Optimization-Aware Compiler-Level Event Profiling

MATTEO BASSO, Università della Svizzera italiana (USI), Faculty of Informatics, Switzerland
ALEKSANDAR PROKOPEC, Oracle Labs, Switzerland
ANDREA ROSÀ and WALTER BINDER, Università della Svizzera italiana (USI), Faculty of Informatics, Switzerland

Tracking specific events in a program's execution, such as object allocation or lock acquisition, is at the heart of dynamic analysis. Despite the apparent simplicity of this task, quantifying these events is challenging due to the presence of compiler optimizations. Profiling perturbs the optimizations that the compiler would normally do—a profiled program usually behaves differently than the original one.

In this article, we propose a novel technique for quantifying compiler-internal events in the optimized code, reducing the profiling perturbation on compiler optimizations. Our technique achieves this by instrumenting the program from within the compiler, and by delaying the instrumentation until the point in the compilation pipeline after which no subsequent optimizations can remove the events. We propose two different implementation strategies of our technique based on path-profiling, and a modification to the standard path-profiling algorithm that facilitates the use of the proposed strategies in a modern **just-in-time (JIT)** compiler. We use our technique to analyze the behaviour of the optimizations in Graal, a state-of-the-art compiler for the Java Virtual Machine, identifying the reasons behind a performance improvement of a specific optimization, and the causes behind an unexpected slowdown of another. Finally, our evaluation results show that the two proposed implementations result in a significantly lower execution-time overhead w.r.t. a naive implementation.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Software testing and debugging*; **Dynamic analysis;**

Additional Key Words and Phrases: Dynamic analysis, profiling, compiler-IR instrumentation, just-in-time compilers, code optimization, debugging

ACM Reference format:
Matteo Basso, Aleksandar Prokopec, Andrea Rosà, and Walter Binder. 2023. Optimization-Aware Compiler-Level Event Profiling. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 10 (June 2023), 50 pages.
https://doi.org/10.1145/3591473

This work has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688).

Authors' addresses: M. Basso, A. Rosà, and W. Binder, Università della Svizzera italiana (USI), Faculty of Informatics, Via la Santa 1, CH-6962 Lugano-Viganello, Canton Ticino, Switzerland; emails: {matteo.basso, andrea.rosa, walter.binder}@usi.ch; A. Prokopec, Oracle Labs, Feldblumenstrasse 100, CH-8134 Adliswil, Canton Zürich, Switzerland; email: aleksandar.prokopec@oracle.com.

0164-0925/2023/06-ART10 $15.00
https://doi.org/10.1145/3591473

1 INTRODUCTION

To analyze effects of compiler optimizations, compiler developers typically instrument the source code or the intermediate language (e.g., bytecode), or rely on hardware performance counters to collect machine events. While source- and bytecode-level approaches allow profiling high-level events that correspond to object-oriented abstractions in the program (such as, e.g., object allocations or method invocations), machine-level profiling allows tracking low-level behaviour (such as, e.g., branch mispredictions or cache-misses). However, neither of these approaches allows collecting information about compiler-internal concepts (e.g., the lock implementation that the compiler selects [25, 27, 108], safepoints [23], or the typechecks that the compiler speculatively executes [29, 30]). They are too low-level to be visible in the source code or bytecode, and too high-level to be tracked in the machine code.

Moreover, it is established that source- and bytecode-level instrumentation leads to inaccurate measurements because compiler optimizations affect the event counts [125]. For example, source-level instrumentation may insert allocation-counting code to count allocations that can be removed by subsequent optimizations [110]. Conversely, the injected instrumentation code also perturbs the compiler's decisions: injected counters may prevent compiler optimizations by interfering with loop optimizations [114], by impacting the cost-analysis in inlining [124, 125], and in various other ways. While these problems are intrinsic to source- and bytecode-level instrumentation, compiler-IR-level instrumentation—i.e., instrumenting within the compiler's **internal representation (IR)**—may also perturb the compiler's decisions and lead to inaccurate results if not properly designed.

Aside from reducing the accuracy of the instrumentation,[1] interaction with the optimizations also introduces performance overheads. Even when the optimizations are not disturbed, instrumentation by definition adds more work into the program [31, 66, 102, 125], impacts memory locality and contention [60], and perturbs the system in a way that affects subsequent execution [69]. The overhead caused by instrumentation may be problematic for practical applications, especially in the common case where one needs to profile many different event types[2] at the same time.

To accurately track compiler-internal events without introducing high overheads that may impair the profiling of practical applications, we present a new instrumentation technique to efficiently and accurately profile 43 event types (Section 3). Our technique *inserts event markers* in the compiler's IR at the compiler phases *after which no subsequent optimizations affect the corresponding events*. These markers are no-ops from the compiler's viewpoint—they do not interact with the compiler optimizations, and are later replaced with the actual instrumentation code. Furthermore, we *delay the insertion of the instrumentation code* until the point at which no subsequent optimizations can occur. Similar to how source-level and bytecode-level instrumentation require understanding the program representation and the tools for modifying it [96, 97], the compiler-IR instrumentation that we propose requires understanding the internals of the compiler our approach is applied to. While user-facing instrumentation tools based on this technique must maintain the invariants of the compiler IR when modifying it, the users are only burdened with expressing the IR patterns that they want to profile.

[1] Accuracy can be defined as $1-(|observed\_count-actual\_count| \,/\, actual\_count)$; *observed_count* is the event count reported by our profiler and *actual_count* is the actual event count in an uninstrumented run. An event is counted in *actual_count* if a JIT-emitted machine-code instruction that corresponds to that event is executed. A perfectly accurate profiler (accuracy = 1) would produce an *observed_count* identical to the *actual_count*. Our technique aims at maximizing the accuracy.

[2] In this article, we use two terms associated with events: *type* and *occurrence*. We use the terminology *event type* (also just called *type* for short) to indicate an entity of interest that we want to profile, such as object allocation and method invocation. Instead, we use the term *event occurrence* (also just called *occurrence* for short) to indicate a single event occurrence of a given type.

We provide a generic API that enables users to customize the set of event types of interest (Section 4). Then, we apply our technique to implement event counting, i.e., the collection of runtime metrics that represent counts of specific execution steps in the program, such as, e.g., object allocations, atomic instructions, lock operations, typechecks, conditional jumps, direct and indirect method calls, foreign calls, memory barriers, arithmetic operations, safepoints, memory loads and stores, **garbage collection (GC)** card writes, deoptimizations, or any custom execution pattern that the user is interested in.

We propose two efficient path-profiling strategies aimed at reducing the profiling overhead when many event types of interest must be collected at the same time. Among several other profiling techniques (such as the one proposed by Ammons et al. [4]), we consider path-profiling particularly suitable for tracking compiler-internal events without introducing high overheads. The first strategy (called *path decoding*) allows one to collect the event counts online (i.e., during application execution), while the second strategy (called *path counting*) further reduces profiling overhead when event counts can be collected at the end of application execution. We implement the two proposed strategies in the open-source **just-in-time (JIT)** compiler Graal [28, 73, 91, 108] for the **Java Virtual Machine (JVM)** to profile 43 different event types at the same time.

We note that the standard path-profiling algorithm, performed at the end of the compilation pipeline where the IR size is bloated due to several optimizations, can produce an exponentially large number of paths, which drastically affects memory consumption and the runtime of the algorithm. To mitigate this issue, we propose a modification to the standard path-profiling algorithm that decreases the total number of paths and makes path profiling applicable in modern JIT compilers such as Graal without introducing prohibitively high memory consumption and compilation-time overhead.

In Section 5, we show that our approach allows compiler developers to analyze and debug the compiler optimization phases using IR metrics, which cannot be collected with methods that work at the source-, bytecode-, or machine-code level. In particular, we analyze the interaction between optimizations, event metrics, and the program execution time in Graal, identifying the reasons behind a performance speedup and the causes of an unexpected slowdown introduced by a compiler phase.

We evaluate our technique by comparing an implementation of our two strategies with a naive implementation without path profiling, called direct event counting (Section 6). In particular, we evaluate execution-time overhead, compilation-time overhead, code-size overhead, and memory consumption by profiling many event types on the Renaissance [93] and DaCapo [13] benchmark suites. Our evaluation results show that the proposed implementations allow one to profile a high number of events simultaneously with a significantly lower execution-time overhead w.r.t. direct path counting, at the cost of a higher memory consumption (and in the case of path decoding, also of a higher compilation time).

To summarize, our work makes the following contributions:

- We present a new instrumentation technique to efficiently and accurately profile the execution of compiler-internal program events (Section 3).
- We propose and implement two efficient path-profiling strategies and a modification to the standard path-profiling algorithm to count 43 different event types at the same time in the Graal JIT compiler for the JVM. Moreover, we provide a generic API that enables users to define their own event types (Section 4).
- We use our approach to identify the reasons behind a performance speedup and the causes of an unexpected slowdown introduced by a compiler phase (Section 5).
- We evaluate our two strategies and we show that they result in a significantly lower execution-time overhead w.r.t. a naive implementation (Section 6).

We complement the paper by presenting the necessary background information (Section 2), discussing the assumptions, other possible usages, and the limitations of our technique (Section 7), and comparing our approach with related work (Section 8). Finally, we give our concluding remarks in Section 9.

2 BACKGROUND

In this section, we present background knowledge necessary to understand the proposed technique and implementations. Section 2.1 introduces the terminology used in the article, while Section 2.2 provides an overview of the Graal compiler.

2.1 Terminology

Optimizing compilers can be broadly categorized as **ahead-of-time (AOT)**, also called static compilers, and **just-in-time (JIT)**, more broadly called dynamic compilers. Compilers apply a sequence of transformations and optimizations on a portion of code received as input, called *compilation unit*. Such code can consist of source code (in the case of an AOT compiler), or bytecode (typically in the case of a JIT compiler that compiles bytecode during application execution). Both source code and bytecode are impractical for most optimizations and for translation to machine code, so, before applying optimizations, optimizing compilers usually transform the compilation unit to a more suitable form, called ***intermediate representation (IR)***. We denote as ***IR language (IRL)*** the language that defines the IR, i.e., the set of instructions that an IR can consist of.

Compilers usually perform manipulations in successive *compiler phases* (henceforth also just called *phases* for short). Each phase takes an IR instance as its input, performs transformations by manipulating the IR, and returns the manipulated IR as its output. Specifically, we refer to phases whose main goal is to perform optimizations as *optimization phases*. Phases are performed one after the other to compose a *compilation pipeline*. The pipeline always starts with a phase that parses the source code (typically in the case of an AOT compiler) or bytecode (typically in the case of a JIT compiler) to IR and ends with a phase that transforms the IR to low-level IR or directly to machine code. Within the pipeline, the IRL does not remain the same, but can vary and progressively gets closer to machine code, by reducing to instructions whose abstraction level is lower. For example, an IR instruction that represents an array load can be transformed to another IR instruction indicating a memory read from a certain address. We call the process of transforming the IR by removing abstraction *lowering*, and the phases that perform lowering *lowering phases*.

2.2 Graal Compiler

Graal [29] is an open-source state-of-the-art compiler for the JVM implemented in Java. Even though in this article we use Graal as a JIT compiler, Graal can be used both as an AOT and as a JIT compiler. JVM implementations use Graal as a JIT compiler, and can load it as a plugin using the **JVM Compiler Interface (JVMCI)** [54]. In Graal, a compilation unit is a Java method scheduled for compilation along with all its callee methods that Graal decides to inline into it. Graal's IR is represented in a graph-based ***Static Single Assignment* (SSA)** [21] form where *nodes* represent either expressions or statements, and directed edges define either the control or data flow of the program [28, 29]. Graph nodes can be divided into two main categories: *fixed* and *floating*. Fixed nodes define the control flow of the program, i.e., the execution order of the different instructions. The predecessors of a fixed node must be always executed before the node itself. Hence, fixed nodes are used when ordering is crucial. Examples of fixed nodes include loops, method invocations, and lock acquisitions. Floating nodes fluctuate around the structure defined by fixed nodes, and are not bound to a specific point in the control flow, since their execution order is purely determined by data- and memory-ordering-dependencies. Floating nodes can represent, for example,

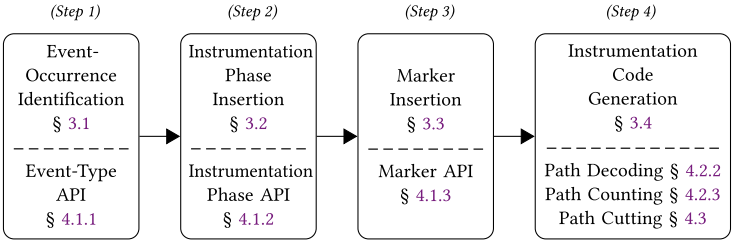


Fig. 1. Overview of the proposed technique for accurate profiling.

arithmetic operations or constants. This distinction allows Graal to perform optimizations such as constant folding [117], strength reduction, code motion [30], and global value numbering [18] efficiently without considering the exact position of the floating nodes until the very last compiler phases, when both fixed and floating nodes are *scheduled* (i.e., ordered and mapped to basic blocks) and translated into machine instructions. For more information, Duboscq et al. [28, 29] provide a detailed description of the Graal IR.

The compilation pipeline of Graal is divided into three main parts called *tiers*, namely the *high-tier* (initial tier), the *mid-tier*, and the *low-tier* (final tier). Each tier represents concepts at different levels of abstraction and uses a separate IRL. For example, the high-tier `LoadFieldNode` (specifying a load of an object field) can be translated to the mid-tier `ReadNode` (specifying a read of a memory address). While the high-tier applies high-level optimizations (such as method inlining), the mid-tier and low-tier apply lower-level optimizations (such as arithmetic-division simplification or null-check removal). A tier always concludes its execution with a lowering phase that transforms the current IRL to the one accepted by the next tier. At the end of the low-tier, the IR is translated to platform-independent low-level IR that does not consist of a graph-based SSA form anymore, but of basic blocks of instructions that point to each other. This low-level IR is later used for register allocation and subsequently machine-code generation for the architecture on which Graal is executed.

3 OPTIMIZATION-AWARE EVENT PROFILING

We now describe the proposed technique for accurate event profiling. Figure 1 shows an overview of our approach, which consists of four steps: identification of the event types of interest (Section 3.1), determination of the target instrumentation points in the compilation pipeline (Section 3.2), insertion of placeholder instructions that represent instrumentation (Section 3.3), and the generation of the instrumentation code (Section 3.4). For each step, Figure 1 reports the methodology section (above the dashed line) and the corresponding implementation sections (below the dashed line).

3.1 Event-Occurrence Identification

Our goal is to profile event types, i.e., to record patterns of interest in a program execution. Hence, the first step in our approach is to define the event-type set E. Since this technique targets compilers, the set E must be expressed in terms of the compiler's IR. Certain event types can be directly mapped to basic IR instructions, such as object-field loads or object allocations, while others must be identified with static analysis.

Definitions. We define the graph IR_k as the IR of a certain compilation unit after the k-th compiler phase in the compilation pipeline. Given the total number of compiler phases N, compiler phases are numbered from 1 to N and k is defined within the interval $[0, N]$. Value 0 does not

represent a compiler phase but indicates the beginning of the compilation pipeline, before the execution of the first compiler phase. An occurrence of the event type $e \in E$ can be statically identified by inspecting the patterns in IR_k. Each event type e can statically occur several times in IR_k. Hence, for each event type, it is possible to define a predicate that determines whether a certain instruction $\iota \in IR_k$ represents an occurrence of that type. Events that are represented by patterns, i.e., event types that are composed of multiple instructions, are identified by a single instruction for which the predicate returns true. We call this instruction the *representative instruction*. The predicate typically describes the pattern around ι.

$$predicate_e(\iota) \equiv \iota \text{ represents } e \tag{1}$$

We apply the predicate to each instruction in IR_k and we obtain a set $I_{k,e}$ that contains those instructions that correspond to e, i.e., the instructions that need to be instrumented:

$$I_{k,e} = \{\iota \in IR_k \mid predicate_e(\iota) \text{ is true}\} \tag{2}$$

We note that the same instruction $\iota \in IR_k$ may represent the occurrence of multiple different event types, i.e., given two event types $e' \in E$ and $e'' \in E$ such that $e' \neq e''$, it may be that both $predicate_{e'}(\iota)$ and $predicate_{e''}(\iota)$ are true. In this case, both sets $I_{k,e'}$ and $I_{k,e''}$ contain ι and subsequent steps of our approach instrument the same instruction ι twice to profile both e' and e''.

Example. Consider the `Character.valueOf` method from the **Java Class Library (JCL)**, which returns a heap-allocated object that corresponds to a primitive 16-bit Unicode character (i.e., an unsigned value that has a minimum value of 0 and a maximum one of 65535).

If the character value is less than 128, the object is loaded (load) from a pre-allocated array `arr`, otherwise a new object is allocated on the heap with new:

```
static Character valueOf(char i) {
  return i < 128 ? arr[(int)i] : new Character(i);
}
```

Figure 2 shows the Graal IR[3] for this compilation unit [28]. We define E so that it contains three event types. Two event types correspond to basic IR instructions—*load-array* and *allocation* correspond to load and new nodes, respectively. The third is a complex event type that we name *object-caching* (introduced in this example for illustration purposes), which consists of any if where one branch loads an object from an array using a primitive value as an index, and the other allocates an object using that same primitive value—to detect this, we must analyze the if and its surrounding expressions. We consider the merge corresponding to the analyzed if as the representative instruction of this event, hence, it is the only instruction to instrument and to include in the instruction set.

