5b did not inline `literalsEqual`. The compiler peeled the inlined loop in Figure 2.5a, but there was no loop to be peeled in Figure 2.5b. The loop from method `literalsEqual` could be peeled even if `literalsEqual` was compiled separately. The loop peeling was not enabled by inlining, but it is merely a result of compiling more code in the compilation unit. Thus, to understand optimization differences, we must capture inlining decisions.

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 8 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.

```
(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
```

Listing 8 Inlining tree of method `main` from the running example (Listing 3).

We store a list of inlining decisions for each node (omitted in Listing 8). 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 callsite in the compiled method body. As an illustration, Listing 9 shows the inlining tree of method `main` just after the method is parsed. 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.

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

Listing 9 Inlining tree after parsing method `main` from the running example (Listing 3).

Non-inlined callsites are linked to the method invocation nodes from the IR that represent them. The compiler might delete such a node from the IR, e.g., when it is in an unreachable branch. These nodes are classified as `deleted`. For example, suppose we peel the loop in Listing 3. The result of this transformation is shown in Listing 4. Notice that some of the call sites were duplicated. The call to `Object.equals` in line 17 may be removed from the peeled iteration because `i == 0` holds in the first iteration. Listing 10 shows the inlining tree after these transformations. There are duplicate inlining-tree nodes for the calls in the peeled loop, and one of the calls to `Object.equals` is marked as `deleted`.

```
(root) Example.literalsEqual(int)
  (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
```

Listing 10 Inlining tree of method `literalsEqual` from the running example (Listing 3) after loop peeling.

2.2.1 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 8).

GraalVM [12] 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. As an illustration, Listing 11 shows a type profile for the indirect call `Object.equals` from the JSON example. The type profile captures a receiver type (e.g., `Integer`), the estimated frequency of the receiver (80%), and the concrete method invoked for the receiver type (`Integer.equals`).

```
(indirect) Object.equals(Object) line 17
  80% Integer -> Integer.equals(Object)
  15%  String -> String.equals(Object)
   5%    List -> List.equals(Object)
```

Listing 11 Indirect inlining-tree node with a type profile.

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 [4]. 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 12 shows the inlining tree after the transformation.

A polymorphic call is devirtualized by replacing it with a type switch (an if-cascade with type checks) for the receiver type [29, 3]. Each branch of the switch leads to a direct inlinable call and possibly to a virtual call or deoptimization as a fallback. Listing 13 shows an if-cascade that could replace the indirect call to `Object.equals` from the JSON program.

Consider the direct invocations created by the devirtualization of an indirect call. We attach the direct invocations to the tree as the children of the indirect callsite. Listing 14 shows the result of inlining all recorded receivers enumerated

```
(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
```

Listing 12 Inlining tree of method `literalsEqual` from the running example (Listing 3) after type-guarded inlining.

```
if (literal instanceof Integer) {
  return ((Integer) literal).equals(first);
} else if (literal instanceof String) {
  return ((String) literal).equals(first);
} else if (literal instanceof List) {
  return ((List) literal).equals(first);
} else {
  deoptimize;
}
```

Listing 13 Devirtualization of the call to `equals` using an if-cascade with type checks.

in Listing 11. By attaching the new nodes as children, we convey the fact that they were created by devirtualization. This improves interpretability when the trees are compared.

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

Listing 14 Inlining-tree snippet for a devirtualized call site.

2.3 Optimizations in Context

Each transformation performed by the compiler affects a set of IR nodes. As explained in Section 2.1, 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. The optimization decisions are attached to the method whose code they transformed. 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 compilations 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 2.6c is built from the trees in Figure 2.6a and Figure 2.6b. 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. The position of an optimization contains the method context and an offset (line number) in the method where the optimization was performed. For example, the method context of `LoopPeeling` in Figure 2.6b is method `literalsEqual` invoked at line 7 in method `main`. The optimization decision was performed in that method context, at line 12. Thus, in the optimization-context tree (Figure 2.6c), the `LoopPeeling` is attached to the node representing `literalsEqual` invoked in method `main` at line 7.

2.3.1 Handling Duplicate Paths

A problem that may arise during the construction of an optimization-context tree is ambiguity after code duplication. Suppose that the call to `literalsEqual` were duplicated due to a code transformation. Now, we have two calls to `literalsEqual` in matching method contexts. If both calls are inlined, and the code of one of the callees is optimized, the position of such an optimization is ambiguous. We cannot distinguish the call sites using the positions because the positions match both call sites.

Listing 15 shows an optimization-context tree for such a situation. There are two inlined calls to `literalsEqual`. The loop peeling cannot be unambiguously linked to either call. Extending the compiler to create unique positions for duplicated IR nodes would be impractical. Therefore, we print a warning for every ambiguous call site and link the optimization with all matching call sites.

We say that the path to the call sites is *duplicate* because the method names

```
(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 2.6 Inlining, optimization, and optimization-context trees 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
    Warning: Duplicate path
    LoopPeeling line 12 with {peelings: 1}
    ...
  (inlined) Example.literalsEqual(int) line 7
    Warning: Duplicate path
    LoopPeeling line 12 with {peelings: 1}
    ...
  (direct) PrintStream.println(boolean) line 7
  (direct) PrintStream.println(boolean) line 7
```

Listing 15 Optimization-context tree with duplicate paths (snippet).

and line numbers from the root node to the call sites are equal for both call sites. If one of the calls was not inlined, we would not report a duplicate path. This is because optimization decisions are attached to *inlined* call sites only.

The path to a particular call site may be duplicate in one compilation unit but not duplicate in another compilation unit. Thus, this is a property we can compare between two optimization-context trees. To enable this, the warnings for duplicate paths are implemented as another type of node attached to the tree. The tree-matching algorithm compares the warning nodes like any other kind of node.