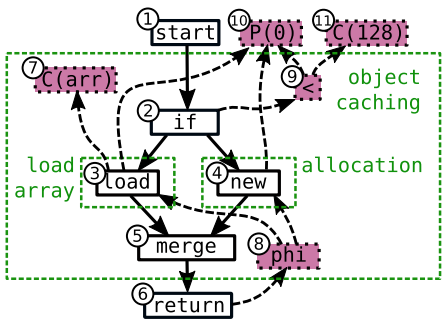


Fig. 2. Graal IR node for method `Character.valueOf`.

[3]We use the following conventions in every figure showing Graal IR. Fixed nodes are shown as white nodes with solid borders, while floating nodes are represented as purple (dark gray in grayscale printing) nodes with dotted borders. Solid arrows flowing from top to bottom represent the control flow, while dashed arrows flowing from bottom to top represent the data flow. We report constants with the notation `C(constant-value)` and parameters with the notation `P(parameter-index)`. To uniquely identify nodes, we assign a progressive ID to each of them, reported at the top-left corner of the node. First, we assign IDs to fixed nodes and then to floating nodes. Markers (Section 3.3.2) and instrumentation nodes (Section 3.4) are numbered after the nodes of the target application.

We can identify the sets $I_{k,e}$ as follows: $I_{k,\texttt{load-array}} = \{n_3\}$, $I_{k,\texttt{allocation}} = \{n_4\}$ and $I_{k,\texttt{object-caching}} = \{n_5\}$, where n_i refers to the node with number i in Figure 2. The value of k will be determined in the next section (Section 3.2).

3.2 Instrumentation-Phase Insertion

The detection of the event occurrences of each type e represented by the set $I_{k,e}$ is performed by a new compiler phase that we call *instrumentation phase* and denote as IP_e. This phase implements and applies the predicate function $predicate_e(\iota)$ to identify the instructions in $I_{k,e}$, and then inserts the marker nodes.

The crucial problem in the instrumentation-phase insertion is determining the phase k of the compilation pipeline after which IP_e should be added. Though the phase k depends on the use case (for example, a compiler developer might want to count the number of instructions after a certain phase, knowing that later optimization phases may remove some of these instructions), in this article, we focus on the common case where one wants to find a phase k that allows accurately quantifying the actual number of occurrences executed *at runtime*. We refer to such a phase with the symbol k_e and the term *target phase*. For example, given a compilation pipeline in which the number of compiler phases is $N \geq 5$, an instrumentation phase $IP_{\texttt{object-caching}}$, and $k_{\texttt{object-caching}} = 5$, $IP_{\texttt{object-caching}}$ would be inserted after the fifth compiler phase. We note that the position of the target phase depends on the event type of interest e; hence, different instrumentation phases (one for each event type) might be inserted in the compilation pipeline at different places.

In the following text, we first explain why finding k_e is crucial and why inserting IP_e at a phase that is not the target phase (as defined above) would not guarantee an accurate profiling for e, and also show an illustrative example. Then, we detail our approach to detect the target phase.

Motivation. Inserting the instrumentation phase IP_e at an arbitrary non-target phase k in the compilation pipeline can lead to two issues:

(1) If instrumentation is performed too early in the compilation pipeline, the profiler could instrument occurrences that are removed by subsequent compiler optimization phases. Similarly, the profiler might not instrument occurrences that are added by subsequent optimization phases. In both cases, the instrumentation would be incorrect: the profiled value would be higher or lower, respectively, than the actual number of occurrences at runtime.
(2) If instrumentation is performed too late in the compilation pipeline, an occurrence of interest might not be profiled anymore because the compiler has lowered such an occurrence by removing instructions that belong to a higher abstraction level. Hence, the profiler does not have enough information to accurately detect the pattern.

Example. We illustrate the above issues with an example (written in Java). We consider the *load-array* event type, i.e., an event type that represents a load of an element from an array, and a simplified compilation pipeline composed of six compiler phases (hence $N = 6$ and $k \in [0, 6]$). The compiler phases are reported below:

(1) *Bytecode parsing* parses the Java bytecode representation and creates the initial IR.
(2) *Inlining* [90] modifies the IR by replacing call sites with the bodies of the callees.
(3) *Double read elimination* replaces two consecutive loads to the same array element or object field with a single load, storing the result in a temporary variable to exploit caching.
(4) *Lowering* [108] transforms the IR to a lower representation by removing abstraction. For instance, array accesses are converted to raw memory accesses.
(5) *Dead code elimination* removes code whose execution does not affect program results.
(6) *Machine code emission* transforms the IR to platform-specific machine code.

```
1  long sumPositive(long[] numbers) {
2    long sum = 0;
3    int length = numbers.length;
4    for (int i = 0; i < length; i++) {
5      boolean isPositive = numbers[i] > 0;
6
7      if (isPositive) {
8        sum += numbers[i];
9      }
10
11   }
12   return sum;
13 }
```

Fig. 3. Method sumPositive, which adds up the positive numbers of the specified array and returns the sum.

```
1  long sumPositive(long[] numbers) {
2    long sum = 0;
3    int length = numbers.length;
4    for (int i = 0; i < length; i++) {
5      boolean isPositive = numbers[i] > 0;
6      eventCounters[loadArrayEvent] += 1;
7      if (isPositive) {
8        sum += numbers[i];
9        eventCounters[loadArrayEvent] += 1;
10     }
11   }
12   return sum;
13 }
```

Fig. 4. The sumPositive method (from Figure 3) after the execution of the $IP_{\texttt{load-array}}$ instrumentation phase that is inserted at $k < 3$.

Even though the compilation pipeline may seem too intricate for the current example, subsequent examples will refer to it, which motivates its complexity.

Let's assume that instrumentation code simply increments a counter associated to each event type (i.e., the profiler counts event occurrences over time). Counters are stored in the array `eventCounters` of element type `long` and are indexed using the constant `loadArrayEvent`, as shown below:

```
// instrumentation code of a load-array
eventCounters[loadArrayEvent] += 1;
```

We analyze the Java `sumPositive` method shown in Figure 3. This method takes a `long` array as a parameter and returns the sum of its positive elements.

In the following text, we show how the accuracy in profiling *load-array* varies by inserting the instrumentation phase $IP_{\texttt{load-array}}$ at every $k \in [0, 6]$. We focus in particular on the *load-array* occurrences profiled in method `sumPositive`, which represents our compilation unit. We execute the `sumPositive` method a single time by providing an array of length 10^9 as a parameter. This array contains $0.5 \cdot 10^9$ positive elements and $0.5 \cdot 10^9$ negative elements. As a result, the condition at line 7 evaluates to `true` and leads to the execution of line 8 in exactly 50% of the iterations.

Table 1 reports the corresponding number of profiled *load-array* occurrences for each k. Below, we illustrate the transformations performed by each compiler phase, and the effects that they have on $IP_{\texttt{load-array}}$ inserted at k:

- $k = 0$ (before the bytecode-parsing phase): the instrumentation phase is inserted at the beginning of the compilation pipeline. Hence, it instruments the array loads at lines 5 and 8 of Figure 3, producing the code in Figure 4. Since the `numbers` array provided as a parameter has length 10^9, the array access at line 5 is executed once for each loop execution (10^9 times) while the array access at line 8 is executed in 50% of the loop executions ($0.5 \cdot 10^9$ times). Therefore, the profiler tracks $1.5 \cdot 10^9$ *load-array* occurrences.
- $k = 1$ (after the bytecode parsing phase) and $k = 2$ (after the inlining phase): since the bytecode parsing phase simply changes representation without losing any abstraction, and the inlining phase does not inline any method calls, the instrumentation phase instruments the same *load-array* occurrences (as previously shown in Figure 4) and returns the same result as in $k = 0$.
- $k = 3$ (after the double read elimination phase): Figure 5 shows the Java code corresponding to the IR produced by the execution of both the double read elimination optimization and instrumentation phase. In particular, the compiler has replaced the two consecutive array loads with a single load whose result is stored in the local variable `element` (line 5 of Figure 5). The instrumentation phase detects and instruments only this *load-array* occurrence at line 5,

```
1  long sumPositive(long[] numbers) {
2    long sum = 0;
3    int length = numbers.length
4    for (int i = 0; i < length; i++) {
5      long element = numbers[i];
6      eventCounters[loadArrayEvent] += 1;
7      boolean isPositive = element > 0;
8      if (isPositive) {
9        sum += element;
10     }
11   }
12   return sum;
13 }
```

Fig. 5. The sumPositive method (Figure 3) after the execution of the $IP_{\texttt{load-array}}$ instrumentation phase inserted at $k = 3$.

Table 1. Occurrences of the *load-array* Event Type in the sumPositive Method (from Figure 3) for Different Instrumentation Phase Indices k

k	load-array
0	1 500 000 000
1	1 500 000 000
2	1 500 000 000
3	1 000 000 000
4	0
5	0
6	0

executed 10^9 times, by inserting the counter update at line 6. We note that this is the actual amount of runtime *load-array* occurrences, as array loads will not be further optimized.

- $k = 4$ (after the lowering phase): even though loads from arrays are still executed at runtime, the lowering phase removes the array-load abstraction, converting it to raw memory reads. The instrumentation phase is not able to detect *load-array* occurrences anymore. As a consequence, no instrumentation code is inserted and the profiler reports 0 *load-array* occurrences.
- $k = 5$ (after the dead-code-elimination phase) and $k = 6$ (after the machine code emission phase): due to lowering, *load-array* occurrences are not detectable by instrumentation phases. Hence, the number of reported occurrences of *load-array* remains 0.

We conclude that $k = 3$ is the only phase where $IP_{\texttt{load-array}}$ can be inserted to instrument the actual array loads occurring at runtime. Indeed, with $k < 3$, instrumentation would be inaccurate and with $k > 3$, *load-array* occurrences would not be detectable anymore. Even though we presented a single example that revolves around the *load-array* type, almost all interesting event types are similarly altered by the compilation pipeline. Hence, it is fundamental to determine the target value of k for each event type of interest.

Determining the Target Phase. As shown by the previous example, event occurrences must be detected and instrumented after they have been fully optimized by the compiler but before they are transformed into a low-level representation (and hence are not available anymore).

We insert the phase IP_e after a target phase k_e, defined as the last phase where the IRL allows identifying e and hence the last phase for which $I_{k,e}$ may be non-empty. As a consequence, k_e is either the last phase of the compilation pipeline or the phase that precedes the lowering of e (after which the transformed IRL does not allow the identification of e). Determining k_e thus requires some understanding of the compiler that the technique is applied to—the implementer must know which lowering phase lowers which event types.

We note that, depending on the event type and compilation pipeline of a specific compiler, there may be multiple compiler phases other than the target phase k_e (defined by our approach) that ensure an accurate profiling of an event type e. Given the target phase k_e, we can identify a compiler phase $y_e \leq k_e$ such that y_e is either the last phase preceding k_e that may optimize event occurrences of type e, or k_e itself if k_e optimizes event occurrences of type e. A compiler phase c allows accurate profiling of the event type e if $y_e \leq c \leq k_e$. Compiler phases not included in this range do not allow accurate profiling of e since phases preceding y_e would lead to inaccurate

profiles (the optimizations performed by y_e would not be considered in the profiles) and phases following k_e do not allow the identification of e anymore. Among the suitable phases, we consider the last phase as the target such that optimizations in the range $y_e \leq c < k_e$ do not need to employ resources to visit nodes corresponding to instrumentation code.

We also note our technique does not prevent us from accurately instrumenting an event type e whose occurrences may be "lowered" at two different positions in the compilation pipeline. In these cases, we simply separate the event type e into two disjoint event types e' and e'', and define their predicates $predicate_{e'}$ and $predicate_{e''}$ such that $k_{e'}$ and $k_{e''}$ can be different.

Consider for example an event type *raw-load* that represents a load of a value from a location specified as an offset relative to an object, without performing preliminary null checks. In the Graal's IR, a *raw-load* event type corresponds to a `RawLoadNode` that can be inserted into the IR both in the high-tier or in the mid-tier. In the high-tier, `RawLoadNodes` are inserted when parsing `Unsafe.get*` methods [67] while in the mid-tier, `RawLoadNodes` are inserted together with monitor acquisitions—a `RawLoadNodes` is used to fetch the mark word (i.e., a word containing information related to the status of the lock) associated with the object whose monitor must be acquired. Depending on where they are inserted, `RawLoadNode` are lowered at two different positions in the compilation pipeline, i.e., end of the high-tier or of the mid-tier, respectively. For this reason, it is not possible to determine a single target phase $k_{\texttt{raw-load}}$ that allows accurate profiling of *raw-load*.

To accurately profile *raw-load*, we define two disjoint event types *raw-load'* and *raw-load"*, their predicates $predicate_{\texttt{raw-load}'}$ and $predicate_{\texttt{raw-load}''}$, and the target phases $k_{\texttt{raw-load}'}$ and $k_{\texttt{raw-load}''}$, such that we can accurately capture every *raw-load* occurrence before it is lowered. In particular, *raw-load'* and *raw-load"* correspond to the *raw-load* occurrences before the high-tier and mid-tier lowering, respectively. The total number of *raw-load* occurrences can be computed by summing the occurrences of *raw-load'* and *raw-load"*.

3.3 Marker Insertion

The previous Section 3.2 details the identification of the target phase after which inserting IP_e. This instrumentation phase identifies and instruments the IR instructions that correspond to the event occurrences of type e. Even though IP_e could insert all the instrumentation instructions in the IR after the target phase, this approach would reduce the accuracy of the collected metrics.

In this section, we first illustrate why inserting instrumentation code at phase k would produce inaccurate measurements. We explain why and how instrumentation code can perturb compiler optimizations (Section 3.3.1), even if performed after the target phase k. Then, we detail how our methodology avoids the mentioned issues (Section 3.3.2).

3.3.1 Instrumentation Perturbs Compiler Optimizations. Generating and inserting the instrumentation code for quantifying the event occurrences of type e (instead of inserting markers) after the target phase k_e may perturb subsequent compiler optimizations, leading to inaccurate measurements of e itself as well as of other event types. A subsequent compiler optimization phase that processes IR perturbed by the instrumentation will in some cases not add or remove event occurrences in a way that reflects what was optimized in an uninstrumented run. As a result, the profiled event counts differ from what the event occurrences would be in an uninstrumented run. In the following text, we provide several motivations for why this behaviour is problematic, each accompanied by an illustrative example.

```
1  public void sumFirstTenPositiveLong() {
2    long[] numbers = new long[10];
3    for (int i = 1; i <= 10; i++) {
4      numbers[i-1] = i;
5    }
6
7    sumPositive(numbers);
8  }
```

Fig. 6. Method `sumFirstTenPositiveLong`, which calls the `sumPositive` method, previously defined in Figure 3.

```
1  public void sumFirstTenPositiveLong() {
2    long[] numbers = new long[10];
3    for (int i = 1; i <= 10; i++) {
4      numbers[i-1] = i;
5    }
6
7    long sum = 0;
8    int length = numbers.length;
9
10   for (int i = 0; i < length; i++) {
11
12
13     boolean isPositive = numbers[i] > 0;
14     if (isPositive) {
15       sum += numbers[i];
16     }
17   }
18 }
```

Fig. 7. The sumFirstTenPositiveLong method (Figure 6) after inlining the sumPositive method (Figure 3).

```
1  public void sumFirstTenPositiveLong() {
2    long[] numbers = new long[10];
3    for (int i = 1; i <= 10; i++) {
4      numbers[i-1] = i;
5    }
6
7    long sum = 0;
8    int length = numbers.length;
9
10   for (int i = 0; i < length; i++) {
11     long element = numbers[i];
12     eventCounters[loadArrayEvent] += 1;
13     boolean isPositive = element > 0;
14     if (isPositive) {
15       sum += element;
16     }
17   }
18 }
```

Fig. 8. The sumFirstTenPositiveLong method (Figure 7) after the $IP_{\texttt{load-array}}$ instrumentation phase inserted at $k = 3$. The instrumentation code is shown in red (gray in grayscale printing).

Motivation 1. Instrumentation code may contain side effects that alter subsequent optimizations, control flow, and instruction scheduling.

Example 1. We consider the *load-array* event type, the simplified compilation pipeline introduced in Section 3.2, and the sumFirstTenPositiveLong method in Figure 6. The method declares a long array (line 2) and initializes it with the first 10 long numbers, starting from 1 (lines 3–5). Then, the sumFirstTenPositiveLong method calls the sumPositive method defined in Figure 3, providing the previously declared variable numbers as a parameter and ignoring its return value (line 7). Essentially, this method invocation adds together the elements of the numbers array.

We summarize the effects of the simplified compilation pipeline on the sumFirstTenPositiveLong method when there is no instrumentation:

- Inlining: the inlining optimization inlines the call site at line 7 of Figure 6 producing the code conceptually shown in Figure 7. In Figure 7, notice that the sum variable is only declared (line 7) and incremented (line 15).
- Double read elimination: double read elimination optimizes the array accesses at lines 13 and 15 (Figure 7), as already explained in the previous example.

```
1  public void sumFirstTenPositiveLong() {
2    long[] numbers = new long[10];
3    for (int i = 1; i <= 10; i++) {
4      numbers[i-1] = i;
5    }
6
7    int length = numbers.length;
8    for (int i = 0; i < length; i++) {
9      eventCounters[loadArrayEvent] += 1;
10   }
11 }
```

Fig. 9. Instrumented sumFirstTenPositiveLong method (Figure 8) after the dead-code-elimination phase.

- Dead code elimination: since the sum variable is not further used after the second loop and the program does not perform any side effects, the whole body of the sumFirstTenPositiveLong method is considered dead code and it is removed.[4] As a result, the sumFirstTenPositiveLong method contains no code.

[4]We note that the compiler must perform several analyses to determine that the code is actually dead code that can be safely removed. For example, the compiler must prove that loops are bounded, that every array access is in-bound, and that the code does not have side-effects (including exceptions). Here, we simplify the explanation for the sake of exemplification.