2.4 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 [30], which is a tree representation of optimization and inlining decisions that are either different or identical.

2.4.1 Comparing Optimization Trees

Recall that the optimization tree captures the applied optimization decisions, phases, and their relative order. If an optimization decision does not have a

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

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 2.7a shows a possible optimization tree of method `literalsEqual` from the running example (Listing 3). Figure 2.7b captures a regression scenario (i.e., the second experiment): the order of the canonicalizer and loop-peeling phases is reversed. Figure 2.7c represents the differences as a delta tree [31]. 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.

2.4.2 Comparing Inlining Trees

Recall that the inlining tree captures inlining decisions. The root node represents the root compiled method. Each non-root node represents a call site in the parent method. A node contains the target method name and the position of the call site in the parent method (e.g., a line number). Each node is categorized based on the optimizations it underwent (e.g., inlined, deleted). Log messages from the inliner and type profiles are collected as well.

Two inlining trees built by compiling the same method may have non-identical shapes. For example, code duplication [2] multiplies the number of 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 2.8a is an inlining tree obtained by parsing method `main` from the running example (Listing 3). Figure 2.8b 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 2.8c shows the differences between the trees in the form of a delta tree [31]. 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.

A shortcoming of comparing call sites by paths from the root is that we cannot distinguish call sites created by duplication. For example, after peeling a loop containing a call, we obtain a node representing the call site in the peeled iteration and another node representing the call inside the loop body. These calls might be optimized differently, but their target methods and paths from the root are equivalent. We cannot discern duplicated call sites because the compiler assigns the same source position to duplicated IR nodes. Note that extending Graal to create unique positions for duplicated code would be impractical.

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

2.4.3 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.

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`, as shown in Figure 2.9a. The second compilation unit does not perform a duplication. Instead, it inlines the call to `literalsEqual`, as shown in Figure 2.9b. Figure 2.9c compares the optimization-context trees of these compilation units.

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.

2.4.4 Tree Edit Distance (TED)

In this section, we apply TED [32] to semantically compare two optimization, inlining, or optimization-context trees. TED 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*.

Figure 2.10 shows two trees, T_1 and T_2, as an example. The nodes are labeled with letters. The figure shows a possible sequence of operations that transform T_1 into T_2. Each operation is associated with a cost. The total cost of the operations is $c_1 + c_2 + c_3$.

In general TED, the node deletion and insertion operations work on any tree nodes (including internal nodes). There are variants of TED [32] solving slightly different problems. In 1-degree TED [10], only subtrees may be inserted or deleted. In this section, we argue that the 1-degree variant matches our problem setting. We show this by enumerating the differences we want to report for each kind of tree.

Consider the potential semantic differences between two optimization trees. An optimization might be applied in one compilation but not the other. This manifests as a missing leaf node in one of the trees. An optimization phase may be missing in one of the trees. This situation can be expressed as a missing

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

(a) Optimization-context tree with duplication and without inlining.

```
(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
```

(b) Optimization-context tree without duplication but with inlining.

```
. (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
```

(c) Delta tree of two optimization-context trees.

Figure 2.9 Two optimization-context trees of method `main` from the running example (Listing 3) and their delta tree.

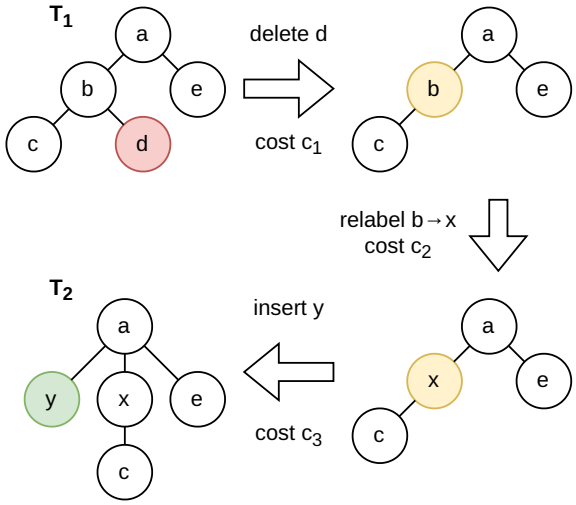

Figure 2.10 Two labeled ordered trees, T_1 and T_2, and the operations that transform T_1 into T_2.

subtree in one of the trees. See Figure 2.7 for an example. Additionally, we could consider detecting the situation when the phase plan is reordered. In the framework of TED, this could be interpreted as a subtree moving to a different tree node.

The potential differences between two inlining trees also include shape changes, i.e., call sites missing in one of the trees. Note that it is impossible for a call site to be missing from the tree while its callees are present. Thus, the missing call sites are always organized as subtrees. A fundamental semantic difference is different transformations applied to the same call site (tree node). Recall that this information is encoded as the category of the call site. Figure 2.8 showed an example of this.

The semantic differences between two optimization-context trees are inlining differences, optimization differences, and extra optimization decisions in code that is present in only one of the compilations. These situations can be captured in the framework of 1-degree TED as subtree deletions or insertions.

We use the ordered variant of TED rather than the unordered [32] because the order of phases matters. However, the inlining tree is unordered, and some phases perform order-independent optimizations. We solve this mismatch by sorting the inlining tree and the children of selected phases. This way, changes in order are not reported as differences.

1-degree TED does not model move operations [31]. Depending on the circumstances, a semantic move would be reported as an independent deletion and insertion. However, the upside of our approach is that insertions and deletions

```
0: identity a
1: insert y
1: relabel b to x
2: identity c
2: delete d
1: identity e
```

(a) Extended edit script.

```
. a
  + y
  * b to x
    . c
    - d
  . e
```

(b) Delta tree.