We now consider the simplified compilation pipeline with $IP_{\texttt{load-array}}$ inserted at $k_{\texttt{load-array}} = 3$. The Java code that corresponds to the IR produced after the execution of $IP_{\texttt{load-array}}$ is shown in Figure 8.

The instrumentation code (shown in line 12) produces side effects and influences the subsequent dead-code-elimination phase that can safely remove only lines 7, 11, and 13–16 of Figure 8. The instrumented Java code that corresponds to the IR produced after the execution of dead-code-elimination phase is shown in Figure 9. The subsequently generated machine code contains more event occurrences, more instructions, and a different control flow w.r.t. the optimized machine code that would have been generated without the presence of instrumentation code.

We conclude that the instrumentation code inserted by $IP_{\texttt{load-array}}$ at $k = 3$ prevents the removal of dead code, thus perturbing the collection of *load-array* occurrences themselves. The profiler records 10 runtime *load-array* occurrences instead of 0. Even though in this example we considered a single *load-array* event type, we note that other event types are similarly affected. For example, while the correctly optimized code (i.e., the empty `sumFirstTenPositiveLong` method) contains no *allocation* occurrences, the instrumented code (Figure 9) contains one *allocation* occurrence (line 2).

Motivation 2. Instrumentation code increases the size of the IR, which reduces the code-size budget of the subsequent optimization phases that make decisions based on the IR size.[5]

Example 2. The inlining optimization uses the current code size of the compilation unit to determine whether to inline a particular callsite or not. If the code size has not reached a certain limit, a particular method call is inlined, which increases the current code size [5, 80, 90, 103, 118]. Once the limit is reached (which may be a function of the compilation unit hotness, the size of the specific callee, and other factors), the algorithm stops inlining method calls, to avoid generating compilation units whose size considerably increases the compilation time of the subsequent phases [90], or causes cache performance degradation [17] (and also to prevent endless inlining of recursive calls).

Instrumentation code can prevent inlining of some callsites, and in the worst case, of every callee, if the IR reaches the code-size limit. Consequently, the compiler not only emits more method calls, but also cannot perform subsequent optimizations that inlining would enable (this is usually the case for a compiler that mainly relies on intraprocedural analysis, like Graal [28], and most mainstream method-based JIT compilers [16, 47, 57, 70, 78]). As illustrated in the previous example, inlining of the `sumPositive` method (line 7 in Figure 6) is a prerequisite for removing the dead code. If the `sumPositive` method is not inlined, the compiler must assume that the method produces side effects and hence cannot remove the call site. The profiler in this case produces event counts that do not correspond to runs of the non-instrumented program.

To increase profiling accuracy, our technique avoids direct insertion of the instrumentation code. We next propose a strategy that solves the aforementioned problems.

3.3.2 Markers. In our methodology, while the instrumentation phase IP_e identifies the event occurrences of type e, it does not insert the instrumentation code to track the occurrence of e. Instead, it inserts a placeholder instruction that we call *marker*, and which denotes the execution of a certain event type. The marker is a special instruction in the IR that has no memory effects, nor other side-effects, and is not a data dependency—from a compiler's point-of-view, the markers are no-ops that do not influence compilation decisions.[6] Their code-size and cycle-count

[5]The code-size budget may be one among several budgets that the compiler uses to make optimization decisions. For example, a compiler may make decisions based on the execution time of the compilation unit. Currently, our technique does not prevent changes in execution-time driven optimization decisions.

[6]Instrumentation code that will later replace markers (Section 3.4) have side-effects. However, until that point in the compilation pipeline, markers are no-ops from a compiler's perspective.

estimates [62] are set to zero, to avoid perturbing budget-driven optimizations. Markers are converted to instrumentation code at the end of the compilation pipeline, and are removed from the IR before the final lowering step (Section 3.4).

Definitions. We define the set M as the set of marker types such that each event type $e \in E$ uniquely corresponds to a marker $m_e \in M$. The instrumentation phase IP_e is in charge of inserting markers of type m_e whenever an occurrence of e is encountered. Depending on e, the marker must either precede or follow the event occurrence. The first case applies for event types represented by instructions that split the control flow (i.e., instructions that have more than one successor, such as `if` and `switch`). Conversely, the second case applies to types that 1) may not be entirely and successfully executed because of potential runtime errors, or 2) are represented by instructions that denote the merging of multiple control-flow paths. In this way, we ensure that, in case of errors or exceptions, the corresponding event occurrence is not profiled; moreover, we insert a single marker in case of a control-flow split or merge.

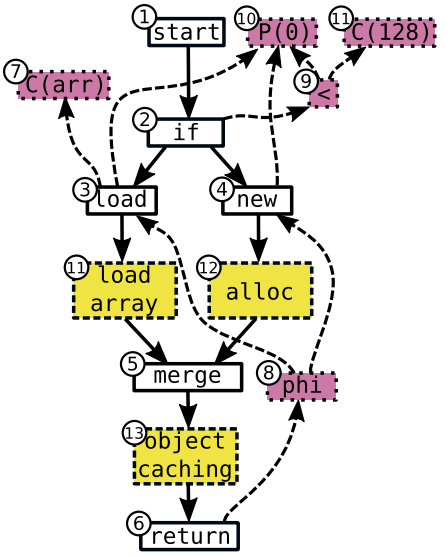


Fig. 10. Marker insertion in method `Character.valueOf`.

Since markers represent placeholder instructions, they do not belong to the original IRL and do not have a corresponding machine code representation. In this way, every subsequent compiler phase can "ignore" the marker nodes and avoid changes in behaviour. Markers must be replaced by instrumentation code defined on the original IRL before machine code emission, as explained in Section 3.4.

Example. Figure 10 shows the valueOf method from Section 3.1. The marker (shown in yellow with dashed border, light gray in grayscale printing) for the *object-caching* event occurrence must follow the corresponding merge instruction, since it might not be executed if it is placed in one of the branches. Instead, a marker for the *allocation* event type must be placed after the new instruction, to prevent the event occurrence from being counted in case of errors (for example, running out of memory). Similarly, a marker follows the load-array event occurrence to avoid the event-count increase if a null-pointer or an index-out-of-bounds exception occurs.

Discussion. Here we discuss two alternative designs to markers for the implementation of the proposed strategies and we compare them to our approach. As a first alternative design, instead of inserting markers to track event occurrences, instrumentation phases may insert IR instructions annotated as "instrumentation code" with code-size and cycle-count estimates set to zero (similarly to markers). Even though this approach may work correctly, it comes with caveats that require major changes in the implementation of the compiler. Every optimization phase should process and optimize annotated and not-annotated instructions separately to avoid changes in compilation decisions. Consider for example the case where instrumentation code uses memory-access instructions (read-add-write). Even if the costs for the heuristics were made zero, memory-access instructions still affect the memory graph and read-write reordering decisions in the compiler, whereas marker nodes do not.

As a second alternative design, instrumentation phases may mark event occurrences by annotating the corresponding program instructions. In this design, each program instruction is annotated

with an event-type set containing the event types the instruction represents. Event-type sets are copied when duplicating instructions, combined when combining instructions, transferred to the lowered instructions corresponding to the annotated instructions, and deleted together with instructions. Similar to markers, event-type sets are replaced by instrumentation code before machine code emission. This alternative design strongly couples annotations with instructions, possibly leading to better accuracy w.r.t. markers. The reason is that markers are independent of the lowered instructions that correspond to the event occurrences. Subsequent compiler phases may duplicate, move, and remove the lowered instructions but not the corresponding markers. Still, both when using markers and event-type set annotations, the instruction selection in the backend compiler may introduce inaccuracies due to combinations across block boundaries. We consider the event-type-set-annotation approach part of future work.

In Graal, where no existing API allows easily implementing event-type set annotations,[7] we consider high-level marker nodes a good compromise between accuracy and compiler-code maintainability.

3.4 Instrumentation-Code Generation

After all compiler and instrumentation phases are performed—i.e., after the $(N + M)$-th phase, where M is the number of instrumentation phases injected in the compilation pipeline—our technique detects and replaces the markers, producing an IR in the original IRL. We refer to this process as *instrumentation-code generation*. We define a new instrumentation-code-generation phase *ICG*, which is injected into the compilation pipeline after phase $N+M$. The phase *ICG* processes the markers and implements a concrete scheme for *recording* event occurrences.

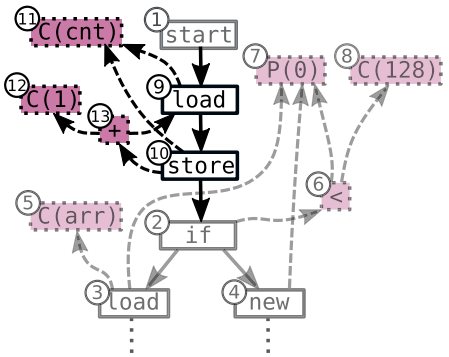


Fig. 11. Instrumentation-code generation in method `Character.valueOf`.

Example. If our technique is used to count the occurrences of each event type, the phase generates code that increments the counters specific to each event type e. On the other hand, if our technique is used to produce an event-trace (i.e., an ordered list of executed event occurrences), then the phase generates code that, for each e, appends a representation of the occurrence of e into a trace-buffer. The marker insertion allows performing several optimizations during this phase, as explained later in Section 4.2. In Figure 11, the *object-caching* marker (Figure 10, node 13) is replaced with the incrementation of a counter at the address cnt (in the IR, the instrumentation code is shown with darker colors, while code of the target application is semi-transparent).

4 IMPLEMENTATION

This section describes the implementation of our technique in the Graal compiler [28, 29, 62, 73, 91, 109, 120]. As shown in Figure 1, Section 4.1 describes our event-definition, instrumentation-phase, and marker APIs. Then, Section 4.2 presents our approach to generating instrumentation code, and proposes two efficient path-profiling strategies to reduce the profiling overhead. Finally,

[7] In Graal, nodes are instantiated throughout the codebase using their constructor. To support event-type set annotations, we would need to analyze all (and refactor some of) the allocation sites to use another constructor that takes the event-type set annotation as a parameter. Similar changes would need to be implemented in Graal's low-level IR.

Section 4.3 describes a modification of the standard path-profiling algorithm [9] that makes the two proposed strategies applicable to modern JIT compilers.

4.1 Events, Markers, and Instrumentation Phase

In this section, we illustrate our event-definition API (Section 4.1.1), instrumentation-phase API (Section 4.1.2), and marker API (Section 4.1.3).

4.1.1 Event-Type API. In our implementation, event types are represented as Java classes. In particular, each $e \in E$ must implement the `EventType` interface defined by the simplified Java code shown in Figure 12. The `name` method defined at line 2 returns a unique intuitive name represented as a Java `String` that describes the event type. The $predicate_e(\iota)$ function is implemented using the `eventAt` method defined at line 3 where ι is represented as an IR node. Recall that the instrumentation markers may be placed before or after the target node (depending on e). Therefore, in contrast to $predicate_e(\iota)$, the `eventAt` method does not return a boolean, but the IR node after which the instrumentation marker must be inserted. If the node provided as a parameter does not represent an event occurrence, then eventAt returns null. Since k_e depends only on e, k_e is defined within the event type itself as the combination of the two methods `tier` and `phase`, which represent the position of the phase in the Graal compiler, declared at lines 4 and 5, respectively.

```
1  interface EventType {
2    String name();
3    Node eventAt(Node node);
4    Class<Tier> tier();
5    Class<Phase> phase();
6  }
```

Fig. 12. Interface for event type definition.

After all the event types are defined, we add them to the set E represented by a Java `Set<EventType>`.

4.1.2 Instrumentation Phase API. As shown in Figure 13, an implementation of the class InstrumentationPhase (which allows defining instrumentation phases) must provide two methods: eventType (which returns the event type that the phase is supposed to mark) and mark (which locates the position and inserts the markers in the IR Graph provided as a parameter). The default implementation of the mark method computes the schedule of floating instructions (i.e., an assignment of floating nodes to fixed-node positions [18]), invokes eventAt for each node in the IR, and then inserts the markers in graph. For more complex pre-marking analyses the users can implement their own marking logic by overriding mark.

```
1  abstract class InstrumentationPhase
2  extends Phase {
3    abstract EventType eventType();
4    void mark(Graph graph) { /* ... */ }
5  }
```

Fig. 13. Class for instrumentation phase definition.

The instantiation and insertion of the different instrumentation phases is performed during the setup of the Graal compiler. We iterate over the user-provided list of instrumentation phases and we call the `tier` and `phase` methods (cft. Figure 12) of the corresponding eventType to determine the insertion position in the compilation pipeline.

```
1  class EventMarkerNode
2  extends FixedWithNextNode {
3    final EventType type;
4  }
```

Fig. 14. Class that represents marker nodes in the Graal IR.

4.1.3 Marker API. Markers are represented by a new single IR node called `EventMarkerNode`, shown in Figure 14. The event type of the occurrence is identified by the type field (line 3).

Since markers must be inserted at a specific point of the control flow of the program, `EventMarkerNode` extends the `FixedWithNextNode` class, i.e., the class used by Graal to define fixed nodes that have successors. If the marked event type is represented by a fixed node, then the marker is inserted after the node that eventAt returns. If the marked event type is represented by a floating node, then the

instrumentation phase must find the fixed node at which the floating node returned by eventAt will be positioned before code generation—for this reason, instrumentation phases run Graal's scheduling algorithm to produce the *schedule* (i.e., the ordering) for floating instructions [18] (however, to decrease the compile-time overhead, the phases that do not work with floating nodes can disable the computation of the schedule).[8]

4.2 Instrumentation-Code Generation

At the end of Graal's compilation pipeline, we insert an additional phase that generates instrumentation code, i.e., a phase that converts marker nodes to code that records event occurrences. In this article, we focus on event counting, hence "recording" in our case means incrementing a thread-safe counter. As an alternative example, in implementations that aim to produce program traces, recording would mean appending information to a trace buffer.

In the examples in this section, we consider the following expression, which represents the creation of a wrapper around a boxed character in Java:

```
Optional.of(Character.valueOf(x))
```

Since the code size of the `Character.valueOf` method is small, we assume that in this example, `Character.valueOf` is inlined into the `Optional.of` method [5, 80, 90, 123].

We record an *if* event type e_0 that represents the execution of `if` statements, and the *allocation* event type e_1 introduced in Section 3.2. Figure 15 shows the IR of this expression, where *allocation* occurrences are represented by the `new` nodes (numbers 4 and 6) and *if* occurrences are represented by if nodes (number 2). The yellow (light gray in grayscale printing) marker node 13 is associated to the if node 2. Nodes 14 and 15 represent markers inserted by the instrumentation phase and associated to new nodes 4 and 6, respectively.

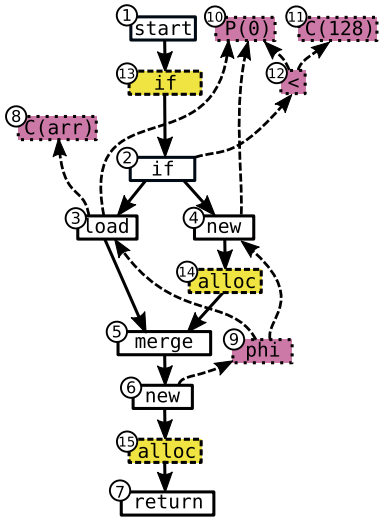


Fig. 15. Graal IR for the expression `Optional.of(Character.valueOf(x))`.

In what follows, we describe three different strategies for inserting the instrumentation code, i.e., replacing the markers in Figure 15 with nodes that represent instructions that perform the instrumentation. First, we introduce a simple naive approach that will be used for comparison purposes (Section 4.2.1). Then, we propose two alternative strategies to reduce the execution overhead, based on path profiling [9] (Sections 4.2.2 and 4.2.3).

4.2.1 Direct Event Counting. A simple approach of implementing instrumentation-code generation is to replace each marker node with an increment at the memory location of the event-type counter. We refer to this strategy with the term *direct event counting*, and we show the corresponding IR in Figure 16, where IR nodes of the original expression are reported with a lighter color than the instrumentation code. Counters are stored into the `cnt` array (nodes 15, 21, and 27) in which

[8] We always instrument the occurrence of a node before it is lowered. For floating nodes, their lowering includes creating a schedule and placing them into basic blocks before converting them to a lower-level representation. This implies that our scheduling result corresponds directly to the scheduling that is immediately done by the lowering phase that follows the insertion of the markers.

the i-th element `cnt[i]` represents the counter associated to e_i. For example, `cnt[0]` is associated to the *if* event type, and `cnt[1]` is associated to the *allocation* event type. As the IR contains three different occurrences, the figure shows three different counter updates framed in light blue boxes. Each counter update uses a `load` node (number 13, 19, and 25), the `cnt` array (nodes 15, 21, and 27), and the constant index associated to the event type (nodes 18, 24, and 30) to obtain the current value of the counter. This value is then used by a + node (number 17, 23, and 29) to compute the new incremented value by adding the constant 1 (node number 16, 22, and 28). A `store` node (number 14, 20, and 26) updates the counter by storing the incremented value into memory.

Complexity. The benefit of this approach is that the memory overhead is low: we allocate a memory region (i.e., the `cnt` array) for the counters, one for each event type, hence the memory consumption is $O(|E|)$, where E is the set of event types. However, the execution time overhead is $O(R)$, where R is the number of runtime event occurrences. Hence, this approach may be convenient when the event-marker frequency is low, or when the event's computational cost significantly outweighs the cost of the counter update. We note that in this article we focus on the common case of many event types and highly frequent event markers where counter updates are expensive (as we show in Section 6). For this reason, Sections 4.2.2 and 4.2.3 propose two alternative strategies to overcome this problem.