Figure 2.11 Extended edit script and a matching delta tree.

are easier to visualize than moves. Moreover, the algorithm [10] for 1-degree TED is fast and always finds an optimal solution.

The original algorithm proposed for 1-degree TED [10] finds the minimum cost of operations. The sequence of the operations, also called an *edit script*, can be computed by backtracking. Finally, the list of operations can be visualized as a delta tree [31]. The delta tree preserves the structure of both compared trees and shows their differences in context. We compute the delta tree by extending the original 1-degree TED [10] algorithm.

Figure 2.11 shows an example of a delta-tree construction. First, we compute an extended edit script. In contrast with the standard edit script [32], the extended edit script is ordered as a pre-order traversal of the delta tree. Moreover, the extended edit script contains the depth of each node in the delta tree. Profdiff reconstructs the delta tree from the extended edit script.

The delta tree includes unchanged nodes. Thus, the extended edit script also contains identity operations. An identity operation does not change a node; rather, it establishes a mapping between nodes. The depths are displayed as numbers before each operation in Figure 2.11a. Note that the depth of a delta-tree node equals the depth of the original node (or nodes) it represents. This is a property of 1-degree TED.

To compare two trees, we must define the cost function. Although the original problem statement is formulated in terms of labels, we use a slightly different formulation. Let us have two tree nodes, u and v. Then, we have a function $nodesEqual(u, v)$, which returns `true` if u and v are equal. Function $relabelCost(u, v)$ returns the cost of relabeling u to v if they are not equal. We always set the cost of inserting or deleting a leaf node to 1.

In the context of the optimization tree, our goal is to recognize matching optimization phases and decisions. Thus, $nodesEqual(u, v)$ returns `true` if u and v are equal nodes, i.e., they are either phases with the same identifier or optimization decisions with the same name, properties, and position. We do not have a use for the relabeling operation; therefore, $relabelCost(u, v)$ returns a large con-

stant. Note that it is always possible to transform T_1 to T_2 using only deletions and insertions.

In the context of the inlining tree, we say that two nodes represent the same call site if their names and paths from the root are equal. We can use the relabeling operation to highlight equivalent call sites with differing call-site categories. To this end, we define $nodesEqual(u, v)$ to be true if u and v are the same call site with equal call-site categories. $relabelCost(u, v)$ returns 1 if u and v represent the same call site with different categories. Otherwise, it returns a large constant.

For the optimization-context tree, the intent is to compare optimization decisions in each inlined method and inlining decisions. To this end, we merge the definitions of $nodesEqual(u, v)$ and $relabelCost(u, v)$ for the optimization and inlining trees. If u and v are different node types, e.g., an optimization decision and a call site, we return `false` and a large constant, respectively. Recall that the tree may also contain warning nodes. We consider a warning to be equal only to another warning node.

To improve the interpretability of the comparison, we preprocess the trees before they are compared. Several optimization phases perform low-impact transformations, such as local IR optimizations and dead-code elimination. These transformations are not driven by heuristics and are unlikely to change in a manner that would cause a regression. Therefore, by default, we remove these phases and their optimization decisions from the trees.

As already mentioned, we sort some of the optimization decisions in the optimization tree. We have a hand-picked list of phases whose applied optimizations we consider order independent. We sort the children of these phases in the optimization tree. The goal is to avoid mere order changes being reported as differences. The sorting criterion is based on source-level positions (i.e., line numbers in this thesis). In the optimization-context tree, we always sort all nodes.

Finally, we apply a postprocessing step to the delta tree. In the scenario where we want to focus only on the differences, profdiff iteratively removes all leaf nodes from the delta tree corresponding to identity operations. This way, we are left with only the differences in their contexts. If the input trees are equivalent, the post-processed delta tree is empty. This is convenient when we compare many pairs of trees but we are looking for differences.

2.5 Compilation Fragments

Inlining [3] 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 [33] 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 16 shows the dedicated compilation unit of method `literalsEqual`. The compiler peeled the method's loop once, as illustrated in Listing 4.

```
(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
```

Listing 16 Optimization-context tree for a dedicated compilation of method `literalsEqual`.

Now, consider the compilation unit of `main` shown in Listing 17. The compiler first duplicates the call to `literalsEqual`. Then, it inlines the call with the constant argument 0, where 0 limits the iterations of the inlined loop. The compiler removes the loop by peeling it and simplifying the peeled condition to the constant `false`.

We construct a *compilation fragment* from Listing 17 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 16 with the just-constructed compilation fragment.

The delta tree in Listing 18 compares the dedicated compilation unit with the compilation fragment. The tree highlights the replacement of the loop condition

```
(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
```

Listing 17 Optimization-context tree after duplication, inlining, and deleting the loop in `literalsEqual`.

with a constant. Thanks to the constant argument, the transformation was performed only in the fragment from Listing 17. We can see that all call sites inside the loop body were deleted in the compilation fragment.

The previous example introduced compilation fragments in terms of subtrees of the optimization-context tree. However, the concept is more general. We can also create fragments from optimization and inlining trees. The compilation fragment of an inlining tree is obtained by taking a subtree of the original inlining tree.

Creating a compilation fragment from an optimization tree is slightly more involved. Suppose that we want to create a compilation fragment for some method inlined in a compilation unit. After the compiler inlines the method, all subsequent optimization passes might potentially affect the code of the inlinee. Therefore, these optimization passes should be a part of the optimization tree of the fragment. Only the optimization decisions unrelated to the fragment must be filtered out. Given that the Graal compiler inlines methods [3] at the early stages of compilation, profdiff filters only the optimization decisions and preserves all optimization phases in the constructed optimization tree.

```
. (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
```

Listing 18 Delta tree of the dedicated compilation unit from Listing 16 and a fragment from Listing 17.