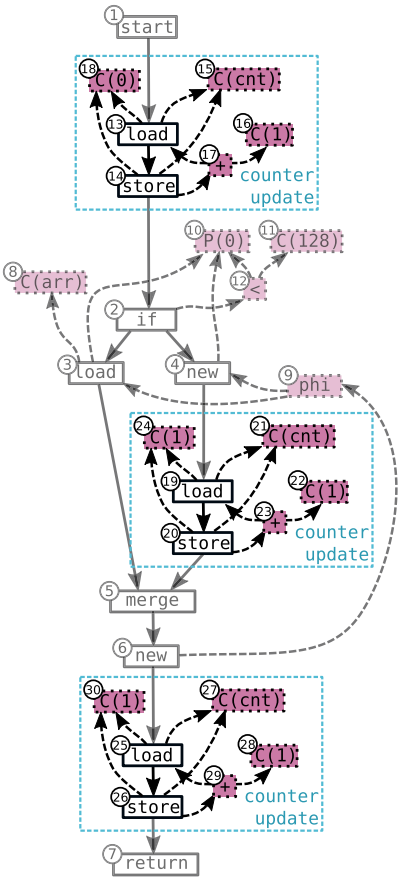


Fig. 16. Direct event-counting IR for `Optional.of(Character.valueOf(x))`.

4.2.2 Path Decoding. To reduce the overhead of accessing memory at every event-marker location, we can exploit path profiling [9]. Path profiling separates the control-flow within the compilation unit into a set of paths. Then, instead of incrementing the event count at every marker, the event count is updated at the end of the path. In this way, if some occurrences are repeated along the path, we pay the cost of the memory update only once. We call this strategy *path decoding*.

The IR of the `Optional.of(Character.valueOf(x))` expression (shown in Figure 15) has two paths in total, we call them P_0 and P_1. The path P_0 includes the first branch of the `if` statements and consists of the fixed nodes <1, 13, 2, 3, 5, 6, 15, 7>.[9] P_1 includes the second branch of the `if` statements and hence fixed nodes <1, 13, 2, 4, 14, 5, 6, 15, 7>. Along P_0, there is one *if* occurrence and one *allocation* occurrence, while along P_1 there is one *if* occurrence and two *allocation* occurrences. Since the *allocation* type repeats along P_1, it is more efficient to increment the count by two units only once at the end of the path instead of incrementing it by one unit twice. We note that in practice (unlike in the example, which is simplified for clarity) most event types often occur

[9]To indicate a path, we use the ordered list of the node IDs composing the path, enclosed in angle brackets <·>.

several times along the same path, making the benefits of this strategy in real-world workloads more significant (the differences are quantified in Section 6).

We keep track of the occurrences of each event type in each path with a *path-decoding table* p_{ec}, in which the j-th row represents the j-th path P_j and the i-th column represents the i-th event e_i. We implement the decoding table as a contiguous memory area that stores rows of length $|E|$ one after another. The number of occurrences of the event e_i along the path P_j are stored in the element at index $j \cdot |E| + i$. For instance, given the current IR, $p_{ec} = [1, 1, 1, 2]$, since entries 0 and 1 represent the first row and hence refer to P_0, entries 2 and 3 represent the second row and refer to P_1: even indices refer to the *if* event type (first column), and odd indices refer to the *allocation* event type (second column). As an example, the number of *allocation* occurrences (e_1) along P_1 is $p_{ec}[j \cdot |E| + i] = p_{ec}[1 \cdot 2 + 1] = p_{ec}[3] = 2$. During the program execution, the JIT compiler incrementally extends the decoding table for each compilation request, so that it contains information about the newly added paths. During instrumentation-code generation, and after the control flow is separated into a set of paths, the compiler populates the table p_{ec} by statically counting markers along those paths. For a particular compilation unit, the contents of the table are never updated but always queried to retrieve intermediate results.

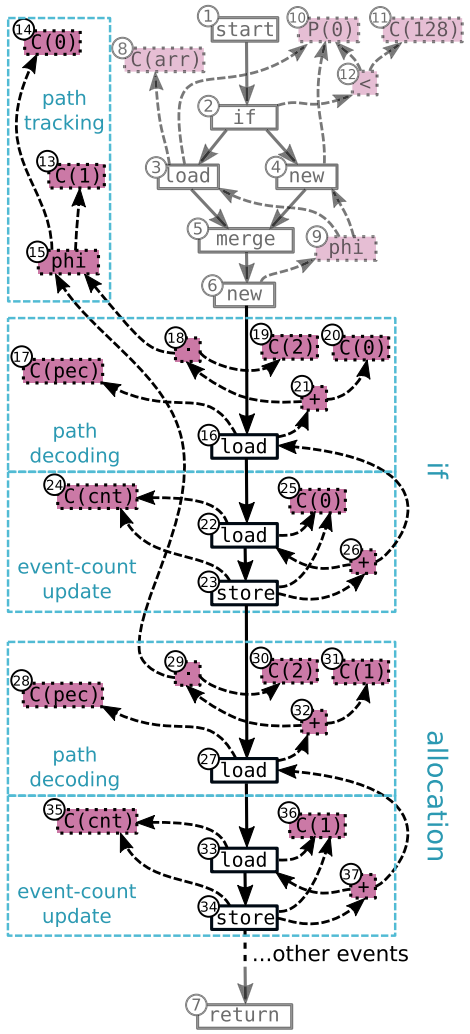


Fig. 17. IR with path decoding for `Optional.of(Character.valueOf(x))`.

Figure 17 shows the IR after the execution of an instrumentation-code-generation phase that implements the path-decoding strategy. While event counts are still stored into the `cnt` array (nodes 24 and 35), the instrumentation code is divided into three steps reported in the light blue boxes. These steps are identified by the corresponding labels: *path tracking*, which maintains the ID of the current path that is being executed, *path decoding*, which extracts the number of occurrences along that path, and *event-count update*, which increments the event count by that number. As the first step, this strategy enumerates all the paths, and inserts a *path identifier* along each path in the control flow graph using the approach proposed by Ball and Larus [9]. In the *path tracking* block, the phi[10] node 15 represents the path identifier, and evaluates to either 0 or 1 depending on the path (nodes 14 and 13, respectively).

[10]Graal's phi nodes represent the ϕ-function of the SSA form, i.e., a function that given N branches with the same join point, each associated to a value, returns the value associated to the branch that has been taken in the current execution. Unlike the original description by Ball and Larus [9], we use SSA values to represent the current value of the path ID.

Then, this strategy inserts one *path-decoding* block and *event-count update* block for each event type in the set E where at least one path ends. Multiple paths ending at the same instruction exploit the same *path-decoding* and *event-count update* block by using a different path identifier. In the figure, the *path-decoding* and *event-count update* blocks for one event type refer to both P_0 and P_1, since they end at the same instruction (node 7). The IR first updates the counter associated with the *if* event type, and then the counter associated to the *allocation* event type. Consider the *allocation* event type: the path identifier produced by node 15 is used in the *path-decoding* part to compute the index $j \cdot |E| + i$, and load the precomputed allocation-event count from the path-decoding table p_{ec} (nodes 27-32). In particular, j is the result of node 15, $|E|$ is represented by node 30 (constant 2), and i by node 31 (constant 1). Finally, the *event-count update* block uses the decoded event count for the respective path (node 27) to increment the existing counter in the `cnt` array. To do so, we generate IR similar to the one produced in the *direct event counting* strategy.

Even though this strategy adds more nodes than direct event counting (i.e., the *path-tracking* and *path-decoding* blocks), we note that the benefits in terms of profiling overhead are substantial when many event types are profiled, as we show in Section 6.2.

Complexity. Let, across all compilation units, P_e be the set of paths in which event type e appears, and let $P_E = \bigcup_{e \in E} P_e$ be the set of all paths in which at least one event occurrence appears. During compilation, path decoding must store up to $|E|$ values into the decoding table for each path (to track the respective count of each event type in that path). The total memory consumption is therefore $O(|E| \cdot |P_E|)$.

One advantage of path decoding is that the event counts are maintained in real-time. This makes it applicable to JIT compilers and VMs, because statistics can be gathered while the program is executing—for example, the GC can use allocation-count statistics to dynamically scale generations [15], or the JIT compiler can deoptimize the code to select a better lock implementation [26]. The disadvantage of this approach is that the overhead grows with the size of E—the *path-decoding* and *event-count update* blocks are injected once for each event type that occurs in any of the paths that lead to the instrumentation point. Thus, the execution-time overhead is $O(|E| \cdot \pi^{-1})$ in the worst case, where E is the event-type set, and π is the average path length.

4.2.3 Path Counting. When real-time updates are not a requirement, the execution-time overhead of the path decoding strategy can be lowered by counting only the number of times that a path was executed, instead of updating the actual event counts. The event counts can then be decoded using the decoding table p_{ec} offline. We call this strategy *path counting*. In our implementation, we install a VM-shutdown hook that performs bulk event decoding.

An example of path counting is shown in Figure 18. Since this strategy does not update event counts, we remove the `cnt` array and we introduce a p_c array (node 18) to store the dynamic path-execution counts, instead. The j-th element $p_c[j]$ represents the counter associated with the path P_j (where j is a global index across all compilation units). For example, $p_c[0]$ is associated with P_0, and $p_c[1]$ is associated with P_1. While the *path-tracking* part remains unchanged w.r.t. the one reported in Figure 17, all *path-decoding* and *event-count update* blocks are replaced with a single *path-count update* block (nodes 16-20). The path identifier j returned by path tracking (previously used to compute the index to access the path-decoding table) is now used to access $p_c[j]$. The path-count update block just increments $p_c[j]$ by one unit, to signal the fact that the path was executed once. We remark that p_{ec}, which contains the number of static occurrences of each event type for each path, is computed via static analysis at compile time and never accessed at runtime. For this reason, the address of the decoding table p_{ec} does not appear in the IR.

Via offline analysis, we can compute the number of event occurrences in each path, and therefore obtain the global event counts. To retrieve the global event count of an event type e_i, we sum the

product of $p_c[j]$ multiplied with the number of occurrences of that event type in the j-th path (obtained by accessing the decoding table p_{ec} as it was accessed in the path-decoding strategy) for each path j, that is: $count(e_i) = \sum_{j \in [0, |P_E|)} p_c[j] \cdot p_{ec}[j \cdot |E| + i]$.

Complexity. The memory consumption of this approach is $O(|E| \cdot |P_E|)$, since the path-decoding table p_{ec}, which contains the number of static occurrences of each event type for each path, has length $|E| \cdot |P_E|$, while the length of array p_c is equal to the total number of paths $|P_E|$. However, we note that the decoding table could be stored in secondary storage, and accessed only during the shutdown phase of the program to do the decoding, which decreases the main-memory requirements to $O(|P_E|)$. Our implementation keeps the path-decoding table in main memory, because none of the benchmarks depleted the memory that was preallocated for path counting.

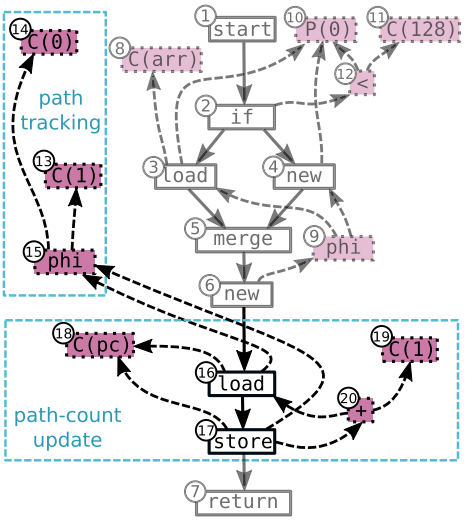


Fig. 18. IR with path counting for `Optional.of(Character.valueOf(x))`.

The execution-time overhead is reduced from $O(|E| \cdot \pi^{-1})$ to $O(\pi^{-1})$, where π is the average path length. In contrast to the path-decoding strategy, at the end of each path, the instrumentation code performs a single path-counter update and does not separately update each event type that occurs in any of the paths that lead to the instrumentation point. Hence, the execution-time overhead depends only on the average path length π.

4.3 Path-Cutting Optimization

In this section, we first motivate why the standard path-profiling algorithm [9] may lead to an intractable overhead in a modern optimizing compiler such as Graal, making it impossible to directly apply the original algorithm to implementing the path-decoding (Section 4.2.2) and path-counting (Section 4.2.3) strategies. Then, we propose a modification to the path-profiling algorithm that makes the proposed strategies usable in Graal.

Motivation. The main downside of path-decoding and path-counting strategies is that path profiling can produce an exponentially large number of paths $|P_E|$, which drastically affects both memory consumption and the runtime of the algorithm. For applications running on a JVM with a modern JIT compiler, this memory overhead is in most cases prohibitively high. Concretely, the path-profiling algorithm does not compute paths belonging only to a single Java method but paths within the entire compilation unit (we recall that a compilation unit is a Java method scheduled for compilation along with all its callee methods that the compiler decides to inline into it).[11] At the end of the compilation pipeline, where our technique inserts instrumentation code, the path-profiling algorithm computes paths using the control-flow of an IR whose size is bloated not only due to inlining [90] but also due to optimizations such as loop unrolling [61], path duplication [63], and various code lowering [108]. For this problem to manifest itself, it is not necessary that the

[11]We note that our technique does not introduce any new phase to resolve or inline additional methods to compute more interprocedural paths. Our goal is to observe the code emitted by the compiler without altering the compiler behavior.

path-count explosion happens in every compilation unit—even if a single compilation unit contains a control-flow pattern that contains an exponential number of paths, profiling becomes infeasible.

Example. We executed multiple experiments to profile the Renaissance benchmark suite [93] with a profiler that implements the path-counting strategy. The evaluation settings that we used were the same as the ones reported later in Section 5.2, with the difference that the profiler implements the standard path-profiling algorithm [9] and not the modified algorithm that we propose in this section. The profiler assigns a unique positive `int` identifier to each path. Using an `int[]` array to represent the counters, it is possible to represent sets of paths P_E where $|P_E| \leq 2^{31}$.

By executing certain benchmarks (for example, this is consistently reproducible in several compilation units in *akka-uct* [1], *als* [122], and *reactors* [82, 84, 92]), we noticed that the standard path-profiling algorithm produces a total path-count $|P_E|$ that exceeds 2^{31}, making it impossible to represent all paths. To see why this happens, consider the control-flow graph in Figure 19(a), in which blue boxes represent basic blocks (i.e., non-branching sequences of fixed nodes). Each time that the control-flow branches and then merges again at a basic block B, the total number of paths from the entry-point to B must be multiplied with the total number of paths from B to the control-flow sink. If there are N consecutive 4-way branch-and-merge patterns, then the total number of paths is 4^N.

In the Graal IR, branch-and-merge patterns are quite common after the last lowering—the major contributors are monitorenter and monitorexit instructions, which are lowered to fast-path spin locks and slow-path OS calls [26]; object allocations, which are lowered to **thread-local allocation buffer (TLAB)** bumps [39] on the fast path, and foreign calls on the slow-path; and polymorphic inlining [24], which inlines and dispatches between several possible call targets.

Discussion. There are multiple ways to overcome this limitation. One approach is to increase the number of identifiers—for example, a `long`-typed path identifier can represent sets of paths P_E such that $|P_E| \leq 2^{63}$. However, we note that a total path-count $|P_E| = 2^{31}$ already results in a prohibitively high memory consumption that impairs the practical usage, even if we consider a favorable scenario with an implementation that tries to minimize memory allocation. Consider the case when $|P_E| = 2^{31}$. The size of the path-count array p_c, which stores the 8-byte counters, would be $|p_c| = 2^{31} \cdot 8B = 16GB$. Moreover, given $|E| = 43$ (since the profiler used for this experiment can track 43 different event types, as further discussed in Section 5.2) and considering a memory-optimized path decoding table p_{ec} that stores each element as `byte` (i.e., a path-decoding table in which each event type may occur at most 255 times in the same path, which is a generous assumption—it is likely that there are occasional compilation units that violate this restriction), the memory consumption of p_{ec} would be $|p_{ec}| \cdot 1B = |E| \cdot |P_E| \cdot 1B = 43 \cdot 2^{31} \cdot 1B = 86GB$. Hence, the total memory consumption would be $|p_c| + |p_{ec}| = 16GB + 86GB = 102GB$. Note that this would be the memory footprint for only a single compilation unit in which that path explosion happened. This memory overhead is prohibitively high for practical purposes.

Moreover, even in cases in which this memory footprint is acceptable, the compilation-time overhead of populating the decoding table is prohibitive for a JIT compiler. In the benchmarks in which we witnessed the path-count explosion, the compiler threads that were assigned to the corresponding compilation units were blocked for hundreds of seconds while populating the path-decoding table.