2.5.1 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 the most 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 in that method.

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. The global call tree may be infinite. However, this is not a problem because we do not need to build the tree. For simplicity, assume that our workloads are deterministic. Thus, the tree is invariant for all runs of the workload. Each node in the tree is linked to a concrete method. Several nodes may represent the same method called from different contexts. Figure 2.12 shows an example of a call tree. Nodes are marked with the concrete methods they represent (m_1 is the entry point).

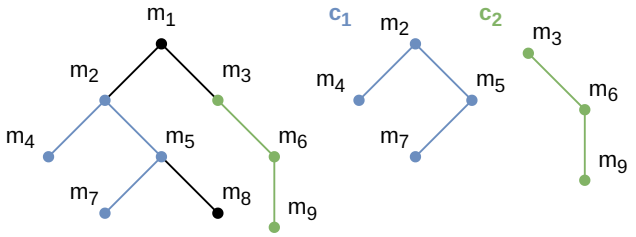

Figure 2.12 Example of a global call tree with entry point m_1. Compilation units c_1, c_2, and their covers are highlighted.

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. Figure 2.12 shows the call trees of two compilation units, c_1 and c_2. We say that a compilation unit *covers* some part of the global call tree. The parts covered by c_1 and c_2 are highlighted in Figure 2.12. 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 compiled the code of method m. Therefore, it is desirable to compare the optimization decisions in method m.

The situation is depicted at the top of Figure 2.13. There is a call tree for some workload shown twice. In the left copy of the tree, the nodes covered by c_1 are highlighted in blue. The nodes covered by c_2 are highlighted in green in the right-side copy of the tree. The nodes representing the methods m_1, m_2, and m are labeled in both tree copies.

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 2.13. 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.

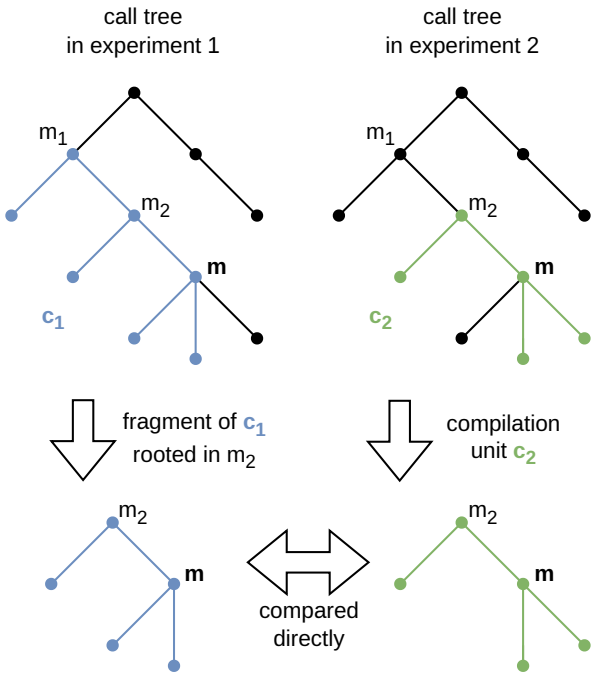

Figure 2.13 Method m and two compilations units, c_1 and c_2, that cover m. A compilation fragment is created from c_1 so that the optimization decisions in m are comparable.

Chapter 3

Case Studies

This chapter presents several case studies where we applied profdiff. The tool aims to help debug performance problems during compiler development. For this reason, the experiments were carried out using development builds of the compiler.

We focused on workloads from the Renaissance benchmark suite [27] 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 (e.g., 30) 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 [27] 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 iden-

tified 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.

3.1 Gauss Mix

The workload `gauss-mix` from Renaissance [27] 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%.

3.2 Scala K-Means

The workload `scala-kmeans` from Renaissance [27] 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 hot compilations of 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. We found that force-inlining a single particular method is 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. Thus, on JDK 17, we achieve a

total speed-up of about 8%.

3.3 Scala Doku

The workload `scala-doku` from Renaissance [27] 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%.

Chapter 4

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 4.1. Performance diagnosis in general [34] aims to analyze and fix already-discovered performance bugs. We talk about these methods in Section 4.2. 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 related to performance diagnosis in the context of a compiler.

Mosaner et al. [35] 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.

4.1 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 [36] is a test suite covering individual compiler optimizations. In random testing, small programs are randomly generated, compiled, and potential issues are detected. 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 [37] statically compares binaries by performance-related criteria such as instruction count, the number of arithmetic operations, or memory accesses. Kitaura and Ishiura [38] statically detect dissimilar code sec-

tions to detect potential performance differences. Then, they execute the code to verify the findings. Theodoridis, Rigger, and Su [39] 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 [40] 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, Grunwald, and Peri [41] 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, Liu, and Regehr [42] employ a white-box approach. They compute static analyses on code fragments and compare the results to the analyses computed by LLVM. This way, they uncover soundness issues or missed optimization opportunities.

4.2 Performance Diagnosis in General Software

We give an overview of tools that analyze performance problems in programs and potentially suggest fixes. These tools are aimed at application developers. In contrast, profdiff approaches performance diagnosis from a compiler engineer's point of view: it seeks to identify suboptimal optimization decisions.

Yu and Pradel [43] present an approach based on profiling to pinpoint root causes of synchronization bottlenecks in concurrent applications. Nistor et al. [44] introduce a method to detect loops that can be exited early, and they suggest possible source-code fixes. Curtsinger and Berger [45] show a method to evaluate the potential impact of speeding up particular lines of code in multithreaded applications. Song and Lu [46] present a tool to detect inefficient loops and suggest fix strategies. Della Toffola, Pradel, and Gross [47] present a tool that suggests memoization for Java methods that repeat the exact computations. Tan et al. [48] introduce a tool to mark useless memory operations. Wen, Liu, and Chabbi [49] present a technique to discover redundant computations.

Conclusion

In this thesis, we introduce the problem of identifying the causes of performance regressions in the context of dynamic compilers. We present a solution based on tracking optimization decisions. In our approach, the compiler captures inlining and optimization decisions as trees. The inlining tree represents the inlining decisions. The optimization decisions are organized in an optimization tree or an optimization-context tree. The former tree follows the dynamic structure of the optimizer, and the latter follows the structure of application code. Using a profiler, we determine which compilation units are hot. Profdiff compares the trees of hot compilations to find changes in optimization decisions that may be responsible for a performance difference. The thesis presents compilation fragments, which allow profdiff to compare differently inlined methods. The tooling is available as part of the open-source Graal compiler. We evaluate the techniques with industry-standard benchmarks and describe three instances where we pinpoint the exact decisions causing performance problems. By overriding these decisions, we achieve a speed-up of about 8% to 30%.

Future Work

The approach described in this thesis introduces new research opportunities to explore.

Reducing Costs of Performance Regression Detection

Performance regression detection is expensive in terms of machine time. For this reason, there are attempts to reduce the costs [7]. We could leverage profdiff as an indicator for commits that are unlikely to affect performance. Performance differences are less likely when optimization and inlining decisions do not change. Before the infrastructure initiates measurements for a particular workload, we could perform a single experiment and capture the profiles and optimization logs. Then, we may compare the experiment with a database of past experiments. If

there were no significant optimization differences in hot methods, the infrastructure could skip measuring the workload.

Classifying Workloads Using Optimizations

The compiler may apply diverse kinds and numbers of optimizations to different workloads in a benchmark suite. Thus, the list of applied optimizations characterizes the workload from the compiler's view. Based on the presence of particular optimizations in hot methods, we may try to encode a single workload as a real-valued vector. We could interpret the distance between such vectors as a measure of dissimilarity. This metric might help designers of benchmark suites evaluate the diversity of workloads in a suite.

Automatic Bug Discovery and Fixing

Performance problems like those presented in Chapter 3 could be detected and confirmed in an automated pipeline. For a workload with fluctuating performance, we could run several experiments, classify each as *fast* or *slow*, and find consistent optimization differences. As another step, we could override the identified decisions and test whether performance improves. If it does, the pipeline may report the offending optimization decisions and the possible speed-up to compiler developers.

References

[1] Gilles Duboscq et al. “Graal IR: An Extensible Declarative Intermediate Representation”. In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. Feb. 2013. URL: http://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf.

[2] David Leopoldseder et al. “Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 126–137. ISBN: 9781450356176. DOI: 10.1145/3168811.

[3] Aleksandar Prokopec et al. “An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 164–179. DOI: 10.1109/CGO.2019.8661171.

[4] Gilles Duboscq et al. “An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler”. In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL ’13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 1–10. ISBN: 9781450326018. DOI: 10.1145/2542142.2542143.

[5] Oracle. *Graal Compiler*. 2023. URL: https://www.graalvm.org/latest/reference-manual/java/compiler/. Accessed 2023-07-10.

[6] Lubomír Bulej et al. *Tracking Performance of Graal on Public Benchmarks*. Presentation at International Workshop on Load Testing and Benchmarking of Software Systems (LTB) 2021. 2021. DOI: 10.6084/m9.figshare.14447823.

[7] Milad Abdullah et al. “Reducing Experiment Costs in Automated Software Performance Regression Detection”. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 56–59. DOI: 10.1109/SEAA56994.2022.00017.

[8] *proftool*. 2023. URL: https://github.com/graalvm/mx/blob/master/README-proftool.md. Accessed 2023-07-10.

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

[10] Stanley M. Selkow. "The tree-to-tree editing problem". In: *Information Processing Letters* 6.6 (1977), pp. 184–186. ISSN: 0020-0190. DOI: 10.1016/0020-0190(77)90064-3.

[11] Oracle. *Java Platform Standard Edition 8 Documentation*. URL: https://docs.oracle.com/javase/8/docs/. Accessed 2023-07-10.

[12] Oracle. *GraalVM*. 2023. URL: https://www.graalvm.org/. Accessed 2023-07-10.

[13] Oracle. *Native Image*. 2023. URL: https://www.graalvm.org/latest/reference-manual/native-image/. Accessed 2023-07-10.

[14] David Leopoldseder. "Simulation-Based Code Duplication in a Dynamic Compiler". PhD thesis. 2019. URL: https://epub.jku.at/obvulihs/download/pdf/4410434.

[15] Lukas Stadler. "Partial Escape Analysis and Scalar Replacement for Java". PhD thesis. 2014. URL: https://ssw.jku.at/Research/Papers/Stadler14PhD/Thesis_Stadler_14.pdf.

[16] Igor Veresov. 2013. URL: https://www.slideshare.net/maddocig/tiered. Accessed 2023-07-10.

[17] Edd Barrett et al. "Virtual Machine Warmup Blows Hot and Cold". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133876.

[18] Sun Microsystems, Inc. *The Java HotSpot™ Performance Engine Architecture*. 2006. URL: https://www.oracle.com/java/technologies/whitepaper.html. Accessed 2023-07-10.

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

[20] John Rose. *MethodData*. 2013. URL: https://wiki.openjdk.org/display/HotSpot/MethodData. Accessed 2023-07-10.