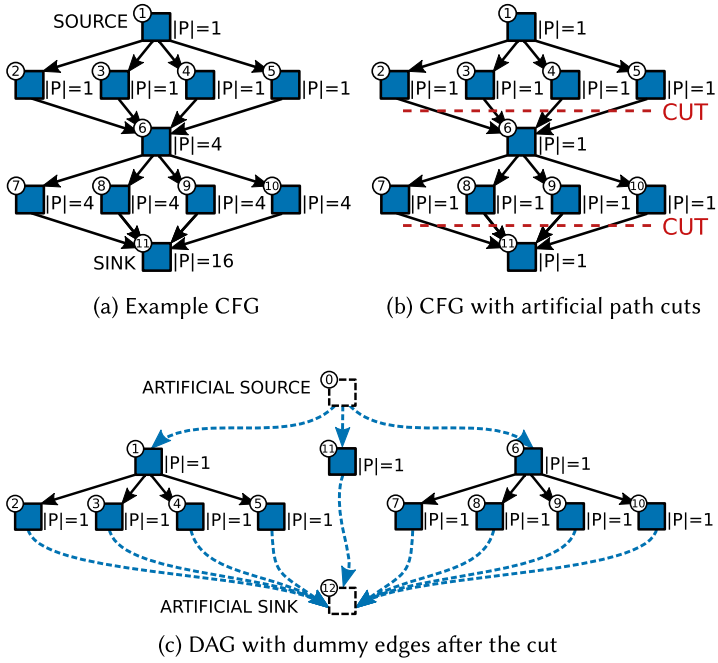
Another approach could be to maintain the path counts in a dynamically sized sparse array (for example, a hash table), and to reconstruct the decoding table from the persisted IR on-demand during shutdown, but only for those paths that have a non-zero path count (speculating that most paths are never executed during the program runtime). We did not pursue this approach for two reasons: first, hash-table updates are more expensive than single-memory location increments, and

our goal is to keep the overheads low. Second, in some compilation units, the program may execute all path combinations in the worst case, which would again lead to a prohibitive memory overhead.

Due to the huge memory consumption even in the optimistic scenario outlined above, and the fact that other workloads produce even larger total path counts $|P_E|$, we decided to implement an approach that decreases the total number of paths $|P_E|$ by cutting them into smaller pieces.

Path cutting. To lower the total path-count $|P_E|$, and consequently reduce memory consumption and compilation time, we modify the standard path profiling algorithm [9] to introduce artificial path cuts. When building the **directed-acyclic graph (DAG)** of the compilation unit's **control-flow graph (CFG)**, we cut the graph in half at *merge nodes*, i.e., at nodes with multiple inbound control-flow edges, by introducing dummy edges to the sink node, i.e., the single node with no outbound control-flow edges. We call this optimization *path cutting*.

Figure 19 shows an example of path cutting. In particular, we report the CFG of a certain compilation unit without artificial path cuts (Figure 19(a)), the CFG that highlights the artificial path cuts (Figure 19(b)), and the resulting DAG with the dummy edges after the cut (Figure 19(c)). Consider the first CFG in Figure 19(a), where blue nodes with solid borders represent basic blocks and are numbered from 1 to 11. Block 1 represents the source (entry point) and 11 represents the block with a sink node. Black arrows that connect basic blocks represent the control-flow edges. Next to each basic block, the figure reports the number of paths $|P|$ that lead to the basic block if path cutting is not performed. For example, while node 2 is reached by a single path (<1, 2>), four paths lead to node 6 (<1, 2, 6>, <1, 3, 6>, <1, 4, 6>, and <1, 5, 6>). Since the CFG has a single sink node, i.e., node 11, the number of paths $|P| = 16$ associated to the sink is the total path-count in that compilation unit.



(a) Example CFG (b) CFG with artificial path cuts

(c) DAG with dummy edges after the cut

Fig. 19. Example of path cutting.

We note that the two subgraphs composed of nodes 1-6 and 6-11 represent the typical control-flow of a `switch` statement. In modern compilers, such a control-flow is often generated when performing polymorphic inlining [24]—inlining virtual method calls by introducing a switch case on the dynamic type of the receiver object. In our example, the two subgraphs could have been generated by inlining two consecutive virtual calls where each of them has four possible call targets. We remark that the example represents a simplified scenario, while in real-world programs compilation units have far more complex CFGs. Without path cutting, the instrumentation code would be inserted only at node 11 and would handle all 16 paths, i.e., path tracking selects one

among the 16 paths that lead to node 11 and provides the path identifier to the instrumentation code of either path decoding or path counting.

The second CFG (Figure 19(b)) conceptually explains path cutting by showing the effect it has on the number of paths $|P|$ leading to any node. In particular, after cutting the graph at nodes 6 and 11, every node has $|P| = 1$.

In the DAG of Figure 19(c), we show the actual transformation performed by the algorithm on the CFG. The figure shows the three subgraphs created by the two artificial cuts, i.e., the first subgraph is composed of nodes 1-5, the second subgraph is composed of nodes 6-10, and the third subgraph is composed of only node 11. First, path cutting adds an artificial source node (white node 0 with dashed border) and an artificial sink node (white node 12 with dashed border). The artificial source node is connected to the actual source node (number 1) with a dummy edge shown in dashed light blue. Similarly, the actual sink node (number 11) is connected to the artificial sink node. Then, the inbound edges of nodes 6 and 11 (i.e., where artificial cuts are performed) are replaced with two dummy edges from the artificial source node leading to such nodes 6 and 11. Finally, the sink nodes of every subgraph (nodes 2-5 and 7-10) are connected with a dummy edge to the artificial sink node 12. This is similar to how loop-back edges are typically eliminated in standard path-profiling [9]. The total path-count $|P_E|$ is now equal to 9 (instead of 16), i.e., the sum of the $|P|$ associated to the sink nodes of every subgraph (nodes 2-5, 7-10, and 11). Instrumentation code is inserted at such sink nodes. For example, instrumentation code inserted at node 2 handles the path <1, 2>, instrumentation code inserted at node 3 handles the path <1, 3>, and instrumentation code inserted at node 11 handles the path <11>.

Even though path cutting increases the number of insertion points of instrumentation code (in Figure 19(c), instrumentation code is inserted nine times instead of only once at node 11), the total path count $|P_E|$, and hence the memory consumption, decrease. This optimization makes path-profiling strategies viable in a modern optimizing compiler such as Graal. In Section 6, we evaluate the path-decoding and path-counting strategies with path cutting enabled.

5 USE CASES

In this section, we describe two use cases where our approach was used to analyze the behaviour of Graal during JIT compilation. We first motivate the use cases by illustrating the common strategies used by compiler developers to analyze optimizations and debug issues (Section 5.1); then, we describe the settings used to obtain experimental data (Section 5.2). In Section 5.3 we describe a use case in which our approach explains why a performance improvement occurs, and in Section 5.4 we investigate an unexpected slowdown introduced by one of Graal's compiler phases.

5.1 Motivation

When developing new optimizations, compiler developers need to carefully analyze the effects of their changes on the compiled code to assess the benefits. However, this task can be challenging. Modifying the IR representation of the program in one phase not only affects the finally emitted machine code, but also the behaviour of other subsequent compiler phases (that will receive a different IR as input). Due to the complexity of the compilation pipeline, the effect of this change can be unpredictable. Moreover, since JIT compilation becomes non-deterministic when compilation is done by compiler threads (and not by application threads), reproducing a specific, previously observed behaviour in a subsequent run can be difficult [41, 72].

A straightforward way to assess the effect of an optimization is to compare the execution time of a benchmark when the optimization is activated and deactivated. When the performance impact is significant, this can confirm that an optimization yields a speedup, but the execution time alone does not identify the precise source of the speedup. A more illuminating approach is to use

debugging tools, such as **Ideal Graph Visualizer (IGV)** [74, 121] for C2 and Graal, to compare the IRs of the compiled units in which most of the execution time is spent. While this strategy in principle allows determining whether the optimizations work as intended, this is a tedious, time-consuming task in practice—after inlining, the IR typically contains thousands of nodes.

With our technique, compiler developers can compare runtime-event metrics that describe the runtime behavior of the program. These metrics can explain the effects of the different compiler phases. The collected metrics are more informative than those provided by commonly used tools such as **JDK Flight Recorder (JFR)** [75] or Oracle Developer Studio [77], since they can track event types that exist in the IR but not in the low-level machine code. Example of such event types include the lock implementation that the compiler selects [25–27, 108] (e.g., in HotSpot, whether the lock is biased, inflated, recursive, or stubbed), safepoints [23], GC-card writes [116], GC-write barriers [52], deoptimizations [50], concrete implementations of typechecks [20], polymorphic-inline-cache hits [51, 90], and guards denoting speculative optimizations. In addition, metrics may reveal information that is not always visible from the execution time or clock cycles [11, 98–100], since thread scheduling may introduce stalls, contention, or non-deterministic garbage collection. In the case of compiler bugs that compromise program performance, our methodology can be used to pinpoint the source of the slowdown, providing dynamic information on the IR instructions that are analyzed. In this sense, our metrics simplify debugging and help compiler developers identify problems.

5.2 Experimental Settings

The experimental data collected in the presented use cases was obtained with an event profiler for Graal, which is implemented as described in Sections 4.2.3 and 4.3, i.e., resorting to the path-counting strategy and the path-cutting modification to efficiently count the number of event occurrences in JIT-compiled code. Table 2 lists the 43 event types that we collect with our profiler. For clarity, event types are divided in categories based on their semantics. The name of each category is reported in bold before the set of event types it contains. For each event type, we report the unique event-type name and a brief description. In the following text, we use the term *metric* to refer to an event counter.

Analyses were conducted on a machine equipped with an 8-core Intel Xeon E5-2680 (2.7 GHz) and 64 GB of RAM. Frequency scaling, turbo boost and hyperthreading were disabled, and CPU governor was set to "performance". The Xeon E5-2680 CPU supports the x86 **Advanced Vector Extensions (AVX)**, but not the Haswell New Instructions (AVX2). The machine ran the Linux Ubuntu operating system (kernel version 5.4.0-58-generic), and a custom GraalVM Enterprise build that includes our event profiler. The VM is based on an open-source fork of OpenJDK 8 that adds **JVM Compiler Interface (JVMCI)** capabilities to the VM [76].

We perform the analyses described in the use cases on Renaissance [93, 94], a benchmark suite with a total of 25 benchmarks used for evaluating compilers [49, 56, 68, 79, 127]. We let each benchmark in Renaissance warm up by executing it for a given number of iterations (specified by the documentation) until dynamic compilation and garbage-collection ergonomics are stabilized, before activating our profiler. We do not execute the program using a more deterministic execution mode (e.g., disabling background compilation) since we favor a setting suitable to be used in a production environment. The event counts reported in this section are collected only during the last (steady-state) iteration.

5.3 Analyzing Compiler Optimization Phases

Here, we describe how compiler developers can use our methodology to analyze the effect of an optimization and understand the reason behind a performance improvement. We first outline the

Table 2. List of Profiled Event Types

Event-Type Name	Description
	allocations
allocation	Heap allocation.
array-duplication	Heap allocation of an array by copying an existing one.
	arithmetics
arithmetic	Every arithmetic operation including but not limited to the next ones.
int-div-rem	Integer division/reminder operation.
float-arithmetic-32	Every 32-bit floating point arithmetic operation including but not limited to the next one.
float-div-rem-32	32-bit floating point division/reminder operation.
float-arithmetic-64	Every 64-bit floating point arithmetic operation including but not limited to the next one.
float-div-rem-64	64-bit floating point division/reminder operation.
	arraycopies
arraycopy	Execution of `System.arraycopy`.
	atomics
cas	Atomic compare-and-swap operation.
	branches
if	Branch operation.
loop-end	Loop backedge.
	invokes[12]
foreign-call	Invocation of a native C/C++ function.
invokestatic	Invocation of a static Java method.
invokespecial	Invocation of a Java constructor, private method, or superclass method.
invokevirtual	Invocation of a Java instance method.
invokeinterface	Invocation of a Java interface method.
	locks
biased-lock-entry	Biased lock acquisition [101] (only ever held by a single thread).
cas-lock-entry	Compare-and-swap-based lock acquisition (short spin-based lock).
inflated-lock-entry	Inflated lock acquisition [101] (lock's information in a separate object).
recursive-lock-entry	Recursive lock acquisition (acquiring an already held lock).
stubbed-lock-entry	Stubbed lock acquisition (VM call).
	memory-accesses
store	Store to an object field or array element.
load	Load from an object field or array element.
	memory-barriers
membar-store-load	Execution of a memory barrier, i.e., an operation to enforce ordering constraints on memory accesses.

(Continued)

[12]We note that we do not provide an *invokedynamic* event type since the bytecode parser transforms *invokedynamic* instructions into *invokestatic* and *invokevirtual*.

Table 2. Continued

Event-Type Name	Description
	safepoints
safepoint	Event type that indicates a position in the IR where all threads are paused so that the VM can execute stop-the-world operations (e.g., garbage collection, root-pointer scanning, or classloading).
	speculative-ops
guard	Every guard execution, i.e., an IR node that potentially deoptimizes based on the outcome of a certain condition. This event type includes but is not limited to the next four.
guard-bounds-check	Guard that performs an array-bounds check.
guard-class-cast	Guard that performs a class-cast check.
guard-null-check	Guard that performs a null check.
guard-unreached-code	Guard that checks the presence of code that should not be reached.
deoptimization	Compiler deoptimization to transfer the control back to the interpreter.
	typechecks
instanceof	`instanceof` check.
typeswitch	Lookup based on the type of the given input.
	unwinds
unwind	Unwind operation, i.e., find and execute the exception handler defined in a caller method.
	vectorization
bulk-map	Transformation of a sequence of elements.
bulk-read	Read of a sequence of elements.
bulk-write	Write of a sequence of elements.
bulk-initialize	Initialization of a sequence of elements using one value.
	wait-notify
wait	Invocation of `Object.wait`.
notify	Invocation of `Object.notify`.
notify-all	Invocation of `Object.notifyAll`.
	writebarriers
writebarrier	Writebarrier operation needed by the garbage collector.

general approach, and then use it to explain the reason for a speedup achieved by the **optimistic aliasing analysis (OAA)** phase in Graal.

Approach. Given a compiler optimization phase ω that we want to analyze, we aim at identifying all metrics that significantly differ due to ω. Increasing or decreasing the occurrence of a given event type is usually the primary goal of adding a new phase, since this change should in turn improve the effect of subsequent compiler phases on the compilation unit, ultimately decreasing its execution time. For example, one goal of the method-inlining phase is to reduce overheads due to method calls, and thus decrease the value of the metrics in the invokes category. Measuring the changes in the metric allows analyzing the behaviour of the optimization ω.

We define Γ as the set of all phases in the compilation pipeline (thus $\omega \in \Gamma$). Then, we define $count(p, m, \Upsilon)$ as a function that returns the mean value of a given metric m across executions of the program p, while executing only a subset of phases $\Upsilon \subseteq \Gamma$. The reason why we define *count* as a mean value is that, due to the inherent non-determinism of JIT compilation (when the compiler threads are separate from the application threads), the value of the each metric is a random variable. This is because the optimizations make decisions based on asynchronously collected profiles, which makes the IR of the compilation units slightly different in each execution.

For example, $count(scrabble, \text{allocation}, \Gamma)$ and $count(scrabble, \text{allocation}, \Gamma \setminus \{\omega\})$ return the mean value of the allocation metric for the execution of the *scrabble* benchmark with and without the ω phase, respectively. To obtain these values in practice, we repeat the *scrabble* program many times, and compute the mean.

Our goal is to determine the set of metrics $\{m\}$ whose mean values differ by more than a given threshold $\varepsilon(p, m)$, between one execution with ω and one without ω of the same program p. We call this set M_p, where p is one program taken from the set of all the possible programs Π. M_p is defined as follows:

$$M_p = \{m \mid |count(p, m, \Gamma) - count(p, m, \Gamma \setminus \{\omega\})| > \varepsilon(p, m)\} \tag{3}$$

The $\varepsilon(p, m)$ threshold allows us to extract only a subset of the metrics that report a significant difference, and allows us to ignore small variations caused by non-determinism of the JIT compilations in different executions of the same program (even when the program itself is completely deterministic). In turn, we define ε as follows:

$$\varepsilon(p, m) = \max(count(p, m, \Gamma), count(p, m, \Gamma \setminus \{\omega\})) \cdot t \tag{4}$$

The parameter t represents the tolerated percentage difference between the two values, and can be tuned to extract different subsets of metrics. Higher values of t restrict the result set to the metrics that differ the most, while low values of t enlarge this set.

Our approach consists of collecting metrics and evaluating Formulas 3 and 4 on a significant set of programs Π, in this way identifying metrics that are most affected by the optimization ω.

Optimistic Aliasing Analysis. We use the aforementioned method to analyze the effects of Graal's **optimistic aliasing analysis (OAA)** phase on the Renaissance benchmark suite. The OAA phase speculatively inserts information into the IR about which object pointers are the same (i.e., are *aliased*) and which are different. The aliasing information can be utilized by optimizations such as read elimination, write elimination, GC-write-barrier elimination, or vectorization [10]. For instance, if two pointers are confirmed to be aliased, then some of their GC write barriers can be fused. Similarly, if two array pointers do not alias, then element copying between the corresponding arrays can be vectorized (because they do not overlap). In practice, it is *a priori* unclear which optimizations are improved (and to which extent) when adding an implementation of OAA to a complex compiler such as Graal. We noticed that enabling OAA introduces significant speedups in certain benchmarks, i.e., up to 15% in the *als* benchmark, but we do not know whether the performance gain is due to vectorization, GC-write-barrier elimination, or some other optimization. Hence, we analyze the impact of OAA with our profiler.

We apply the aforementioned approach, setting the parameters of Formulas 3 and 4 as follows: $\omega = OAA$, $\Pi =$ Renaissance benchmarks, and $t = 0.1$. We execute the benchmark suite using our profiler, turning OAA on and off. Then, we collect the set M_p for all benchmarks in $p \in \Pi$. We notice that all the M_p sets are empty, apart from M_{als}, whose metrics are reported in Table 3. The values in the table indicate that OAA has a significant effect on *als*, as several metrics vary by orders of magnitude. In particular, the vectorization metrics (i.e., bulk-map, bulk-read, bulk-write, and bulk-initialize) are much higher when OAA is enabled, suggesting that OAA enables

Table 3. Metrics Significantly Affected by the Optimistic Aliasing Analysis (OAA) Phase in the *als* Benchmark

Metric (m)	count(als, m, Γ)	count(als, m, Γ \ {OAA})
bulk-map	989 105 986	71 523 670
bulk-read	503 935 809	45 313 789
bulk-write	517 377 287	60 224 210
bulk-initialize	457 782 740	830 084
loop-end	937 458 858	2 914 938 151
if	8 459 893 954	9 574 417 323
arithmetic	18 709 056 784	42 322 473 348
float-arithmetic-64	1 326 420 147	5 675 174 306

the vectorization of more loops. The loop-end and if metrics (i.e., the number of loop iterations and branch instruction executions, respectively) confirm this statement—since vector operations decrease the number of loop iterations, as the vectorization metrics increase, the loop-end and if metric decrease. Moreover, vectorization decreases floating point scalar operations related to the *als* benchmark and scalar arithmetic operations required to compute loop indexes and memory addresses. This explains the decrement of the float-arithmetic-64 and arithmetic metrics, respectively. We note that, even though the bulk-read and bulk-write metrics are significantly affected by OAA, the table does not report the high-level load and store metrics, meaning that OAA does not have a significant effect (in the sense of Equations (3) and (4)) on the number of accesses to object fields and array elements. The main difference is in the accesses that can be parallelized by vectorization: when OAA is enabled, most of these accesses are vectorized, and when OAA is disabled, most of them are scalar.

From these results, we can conclude that the OAA phase in Graal mainly improves vectorization, while other phases do not seem significantly affected. This finding implies that the 15% performance speedup in *als* from enabling OAA is mostly caused by the vectorization phase.

Discussion. Our approach allowed us analyzing the effects of OAA by profiling the benchmarks in two settings—with OAA activated and deactivated. The values of the collected metrics clearly pinpointed that the OAA phase mainly improves vectorization. With this knowledge, one may think that we could have reached the same conclusion by inspecting the execution time of the benchmarks, activating and deactivating OAA and vectorization individually. While this method easily allows validating our conclusion *a posteriori*, it is unpractical *a priori* for two reasons. First, without already knowing that the OAA phase mainly improves vectorization, one would need to activate and deactivate all the phases individually to find the one whose deactivation leads to a slowdown that is close to the speedup enabled by OAA, which most likely requires significant computational time. Second, a drop in performance when deactivating OAA can imply that some other optimization was affected (not necessarily vectorization), potentially pointing the compiler developer to the wrong conclusion. Instead, our approach can easily find a clear cause-effect correlation by just profiling metrics in two settings, saving significant computational time.

5.4 Simplifying Debugging

We now describe a case study in which our profiler helped us identify the cause of a slowdown. In particular, we noticed that Graal's ***convert-deoptimizations-to-guards* (CDTG)** phase introduces an unexpected slowdown (around 7%) when executing the *scala-doku* benchmark. This benchmark heavily exercises Scala collections [71, 88], and the hash-trie data structure [6, 83, 85–87, 89]. We would expect a compiler phase to consistently speed up the program, or at least to

Table 4. Metrics Significantly Affected by the Convert Deoptimizations to Guards (CDTG) Phase in the *scala-doku* Benchmark

Metric (m)	**count(scala-doku, m, Γ)**	**count(scala-doku, m, Γ \ {CDTG})**
allocation	164 016 436	4 932 271
instanceof	1 020 482 952	581 007 038
load	1 260 071 141	1 101 575 841
invokestatic	346 730	590 133
invokevirtual	36 789 588	141 577
if	3 628 779 099	2 482 745 508
arithmetic	2 481 575 004	2 004 573 745
safepoint	162 328 445	40 192 560
guard	1 472 294 639	794 692 405

not cause slowdowns. Hence, we are interested in understanding the reasons for the performance slowdown to fix the issue.

Graal uses *deoptimization nodes* to represent deoptimization points in the IR [50], which transfer the control back to the interpreter. In addition, Graal uses *guard nodes*, i.e., nodes that represent a *potential* deoptimization [29], which happens only when a certain condition is violated. The CDTG phase finds and converts branches that end with a fixed deoptimization node to fixed guard nodes. Fixed guard nodes are later transformed to floating guard nodes, and this conversion usually enables subsequent optimizations—due to fewer ordering constraints between floating nodes, it is often possible to pull guards out of loops, or coalesce guards when the condition of one guard implies the other guard's condition. For example, if two guards occur next to each other, one with condition "`x instanceof Integer`" and another "`x instanceof Number`", then the second condition is always true (`Integer` is a subtype of `Number` in Java), and the corresponding guard can be removed.

To analyze the cause of the slowdown in CDTG, we follow the approach from the previous section—we collect the metrics of interest in *scala-doku* including and excluding CDTG, and obtain the $M_{scala\text{-}doku}$ set of metrics that differ significantly. Table 4 reports these metrics, obtained with Formulas 3 and 4, and the following parameters: $\omega = CDTG$, $p = scala\text{-}doku$, and $t = 0.1$.

We identify now metrics in Table 4 that may be correlated. For convenience, we define $m_- = count(scala\text{-}doku, m, \Gamma \setminus \{CDTG\})$ and $m_+ = count(scala\text{-}doku, m, \Gamma)$ for a certain metric m. For example, we write if_- to indicate $count(scala\text{-}doku, \text{if}, \Gamma \setminus \{CDTG\})$. Since CDTG converts deoptimization nodes to guard nodes, we expect $guard_+$ to be significantly higher than $guard_-$. Indeed, the former is almost twice the latter, as CDTG has introduced $(guard_+ - guard_-) \approx 678$ million guard-node executions.

In a successive phase, guard nodes are lowered and become again if branches (i.e., back to a lower-level abstraction, from which code can be more easily generated). If the quantity $(if_+ - if_-)$ is roughly the same as $(guard_+ - guard_-)$, then the additional if nodes would be caused by the lowering of the guard nodes introduced by CDTG. However, this is not the case: as we can see from the table, $(if_+ - if_-)$ amounts to roughly 1.2 billion, which is far above 678 million. This implies that other phases caused a significant increase in if nodes, and were thus influenced by CDTG.

Consider the instanceof metric in Table 4, which is typically related to if nodes, since typechecks are often executed in conditional statements. We can see that CDTG also causes the execution of $(instanceof_+ - instanceof_-) \approx 440$ million more instanceof nodes. The joint increment of if and instanceof can be a sign that additional compiler-generated instanceof nodes were executed. For example, with polymorphic inlining [51], the compiler can emit an if-elseif chain of typechecks whose conditions fail more often. This chain is created based on the receiver-type profile at the

respective callsite—the chain checks against most probable receiver types first, and less probable ones later. In case of a successful typecheck, a direct call (which can be inlined) is inserted. The if and instanceof metrics suggest that CDTG negatively affects the receiver-type profiles that polymorphic inlining relies on when emitting the if-elseif chain, which in turn causes the execution of more typechecks, leading to a slowdown.

To verify our conclusion, we determined the hottest compilation unit of the *scala-doku* benchmark,[13] which is `HashTrieSet.foreach`, and we analyze the IR of this single method before and after performing CDTG. As expected, we found callsites at which the profiles used for the polymorphic inlining were more polluted with wrong types in the execution that uses CDTG compared to the execution without CDTG. We found that the differences in deoptimizations caused different compilation orders, so different callers of the `HashTrieSet.foreach` method were compiled at different points in time, which furthermore caused the differences in the receiver-type profiles that the interpreter collected at callsites within `HashTrieSet.foreach`.

To summarize, using our profiling approach, we were able to explain the 15% performance gain obtained by the OAA phase in *als* and to identify the unexpected cause of a 7% performance slowdown introduced by the CDTG phase in *scala-doku*, without manually comparing the IR of the method, which is an error-prone and time consuming approach. Our profiler was essential in collecting the metrics of interest, since existing tools [75, 77] are not able to collect low-level compiler internal metrics such as the *guard* count. We reported the issue with CDTG to the Graal developers who validated it and are currently considering potential solutions to avoid the reported performance degradation.

6 PERFORMANCE EVALUATION

In this section, we evaluate the proposed profiling approach. We first present the experimental setup in Section 6.1, and analyze the execution-time overhead of our technique in Section 6.2. Then, we evaluate the compilation-time in Section 6.3, the code-size overhead in Section 6.4, and the memory consumption in Section 6.5.

6.1 Evaluation Settings

We perform our experiments using the same machine and experimental setting that were described in Section 5.2. In addition to Renaissance [93–95], we evaluate our approach on the DaCapo [13] benchmark suite, using the input size and the number of warmup iterations reported in Table 5. From DaCapo, we exclude benchmarks *tomcat*, *tradebeans*, and *tradesoap* due to well known issues [111, 112], and *batik* since it employs non-standard classes from a package (`com.sun.image.codec.jpeg`) that is not available in OpenJDK, on which our evaluation VM is based [42]. Following the recommendation of the DaCapo developers [22], we consider benchmark *lusearch-fix* in place of *lusearch*.

We evaluate our technique on four different event type categories (i.e., we profile four different subsets of event types separately) listed in Table 2: *all*, *arithmetics*, *branches*, and *arraycopies*. Category *all* contains all the event types that our technique supports and allows evaluating the performance of our approach when simultaneously profiling many event types that occur frequently in the application code ($\approx 2.35 \cdot 10^6$ *occurrences*/*ms* on average). Profiling *arithmetics*, *branches*, and *arraycopies* separately allows evaluating our approach when profiling a few or just one event type in isolation. In particular, *arithmetics* and *branches* represent event types that occur frequently at runtime, on average $\approx 6.76 \cdot 10^5$ *occurrences*/*ms* and $\approx 5.11 \cdot 10^5$ *occurrences*/*ms*, respectively. The

[13] While path counts could be used to deduce the most frequently executed compilation unit, this can also be done with various external tools, and we used Oracle Developer Studio [77].

Table 5. Input Size and Number of Warmup Iterations of the Considered DaCapo Benchmarks

DaCapo Benchmark	Input Size	# w.u.
avrora	large	20
eclipse	large	10
fop	default	40
h2	huge	10
jython	large	20
luindex	default	40
lusearch-fix	large	20
pmd	large	20
sunflow	large	20
xalan	large	20

former is composed of six event types, while the latter contains only two event types. Both categories highly benefit from path-based optimizations when profiled. On the other hand, *arraycopies* represents a single infrequent event type (on average $\approx 8.42 \cdot 10^2$ *occurrences*/*ms*) whose profiling hardly benefits the complex path-based optimizations.

We compare three different versions of our event profiler. The first version implements the direct event counting strategy (Section 4.2.1), the second version implements the path-decoding strategy (Section 4.2.2), while the third one implements the path-counting strategy (Section 4.2.3). For brevity, in the following text, we abbreviate the three versions with the terms ***dec* (direct event counting)**, ***pd* (path decoding)**, and ***pc* (path counting)**. We note that all strategies perform the instrumentation at the target position in the compilation pipeline, and insert markers in the IR. The *pd* and *pc* strategies use the path-cutting modification (Section 4.3). We remark that the *pd* and *pc* implementations are the ones proposed in this article, while *dec* serves as a comparison to show the benefits of the other two approaches.

For each event-type category, profiler version, and benchmark, we report the arithmetic mean obtained on 10 steady-state iterations, except for the compilation-time overhead (Section 6.3) and code size overhead (Section 6.4) where the arithmetic mean instead includes the compilation time and code size across all iterations, respectively. The reason is that the analyzed benchmarks were designed to avoid the generation of new code at runtime [93], so once steady state is reached, no more compilations occur. That is, compilation time is always zero and code size does not vary in the steady state of the evaluated benchmarks.

We note that we do not present an experimental comparative evaluation with prior work since, to the best of our knowledge, there is no publicly available related technique that is suitable for comparison. Related techniques (detailed later in Section 8) either lack open source implementations, focus on different goals/event types, or target different platforms (e.g., Jikes RVM [3]). Moreover, bytecode-level instrumentation approaches are not suitable for comparison since they collect significantly inaccurate event counts, as related work [125] shows, and both bytecode-level and machine-code-level instrumentation cannot collect compiler-internal event types.

The approach that is the closest to our technique is the one proposed by Zheng et al. [124, 125], which has a scope similar to ours and has also been implemented in a production Graal compiler. Unfortunately, a working implementation of their approach is currently not available, as it was removed from the Graal compiler many years ago. To compare the proposed approaches with their technique, one would need to significantly modify their technique/implementation such that their approach can be implemented in the latest version of Graal, which has undergone major

modifications in the last years. The implemented version of their technique would necessarily differ significantly w.r.t. the one described in their paper, decreasing the effectiveness of a comparison between their work and our approach. Finally, their technique can collect only a subset of the event types that we collect and does not employ path-profiling—their approach would instrument the program by inserting the instrumentation code before/after each event occurrence. This is similar to the behavior of the evaluated naive strategy *dec*, thus used for comparison against the proposed approaches.

6.2 Execution-time Overhead

Here, we evaluate the execution-time overhead introduced by our profiling technique. For each combination of an implementation and a benchmark, Figure 20 reports the overhead factor computed as $T_{instrumented}/T_{uninstrumented}$, where $T_{instrumented}$ refers to the execution time of an instrumented run (using a given implementation) and $T_{uninstrumented}$ refers to the execution time of an uninstrumented run. Each of the four plots of Figure 20 refers to a different event-type category (whose name is reported below the x-axis), in order: *all*, *arithmetics*, *branches*, and *arraycopies*. Benchmarks are reported on the x-axis of the plot, in alphabetical order, Renaissance benchmarks first (until *scrabble*) and DaCapo benchmarks later. The execution-time overhead is reported on the logarithmic y-axis. Above each bar, we report the exact value of the overhead. The black error bars represent the 95% **confidence interval (CI)** of the measurements.

First, we analyze the evaluation results when profiling all metrics. In terms of overhead factor, the *pc* implementation performs better than *dec* and *pd* in all benchmarks. Path-counter updates performed by *pc* are less frequent than the event-counter updates of the *pd* and *dec* strategies, and hence lead to lower overhead. The overhead of *pc* ranges from 1× (*akka-uct* and *scala-stm-bench7* [48]) to 1.59× (*finagle-http* [2]), with a geomean of 1.18×.[14] The *pd* strategy shows that online path decoding and bulk event-counter updates help in reducing the cost of single event-counter updates in every benchmark except *dotty*, *finagle-http*, *fop*, and *pmd*. In these benchmarks, different inlining decisions—taken by the JIT compiler and guided by different JVM-internal profiles—lead to higher execution time. We clarify this phenomenon in Section 7.3. Overhead factors of *pd* range from 1.04× (*akka-uct*) to 3.86× (*fop*), while overhead factors of *dec* range from 1.35× (*future-genetic*) to 12.08× (*als*). On average, *pd* and *dec* slow application execution down by 2.11× and 3.55×, respectively.

We discuss now the execution-time overhead of the other three evaluated categories. For almost every evaluated benchmark and implementation (for the exceptions, we refer to the phenomenon explained in the previous paragraph and later detailed in Section 7.3), the execution-time overhead of *all* is higher than the one of the other three categories, because *all* contains (among many others) the event types of the three other categories. Moreover, the execution-time overhead of *arithmetics* is higher than the one of *branches*, which in turn is higher than the one of *arraycopies*. This is expected, as in any dynamic profiling technique the execution-time overhead decreases together with the frequency of the event occurrences. The low frequency of *arraycopies* leads to an execution-time overhead close to 1 in all except a few benchmarks, such as *philosophers*, *eclipse*, and *jython*.

The execution-time overhead of *dec*, *pd*, and *pc* is correlated to the number of executed event occurrences, path-event combinations, and paths containing at least one event occurrence, respectively. For this reason, the difference between the execution-time overhead factor of *dec* for category *all* and the other three categories is relatively higher than the ones of *pd* and *pc*. This gap narrows for *pd* and even more for *pc*. In the case of *pd*, the gap narrows due to the aggregation

[14] Average overhead factors across multiple benchmarks are computed using the geometric mean.