[21] Thomas Kotzmann et al. "Design of the Java HotSpot™ Client Compiler for Java 6". In: *ACM Trans. Archit. Code Optim.* 5.1 (May 2008). ISSN: 1544-3566. DOI: 10.1145/1369396.1370017.

[22] Andy Georges, Lieven Eeckhout, and Dries Buytaert. “Java Performance Evaluation through Rigorous Replay Compilation”. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA ’08. Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 367–384. ISBN: 9781605582153. DOI: 10.1145/1449764.1449794.

[23] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. “Exploring Impact of Profile Data on Code Quality in the HotSpot JVM”. In: *ACM Transactions on Embedded Computing Systems* 19.6 (Oct. 2020). ISSN: 1539-9087. DOI: 10.1145/3391894.

[24] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. “Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler”. In: PPPJ ’14. Cracow, Poland: Association for Computing Machinery, 2014, pp. 187–193. ISBN: 9781450329262. DOI: 10.1145/2647508.2647521.

[25] James Gosling et al. *The Java® Language Specification*. 2023. URL: https://docs.oracle.com/javase/specs/jls/se20/html/index.html. Accessed 2023-07-10.

[26] Chris Siebenmann. *Understanding Resident Set Size and the RSS problem on modern Unixes)*. 2012. URL: https://utcc.utoronto.ca/~cks/space/blog/unix/UnderstandingRSS. Accessed 2023-07-10.

[27] Aleksandar Prokopec et al. “Renaissance: Benchmarking Suite for Parallel Applications on the JVM”. In: *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019, p. 17. DOI: 10.1145/3314221.3314637.

[28] Lukas Stadler et al. “An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance”. In: *Proceedings of the 4th Workshop on Scala*. SCALA ’13. Montpellier, France: Association for Computing Machinery, 2013. ISBN: 9781450320641. DOI: 10.1145/2489837.2489846.

[29] Urs Hölzle and David Ungar. “Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback”. In: *SIGPLAN Not.* 29.6 (June 1994), pp. 326–336. ISSN: 0362-1340. DOI: 10.1145/773473.178478.

[30] Sudarshan S. Chawathe et al. “Change Detection in Hierarchically Structured Information”. In: *SIGMOD Rec.* 25.2 (June 1996), pp. 493–504. ISSN: 0163-5808. DOI: 10.1145/235968.233366.

[31] Sudarshan S. Chawathe et al. “Change Detection in Hierarchically Structured Information”. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 493–504. ISBN: 0897917944. DOI: 10.1145/233269.233366.

[32] Philip Bille. “A survey on tree edit distance and related problems”. In: *Theoretical Computer Science* 337.1 (2005), pp. 217–239. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2004.12.030.

[33] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. “Partial Escape Analysis and Scalar Replacement for Java”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’14. Orlando, FL, USA: Association for Computing Machinery, 2018, pp. 165–174. ISBN: 9781450326704. DOI: 10.1145/2544137.2544157.

[34] Xue Han, Tingting Yu, and Gongjun Yan. “A systematic mapping study of software performance research”. In: *Software: Practice and Experience* 53.5 (2023), pp. 1249–1270. DOI: 10.1002/spe.3185.

[35] Raphael Mosaner et al. “Machine-Learning-Based Self-Optimizing Compiler Heuristics”. In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR ’22. Brussels, Belgium: Association for Computing Machinery, 2022, pp. 98–111. ISBN: 9781450396967. DOI: 10.1145/3546918.3546921.

[36] Nullstone Corporation. *NULLSTONE for Java*. 2012. URL: http://www.nullstone.com/htmls/ns-java.htm. Accessed 2023-07-10.

[37] Gergö Barany. “Finding Missed Compiler Optimizations by Differential Testing”. In: CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 82–92. ISBN: 9781450356442. DOI: 10.1145/3178372.3179521.

[38] Kota Kitaura and Nagisa Ishiura. “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*. A-TEST 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 38–44. ISBN: 9781450360531. DOI: 10.1145/3278186.3278192.

[39] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. “Finding Missed Optimizations through the Lens of Dead Code Elimination”. In: ASPLOS ’22. Lausanne, Switzerland: Association for

Computing Machinery, 2022, pp. 697–709. ISBN: 9781450392051. DOI: 10.1145/3503222.3507764.

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

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

[42] Jubi Taneja, Zhengyang Liu, and John Regehr. "Testing Static Analyses for Precision and Soundness". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 81–93. ISBN: 9781450370479. DOI: 10.1145/3368826.3377927.

[43] Tingting Yu and Michael Pradel. "Pinpointing and repairing performance bottlenecks in concurrent programs". In: *Empirical Software Engineering* 23.5 (Oct. 2018), pp. 3034–3071. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9578-1.

[44] Adrian Nistor et al. "Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 902–912. ISBN: 9781479919345.

[45] Charlie Curtsinger and Emery D. Berger. "Coz: Finding Code That Counts with Causal Profiling". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 184–197. ISBN: 9781450338349. DOI: 10.1145/2815400.2815409.

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

[47] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. "Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 607–622. ISSN: 0362-1340. DOI: 10.1145/2858965.2814290.

[48] Jialiang Tan et al. “What Every Scientific Programmer Should Know about Compiler Optimizations?” In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS ’20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392754.