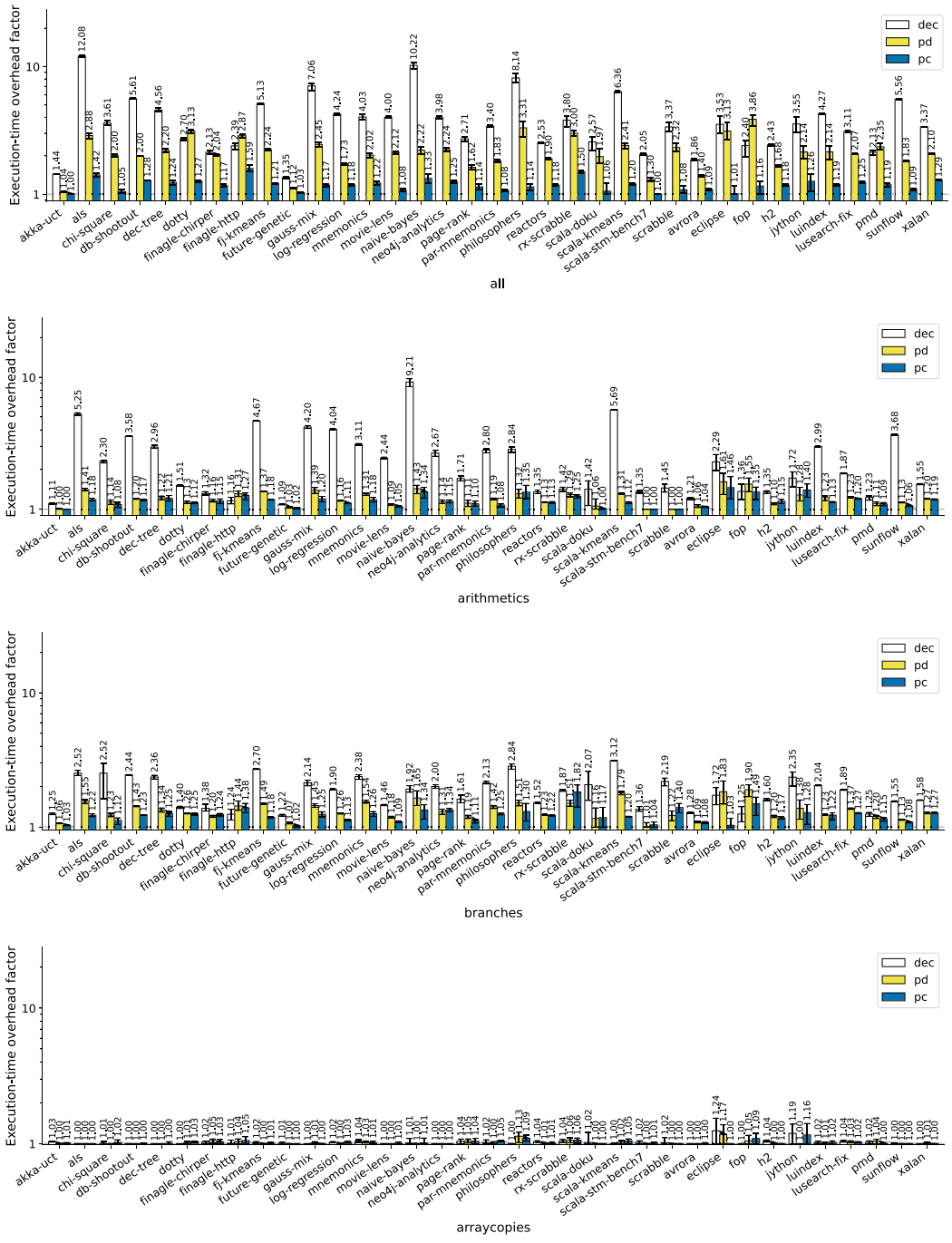


Fig. 20. Execution-time overhead factor of the *dec*, *pd*, and *pc* strategies.

of event occurrences in each path, while for *pc*, a narrow gap indicates that the number of executions of paths containing at least one event occurrence does not significantly differ between *all* and *arithmetics*, i.e., arithmetic operations occur in most of the program paths that contain at least one event occurrence.

Our results demonstrate the benefits of the proposed implementations (path decoding and path counting) in terms of reducing the execution-time overhead in a counting profiler. This reduction is enabled by path profiling with the path-cutting modification. In particular, by counting paths online and by relying on offline decoding, the *pc* strategy enables efficient collection of both large sets (e.g., category *all*) and small sets (e.g., categories *arithmetics* and *branches*) of frequent event types, significantly outperforming *dec*. The other proposed implementation, *pd*, reduces the runtime overhead compared to direct event counting, and is useful when real-time counters must be gathered during the execution of the program. The naive *dec* strategy is comparable to *pc* and *pd* only for collecting a few event types that do not frequently occur in program code, such as *arraycopies*. However, this is expected, any profiler has low overheads if profiling is infrequent.

6.3 Compilation-time Overhead

We now analyze the additional time needed by the JIT compiler to compile the code, and quantify the overhead of the additional instrumentation phases in the proposed approach. For each implementation, Figure 21 reports on the logarithmic y-axis the compilation-time overhead factor, that is, $CT_{instrumented}/CT_{uninstrumented}$, where $CT_{instrumented}$ refers to the compilation time of an instrumented run (using a given implementation) and $CT_{uninstrumented}$ refers to the compilation time of an uninstrumented run.

When profiling all the supported metrics, in every benchmark (except *future-genetic*, which we discuss later in this section), *pc* introduces less overhead than *dec*, which introduces less overhead than *pd*. In particular, the *dec* implementation slows down compilation by a factor ranging from 1.51× (*future-genetic*) to 2.42× (*jython*), the *pd* implementation from 2.14× (*chi-square* [122]) to 6.46× (*jython*), and the *pc* implementation from 1.29× (*page-rank* [122]) to 1.69× (*fj-kmeans* [59]). The average overhead factor is 2.01×, 3.06×, and 1.48× for *dec*, *pd*, and *pc*, respectively. The increase in compilation time across all benchmarks is because all the three implementations require additional compilation time to mark event occurrences, replace markers with IR, lower the IR, and emit instrumentation machine code. We note that *pc* introduces the smallest overhead while *pd* slows compilation down the most. Even though *pc* performs path profiling, which requires additional compilation time, the instrumentation code generated by *pc* is small (as discussed in the next section), which reduces the IR lowering and machine code emission time w.r.t *dec*, leading to lower compilation times—in *pc*, several event occurrences are replaced with a single path-counter update. This is not the case for *pd*, which inserts instrumentation code to perform path decoding and event-count update for each path-event combination (as explained in Section 4.2.2). The generation of such instrumentation code, combined with the time required to execute path profiling, makes the compilation with *pd* slower than with *dec*. Nonetheless, a higher compilation time is less critical than a high execution-time overhead, and does not make *pd* inapplicable to profile applications in real-time (as we practically verified on the Renaissance and DaCapo suites).

We note that the 95% CIs associated with a few benchmarks (such as *chi-square*, *future-genetic*, *eclipse*, *h2*, and *jython*) report high measurement variability regardless of the implementation used. However, such variability is not a consequence of the *dec*, *pd*, and *pc* implementations, since uninstrumented runs (i.e., the baseline) are also affected. We investigated the causes of this behaviour, and determined that variability is caused by the non-determinism in the VM and compiler. Indeed, the scheduling and interleaving of different threads could lead to different profiles—not produced by our approach but produced by the VM and later used by Graal to make compilation

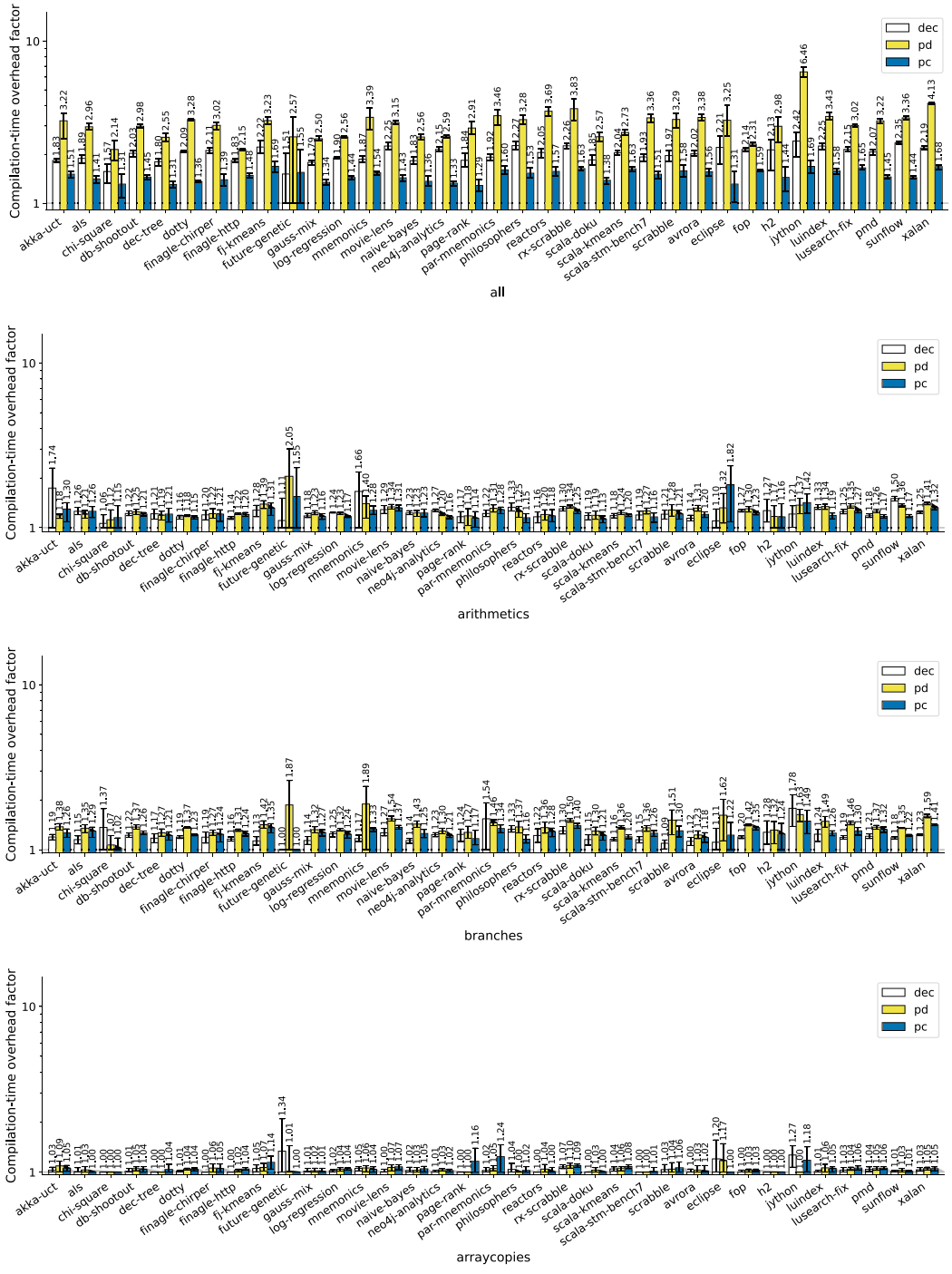


Fig. 21. Compilation-time overhead factor of the *dec*, *pd*, and *pc* strategies.

decisions—between different benchmark runs and therefore compilation of different methods among different benchmark runs. The authors of the Renaissance benchmark suite are currently investigating the sources of this high variability for benchmarks *chi-square* and *future-genetic*. We note that the high variability of these benchmarks will also characterize evaluation results in the subsequent sections.

We analyze now the evaluation results obtained on the other three categories. Similar to the results on category *all*, for almost every evaluated benchmark and event type category, *pd* slows down compilation the most. However, the difference with the other strategies is much smaller than when profiling all metrics. This is because the small event-type sets of these categories lead to a reduction in the number of path-event combinations. For the three evaluated categories and in contrast to category *all*, *dec* introduces a compilation-time overhead that is comparable to the compilation-time overhead of *pc*. This is because, as we will show in Section 6.4, the size of the instrumentation code emitted by *dec* is comparable to the size of the instrumentation code emitted by *pc*.

For the *pc* strategy, the sum of the overheads of the categories *arithmetics*, *branches*, and *arraycopies* is higher than the overhead of the category *all*. The reason is that the overhead of *pc* depends on the number of paths containing at least one occurrence of any event type and different event types may share the same paths (in contrast, the overhead of *dec* depends on the occurrences of the profiled event types, whereas the overhead of *pd* depends on the number of path-event combinations). In practice, most of the program paths instrumented for the category *all* are also instrumented for the categories *arithmetics* and *branches*, separately, i.e., *arithmetics* and *branches* occur in most of the program paths. This explains why the sum of their overheads is higher than the overhead of the category *all*. The *pc* strategy is particularly suitable to count several event types at the same time.

To summarize, the proposed *pc* approach allows efficiently profiling large sets of event types with the lowest compilation-time overhead. The compilation-time overhead of *dec* becomes comparable with the one of *pc* only when profiling small sets of infrequent event types. We remark that a high compilation time is a minor drawback compared to a high execution time overhead, which instead may impair real-time profiling in production settings.

6.4 Code-size Overhead

In this section, we analyze the code-size overhead of our approach, i.e., we quantify the additional memory space required to store the instrumentation code emitted by the JIT compiler implementing the proposed approaches. For each implementation, Figure 22 reports on the logarithmic y-axis the code-size overhead factor, that is, $CS_{instrumented}/CS_{uninstrumented}$, where $CS_{instrumented}$ refers to the code size of an instrumented run (using a given implementation) and $CS_{uninstrumented}$ refers to the code size of an uninstrumented run.

As done in the previous sections, we first discuss the evaluation results obtained when profiling all metrics. Our results confirm the explanation reported in Section 6.3—the instrumentation code generated by *pc* to update path-counters is small w.r.t. the instrumentation code generated by *dec* and *pd* when profiling several event types simultaneously. Moreover, the instrumentation code generated by *pd* is smaller than the one generated by *dec* for most of the evaluated benchmarks. Evaluation results confirm also that the emission of the additional instrumentation code (i.e., the code that corresponds to the additional code size) greatly influences compilation time. Without considering *jython*, we find that there is a strong positive correlation between code-size and compilation-time overheads, as remarked by the Pearson correlation coefficient [35] which is equal to 0.72, 0.62, and 0.85 for *dec*, *pd*, and *pc*, respectively. Considering *jython*, the Pearson correlation coefficient decreases to 0.64, 0.73, and 0.22 for *dec*, *pc*, and *pd*, respectively. Analyzing *jython*,

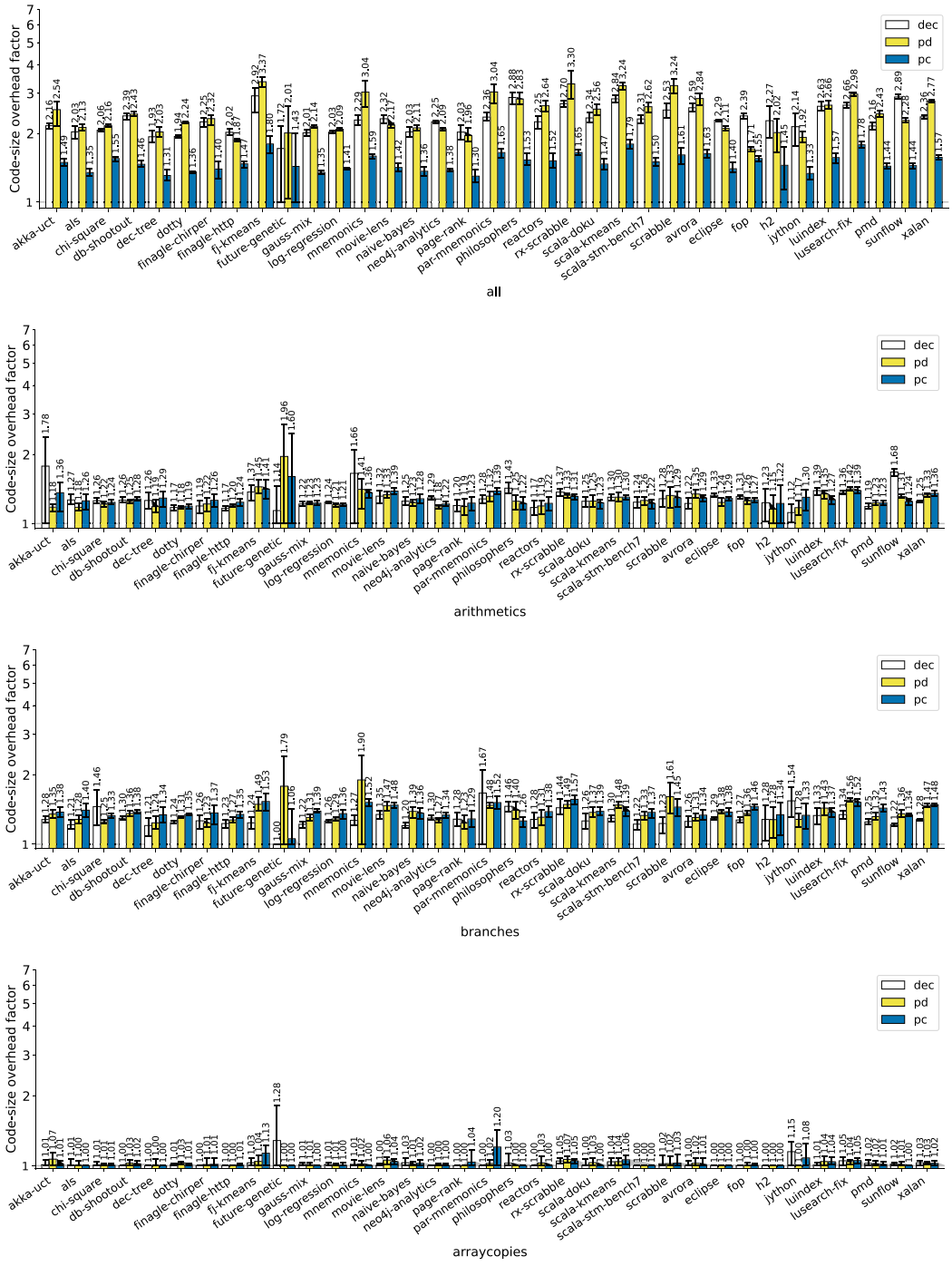


Fig. 22. Code-size overhead factor of the *dec*, *pd*, and *pc* strategies.

we discovered that Graal spends much more time in emitting code (and in particular in allocating registers) w.r.t. to the other benchmarks. When *jython* is instrumented using the *pd* strategy, even if *pd* emits less code than *dec*, this phenomenon is particularly evident. We are in contact with the Graal team that is investigating the compilation-time issue in the backend compiler for the *jython* benchmark. The *dec* implementation introduces a code-size overhead factor ranging from 1.71× (*future-genetic*) to 2.92× (*fj-kmeans*), while the code-size overhead of *pd* ranges from 1.71× (*fop*) to 3.37× (*fj-kmeans*), and the one of *pc* from 1.30× (*page-rank*) to 1.80× (*fj-kmeans*). The average overhead factor is 2.29×, 2.41×, and 1.49× for *dec*, *pd*, and *pc*, respectively, which is compatible to the average compilation-time overhead, as shown in the previous section. In comparison to *dec* and *pd*, the code-size overhead of *pc* is significantly lower when profiling all event types.