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

Appendix A

Using Profdiff

Profdiff is available as part of the public compiler repository.[1] Graal and profdiff depend on the build tool mx.[2] We recommend building a Docker image with all the dependencies installed and running the examples in a one-off container. The profiler proftool depends on perf,[3] which may not work on non-Linux hosts. Use the command below to prepare a Docker image to run the examples.

```
docker build -t profdiff - << EOF
    FROM ubuntu:22.04
    RUN apt-get update && apt-get install -y build-essential \
        git python3-pip flex bison libelf-dev libdw-dev \
        libunwind-dev libtraceevent-dev libbfd-dev && \
        pip install ninja_syntax
    WORKDIR /source
    RUN git clone --depth 1 --branch profdiff \
            https://github.com/pecimuth/mx.git
    RUN git clone --depth 1 --branch profdiff \
            https://github.com/pecimuth/graal.git
    RUN git clone --depth 1 --branch v6.2 \
            https://github.com/torvalds/linux
    RUN ln -s /source/mx/mx /bin/mx && mx --quiet fetch-jdk && \
        ln -s ~/.mx/jdks/labsjdk-* labsjdk
    ENV JAVA_HOME=/source/labsjdk
    RUN mx -p mx build && mx -p graal/compiler build && \
        mx -p graal/vm --env ni-ce build && \
        make -C linux/tools/perf && \
        ln -s /source/linux/tools/perf/perf /bin/perf
    ENV EXTRA=-Dnative-image.benchmark.extra-image-build-argument
EOF
```

The requirements for the guest environment are git to clone the repositories,

[1] https://github.com/oracle/graal

[2] https://github.com/graalvm/mx

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

standard C development tools, pip to install a dependency of mx, and libraries to build perf. The build script clones Graal, mx, and Linux to the directory `/source`. Graal requires a JVMCI-enabled[4] JDK, which we download using `mx fetch-jdk`, and we set `JAVA_HOME` to the root of the JDK. We build the JIT compiler, Native Image, and mx itself using the command `mx build` with the parameter `-p` pointing to the root of the particular suite (project). The script also builds perf using make. Finally, the image sets the environment variable `EXTRA`, which we use in the AOT section to improve the readability of the commands.

Run the following command to start a one-off interactive container from the above image. The flag `--privileged` is required to run perf.

```
docker run -it --rm --privileged profdiff
```

All subsequent commands in Appendix A should be invoked inside the container. To verify that profdiff works, we can run unit tests using the command below.

```
mx -p graal/compiler unittest org.graalvm.profdiff
```

A.1 JIT Experiments

The command below runs a workload from the Renaissance [27] benchmark suite. Note that this is a simple use case without profiling; see how to enable profiling below. We use an option to track node source positions so that the compiler can associate optimization decisions with bytecode positions. We also instruct the compiler to save optimization logs (i.e., the inlining and optimization trees) to directory `opt_log`. Read the technical manual[5] to learn more about the optimization log.

```
mx -p graal/compiler benchmark renaissance:scrabble \
    --tracker none -- \
    -Dgraal.TrackNodeSourcePosition=true \
    -Dgraal.OptimizationLog=Directory \
    -Dgraal.OptimizationLogPath=$PWD/opt_log
```

Now, we run profdiff to display the collected logs. To learn more about the profdiff command, read Appendix A.3 or the technical manual.[6]

```
mx -p graal/compiler profdiff report opt_log | less
```

The output starts with a short explanation of the concepts and formats. Note that the output is long because it includes all Graal compilations. Profdiff groups

[4] `https://openjdk.org/jeps/243`

[5] `https://github.com/oracle/graal/blob/master/compiler/docs/OptimizationLog.md`

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

compilation units by the name of the root method. For each compilation unit, it shows an inlining tree and an optimization tree. We may use the following command to display optimization-context trees instead.

```
mx -p graal/compiler profdiff --optimization-context-tree true \
    report opt_log | less
```

A.1.1 JIT Experiment with Profiling

The command below executes a JIT workload with profiling. We enable proftool using `--profiler proftool`. It is not necessary to explicitly enable node source positions because proftool inserts the flags to enable them. Proftool saves the profiles to directories with the prefix `proftool_scrabble_`. For more details, refer to the proftool manual.[7] To pass these profiles to profdiff, we must convert them to JSON using `mx profjson`.

```
mx -p graal/compiler benchmark renaissance:scrabble \
    --tracker none -- --profiler proftool \
    -Dgraal.OptimizationLog=Directory \
    -Dgraal.OptimizationLogPath=$PWD/jit_log_1
mx profjson proftool_scrabble_* -o jit_prof_1.json
```

The command below reads the logs, marks the most frequently executed compilation units as hot, and displays their inlining and optimization trees.

```
mx -p graal/compiler profdiff report jit_log_1 jit_prof_1.json | less
```

We may use command-line options to configure which compilations are considered hot. For example, the command below marks the top 100 compilation units with the highest execution shares as hot. Read Appendix A.3 to learn about the available options.

```
mx -p graal/compiler profdiff \
    --hot-max-limit 100 --hot-percentile 1 \
    report jit_log_1 jit_prof_1.json | less
```

A.1.2 Comparing JIT Experiments

To compare two JIT experiments, we first rerun the same workload with profiling and convert the profiles.

[7] `https://github.com/graalvm/mx/blob/master/README-proftool.md`

```
rm -rf proftool_scrabble_*
mx -p graal/compiler benchmark renaissance:scrabble \
    --tracker none -- --profiler proftool \
    -Dgraal.OptimizationLog=Directory \
    -Dgraal.OptimizationLogPath=$PWD/jit_log_2
mx profjson proftool_scrabble_* -o jit_prof_2.json
```

We use the profdiff command `jit-vs-jit` to compare two JIT experiments. The tool marks hot compilations units and compares all pairs of hot compilations for each method.

```
mx -p graal/compiler profdiff jit-vs-jit \
    jit_log_1 jit_prof_1.json jit_log_2 jit_prof_2.json | less
```

By default, pairs of hot compilation units (or fragments) are compared using their inlining and optimization trees. We can use the command below to build and compare optimization-context trees instead.

```
mx -p graal/compiler profdiff \
    --optimization-context-tree true jit-vs-jit \
    jit_log_1 jit_prof_1.json jit_log_2 jit_prof_2.json | less
```

A.2 AOT Experiments

We can run a Native Image benchmark using the script below. In contrast to JIT compilation, Native Image must compile all reachable methods rather than just hot methods. Thus, there are often many more compilation units. The command below also enables the profiler proftool. The profiles help us identify hot compilation units so that we can focus only on hot code.

```
rm -rf proftool_scrabble_*
mx -p graal/vm --env ni-ce \
    benchmark renaissance-native-image:scrabble \
    --tracker none -- --profiler proftool \
    --jvm=native-image --jvm-config=default-ce \
    $EXTRA=-H:+TrackNodeSourcePosition \
    $EXTRA=-H:OptimizationLog=Directory \
    $EXTRA=-H:OptimizationLogPath=$PWD/aot_log_1
mx profjson proftool_scrabble_* -o aot_prof_1.json
```

We can view hot compilation units using the following command.

```
mx -p graal/compiler profdiff report \
    aot_log_1 aot_prof_1.json | less
```

A.2.1 Comparing JIT and AOT

Profdiff can compare the JIT experiment we ran previously with the AOT experiment.

```
mx -p graal/compiler profdiff jit-vs-aot \
    jit_log_1 jit_prof_1.json aot_log_1 aot_prof_1.json | less
```

A.2.2 Comparing AOT Experiments

Using the command below, we compile and rerun the same workload.

```
rm -rf proftool_scrabble_*
mx -p graal/vm --env ni-ce \
    benchmark renaissance-native-image:scrabble \
    --tracker none -- --profiler proftool \
    --jvm=native-image --jvm-config=default-ce \
    $EXTRA=-H:+TrackNodeSourcePosition \
    $EXTRA=-H:OptimizationLog=Directory \
    $EXTRA=-H:OptimizationLogPath=$PWD/aot_log_2
mx profjson proftool_scrabble_* -o aot_prof_2.json
```

Finally, we can use profdiff to compare two AOT experiments.

```
mx -p graal/compiler profdiff aot-vs-aot \
    aot_log_1 aot_prof_1.json aot_log_2 aot_prof_2.json | less
```

A.3 Command-Line Options

The syntax to invoke profdiff is `mx profdiff [OPTIONS] COMMAND`. We describe the available options below. Each command enables a particular use case. Run `mx profdiff help` to show the usage of profdiff or `mx profdiff help COMMAND` to show the help message for a particular command. The available commands are:

- `report` — view the logs of an experiment with an optional profile,
- `jit-vs-jit` — compare two profiled JIT experiments,
- `aot-vs-aot` — compare two profiled AOT experiments,
- `jit-vs-aot` — compare a JIT experiment with an AOT experiment.

All profdiff commands accept the same set of options. None are mandatory, and they are all set to default values suitable for quickly identifying differences. Run `mx profdiff help` to view the defaults. Note that most of these are binary options, which expect the literal `true` or `false`, e.g., `--long-bci=true`.

The algorithm that marks compilation units as hot is parametrized by the options `--hot-min-limit`, `--hot-max-limit`, and `--hot-percentile`. These options are relevant only when a profile from proftool is provided. `--hot-min-limit` and `--hot-max-limit` are hard upper and lower bounds, respectively, restricting the absolute number of compilation units marked as hot. `--hot-percentile` is the requested percentile of the execution period taken up by hot compilation units.

The option `--optimization-context-tree` enables the optimization-context tree. If enabled, optimization-context trees replace inlining and optimization trees in the output. This option is disabled by default. Read Section 2.3 to learn more about the optimization-context tree.

If `--diff-compilations` is enabled, profdiff compares all pairs of hot compilations (units or fragments) of the same method in two experiments. If disabled, profdiff prints all compilations without any comparison. This option is enabled by default. Read Section 2.4 to learn how profdiff compares compilation units.

If `--long-bci` is enabled, profdiff displays the complete position of each optimization. The option is disabled by default. We explained optimization positions in Section 2.1.1. If the option is disabled, profdiff formats the positions as single bytecode indexes relative to the root method of a compilation unit. If the option is enabled, profdiff prints complete positions, i.e., call stacks relative to the root method.

Depending on the value of `--sort-inlining-tree`, profdiff lexicographically sorts the children of each node in the inlining tree. The sorting criteria are the bytecode index of the call site and the method name. The option is enabled by default to avoid reporting differences caused by a different node order, as explained in Section 2.4.4.

If `--sort-unordered-phases` is enabled, profdiff sorts the children of selected optimization phases in the optimization tree. The goal is to establish a fixed order of optimization phases across compilation units to avoid reporting differences caused by a different order of optimizations, as explained in Section 2.4.4. This option is enabled by default.

Selected optimization phases are removed from the optimization tree when `--remove-detailed-phases` is enabled. These include the canonicalizer and dead-code elimination, as explained in Section 2.4.4. This option is enabled by default.

If `--prune-identities` is enabled, profdiff displays only the differences between two compared trees. In particular, profdiff removes leaf nodes representing identity operations from delta trees, as explained in Section 2.4.4. Note that a pruned delta tree is the empty tree when the compared trees are identical. This option is enabled by default.

The option `--create-fragments` controls the creation of compilation frag-

ments. This option is enabled by default. Read Section 2.5.1 to learn how profdiff creates compilation fragments.

If `--inliner-reasoning` is enabled, profdiff prints the reasons for all inlining decisions. This option is disabled by default. Read Section 2.2 to learn how we capture inlining decisions.