Now, we discuss the evaluation results of categories *arithmetics*, *branches*, and *arraycopies*. As the figure shows, the code size of *dec* considerably decreases together with the number of event occurrences, while the one of *pd* considerably decreases together with the number of instrumented event types. Since the code-size overhead of *pc* was already low for category *all*, its code-size overhead for categories *arithmetics* and *branches* decreases only slightly together with the number of paths. As detailed in Section 6.3 for the compilation-time overhead factor, we remark that most of the program paths instrumented for category *all* are also instrumented for category *arithmetics* and *branches*, separately. Hence, the code-size overhead of category *all* is lower than the sum of the code-size overheads of categories *arithmetics* and *branches* for *pc*. The code-size overhead of *dec* is comparable to the code-size overhead of *pc* and *pd* for categories *arithmetics* and *arraycopies* while it is slightly lower for *branches*. The reason is that category *branches* represents an unfavorable scenario for path-based approaches implementing path-cutting, in terms of code-size overhead. Consider for example the *dec* and *pc* strategies, the program paths in Figure 19(c), and a branch event occurrence in basic block 1 of Figure 19(c) that represents a jump to either basic block 2, 3, 4, or 5. In *dec*, the event occurrence leads to the generation of exactly one event-counter update in basic block 1. In *pc*, the event occurrence leads to the generation of four path-counter updates (one for each basic block 2, 3, 4, and 5). As a consequence, the code-size overhead of *pc* is higher than the one of *dec*.

We note that, even if the code-size overhead of the evaluated strategies is similar, the instrumentation code emitted by *dec* is significantly different from the one emitted by the other strategies. The instrumentation code of *dec* is inserted in correspondence to the event occurrences, while the instrumentation code generated by *pd* and *pc* is inserted at the end of program paths. For this reason, when profiling a few event types (depending on how frequently they occur), path tracking and counter updates may lead to the emission of more machine code w.r.t. *dec*. Path-based strategies may insert instrumentation code at the end of many program paths that are rarely or never executed, e.g., program paths that end with exception throwing or deoptimizations. The difference in the execution-time overhead (Section 6.2) and the code-size overhead is motivated by the fact that a program instrumented using *dec* performs many event-counter updates (and hence executes most of the emitted instrumentation code) while the same program instrumented using *pc* performs fewer path-counter updates (and hence executes only a minor sub-part of the emitted instrumentation code). We discuss other details of the update frequency of path-based strategies that employ path-cutting in Section 7.3.

To summarize, our evaluation results show that the code-size overhead correlates with the compilation-time overhead (Section 6.3). Hence, similarly to the compilation-time overhead, the proposed *pc* approach allows efficiently profiling large sets of event types with the lowest code-size overhead. When collecting a few event types (such as *arithmetics* and *arraycopies*), the code-size overhead of *pc* and *pd* is comparable to the one of *dec*. When profiling category *branches*,

that represents a particularly unfavorable scenario for path-based approaches implementing path-cutting, the code-size overhead of *dec* is only slightly lower than the one of *pc* and *pd*.

6.5 Memory Consumption

Here, we evaluate the memory consumption of the three strategies, i.e., the memory required to store the data structures referenced by the instrumentation code to implement the techniques as described in Section 4. As discussed in Section 4.2, an implementation based on direct event counting (*dec*) results in a memory consumption of $O(|E|)$, while an implementation based on path-decoding (*pd*) or path-counting (*pc*) in $O(|E| \cdot |P_E|)$. We implemented both P_E and E as long arrays.

The *dec* strategy only requires an array of length $|E|$ (to store the event count), while *pd* and *pc* require the allocation of two long arrays of different length. Following the terminology introduced in Section 4.2.2, *pd* allocates two arrays *cnt* and p_{ec} of length $|cnt| = |E|$ and $|p_{ec}| = |P_E| \cdot |E|$, respectively. Given that each long value occupies 8 bytes:

$$memory\_consumption_{pd}\ [\mathrm{B}] = 8 \cdot (|cnt| + |p_{ec}|) \tag{5}$$

Similarly to *pd*, *pc* allocates the p_{ec} array. However, the *cnt* array is replaced by a p_c array of length $|p_c| = |P_E|$ (introduced in Section 4.2.3). The memory consumption of the *pc* implementation can be expressed in bytes as follows:

$$memory\_consumption_{pc}\ [\mathrm{B}] = 8 \cdot (|p_c| + |p_{ec}|) \tag{6}$$

In Figure 23, we report the additional memory consumption introduced by the instrumentation expressed in megabytes (the y-axis is logarithmic). Since E is fixed (for a given category) and hence the length of the array that stores counters is not influenced by the workload, *dec* allocates the same amount of memory for every benchmark (344, 48, 16, and 8 bytes for categories *all*, *arithmetics*, *branches*, and *arraycopies*, respectively). Such a memory consumption is several orders of magnitude lower than the memory required by the *pc* and *pd* implementations. Because the memory consumption of *dec* is negligible, the figure reports only *pc* and *pd*.

We note that the additional memory consumption due to the instrumentation does not depend on the original memory footprint. For instance, two programs with the same $|E|$ and $|P_E|$ might consume a different amount of memory when uninstrumented, even though the space required by the instrumentation is the same. However, for comparison, Figure 23 reports also the heap size required by the uninstrumented benchmarks. To compute the heap size, we run each workload with an initial heap size of 1 MB and we let the GC ergonomics of the JVM increase the heap size until it stabilizes. We report the resulting heap size in the figure.

First, we evaluate the memory consumption of category *all*. In *pc* and *pd*, memory consumption varies among benchmarks since $|P_E|$ depends on the control flow of the instrumented program. Moreover, even for the same benchmark, $|P_E|$ (and hence memory consumption) may vary among different executions due to non-deterministic decisions taken by the VM or by the compiler. In the *pc* and *pd* implementations, most of the allocated memory is dedicated to the decoding table. In the case of *pd*, the memory consumption ranges from 7.87 MB (*scala-kmeans*) to 405.21 MB (*eclipse*), while in the case of *pc*, the memory consumption ranges from 7.17 MB (*scala-kmeans*) to 500.44 MB (*eclipse*). On average, *pd* and *pc* require 44.58 MB and 47.30 MB, respectively. The heap size required by the benchmarks ranges from 80.33 MB (*scala-kmeans*) to 10071.83 MB (*akka-uct*), and is 1056.12 MB on average. Both *pd* and *pc* require $\approx 4\%$ of the heap size on average.

We note that, for the same benchmark, *pc* typically requires more memory than *pd*, since $|P_E|$ is always greater than $|E|$, and hence $|p_c| > |cnt|$ (see Formulas 5 and 6). For certain benchmarks, such as *fj-kmeans* and *rx-scrabble*, *pd* allocates more memory than *pc*, meaning that the sets P_E obtained by the two implementations are significantly different. Similarly to what we report in

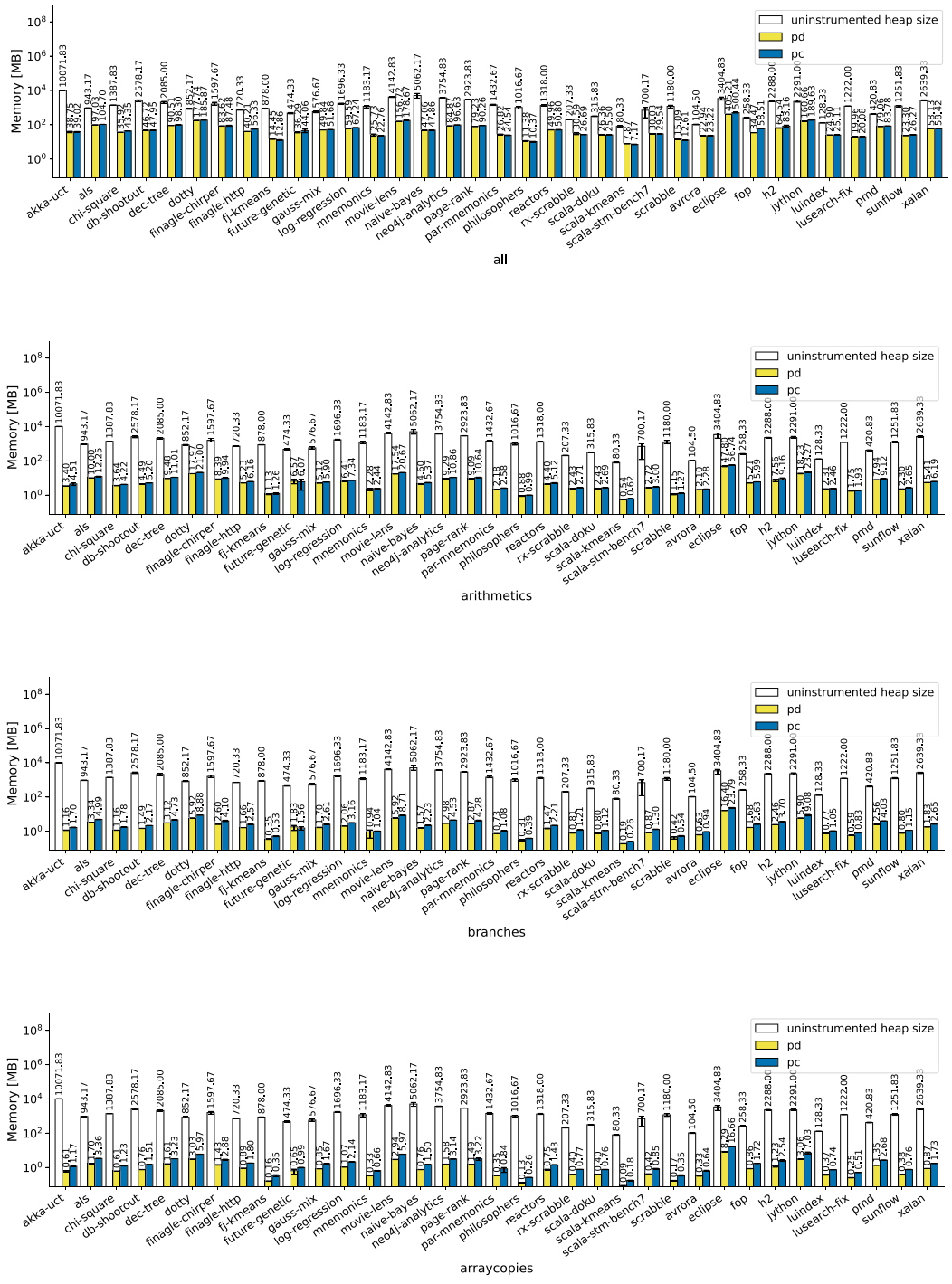


Fig. 23. Memory consumption of the *pd* and *pc* strategies, compared with the heap size required by the benchmarks.

Section 6.2 and later detail in Section 7.3, this difference in the sets P_E is caused by the fact that different profiles lead to different inlining decisions and speculative optimizations that alter the control-flow of the compilation units.

Our evaluation results on the other three evaluated categories show how memory consumption decreases together with the number of event types (i.e., $|E|$) and the number of paths that contain at least one event (i.e., p_{ec}). For this reason, the total memory consumption of the *arithmetics*, *branches*, and *arraycopies* categories is much less than the memory consumption of category *all*.[15] Moreover, similar to the other analyses reported in the section, the memory consumption decreases for *arithmetics*, and even more for *branches* and *arraycopies*. For most of the benchmarks, even when profiling all event types simultaneously, the memory required by the proposed approaches is small w.r.t. the heap size.

7 DISCUSSION

In this section, we first discuss the assumptions of our technique (Section 7.1). Then, we discuss additional potential usages of our approach (Section 7.2) and report its limitations (Section 7.3).

7.1 Assumptions

To be applicable, our technique requires a compilation pipeline where an event type of interest can be identified in isolation and where one can clearly locate a phase after which an event type cannot be detected anymore. We note that our technique is applicable to the HotSpot C2 compiler [19] and Graal. Moreover, to the best of our knowledge, it should be applicable also to Turbofan (V8) [44, 45, 107], LLVM [65], and GCC [37], for the event types defined in this paper. To use markers, a compiler should allow fixed no-op instructions, which have no side-effects and whose code-size and cycle-count estimates are set to zero.

7.2 Potential Additional Usages

A compiler developer could use our methodology to define optimizable patterns as new event types and perform exploratory analyses to quantify how frequently these patterns are executed. In this way, it is possible to determine whether it is worthwhile to implement a certain compiler phase, or whether its potential speedup is not sufficient because the pattern is rarely executed. For example, to extend the partial escape analysis phase [110] so that it stack-allocates a certain new type of an IR node, we can collect an event count that represents allocations of this node that did not escape.

Compiler developers can also employ our methodology to study how compiler phases affect event types. For instance, it is possible to compare the same event types, collected before and after a certain compiler phase, rather than before the phase that performs the lowering.

One important application area of the proposed profiler is to track performance regressions in an optimizing compiler. Rather than tracking the running time of a benchmark to detect regressions, compiler developers can track metric counts that reflect IR-internal instructions. Such metrics serve as proxies for the program running time, and they can reflect the effectiveness of a particular optimization more precisely.

Our methodology can be used by developers who want to characterize or investigate new workloads and benchmarks that are specifically designed to stress specific kinds of compiler optimizations. A good benchmark suite should have a large variety of benchmarks that exploit different optimizations and show high diversity in terms of metrics. Our approach can help to verify that the chosen workloads show the required diversity.

[15] We remark that for categories *arithmetics*, *branches*, and *arraycopies*, $|E|$ is equal to six, two, and one, respectively. For the category *all*, $|E|$ is equal to 43.

While the implementation of the intrumentation-code generation phase and the use cases shown in the article largely focus on JIT compilation, our approach can also be applied to static compilation. Concretely, our profiler can be easily ported to Native Image [119], a static ahead-of-time (AOT) snapshotting tool based on the Graal Compiler. Doing so would allow studying the differences between AOT and JIT compilation. For example, assuming to analyze the steady state (where almost no interpretation occurs), our approach can be used to compare the effect of JIT-compilation-specific speculative optimizations against AOT optimizations.

7.3 Limitations

Our approach allows one to collect event types that can be represented at the compiler-IR level. As any other compiler-IR profiling technique, our approach cannot profile event occurrences related to code that has not been compiled, or low-level hardware event types such as branch- or cache-misses. For the typical use cases presented in this article, i.e., analyzing and debugging compiler optimizations (Sections 5.3 and 5.4), this is a desirable property—a compiler developer exactly wants to profile the JIT compiled code, excluding native and interpreted code. However, for other use cases, this property may be a limitation. For example, our approach may provide insufficient information to a performance analyst that needs a complete view of the whole program execution.

Our implementation is not able to collect event occurrences that take place in C/C++ code not compiled by Graal, such as the internal implementation code of the JVM. To overcome this limitation, we could recompile such native code with a C++ compiler that implements our profiling technique. For example, as later mentioned in Section 8, LLVM provides an API to insert user-defined compiler phases [64]. This API allows implementing our technique in LLVM. While the IR of LLVM and the IR of Graal are different, one can identify, for each event type detected by our approach, the nodes in the LLVM IR that represent the same event type. If the program includes calls to native code, then the final count for an event type is the sum of the executions of all LLVM and Graal nodes that represent such an event type.

Since our implementation is not able to profile event occurrences that take place outside of code compiled by Graal, our approach cannot collect event occurrences that take place in interpreted code. However, we conducted some experiments to confirm that in the steady-state iterations of iterative workloads (i.e., programs that repeatedly perform the same computation, including benchmark suites such as Renaissance), the execution time spent in the interpreter is very little. Our results show that when executing a steady-state iteration of the Renaissance benchmarks, the time spent by the JVM in interpreted code is on average 0.04% of the total execution time, ranging from 0% to 0.46%. This indicates that the vast majority of the executed code is compiled and can be profiled with our approach. In non-iterative workloads, our approach may not collect event occurrences taking place in infrequently executed code. However, our approach could be still be employed if the program is compiled ahead-of-time.

Even though markers allow increasing profiling accuracy, markers may still yield less accurate profiles w.r.t. alternative designs, such as the event-type-set-annotations discussed in Section 3.3.2. We consider investigating alternative designs as interesting future work.

Our methodology lowers the perturbation on compiler phases by inserting markers that will later be converted to instrumentation code. Unfortunately, some level of perturbation is still present. For example, the memory overhead (Section 6.5) introduced by our strategies may alter cache locality and the instrumentation code can affect register allocation that occurs after the instrumentation-code generation phase, increasing register pressure. We note that the impact on cache behavior and register allocation does not directly affect the profiles collected using our strategies but influences other runtime metrics collected using **Hardware Performance Counters (HPCs)**, such as cache misses. Moreover, since the instrumentation code is inserted at the

end of the compilation pipeline, this code is not optimized by optimization phases and may introduce unnecessary overhead. Generally, our profiler may affect thread scheduling which in turn can affect locality. We are investigating approaches to mitigate these issues.

In JIT compilers, even though our technique lowers perturbation on compiler phases within a single compilation unit, other sources of perturbations are still present. For example, the additional compilation time required to instrument a certain method may delay the compilation of other compilation units. During this delay, code associated to these other compilation units is executed by the interpreter, which lets the VM collect additional profiles that later guide compiler decisions. Since these profiles are different w.r.t. the profiles that the VM would collect in an execution without instrumentation, the instrumentation perturbs compilation—the compiler may take different optimization decisions that lead to different execution time and memory consumption. Optimizations that leverage profiling data include method inlining and speculative optimizations. Such a perturbation is the root cause of the higher execution-time overhead of path decoding w.r.t. direct event counting and of the higher memory consumption of path decoding w.r.t. path counting related to some benchmarks, as reported in Sections 6.2 and 6.5, respectively. We are investigating techniques to reduce such perturbations.

Finally, path cutting decreases the total path-count $|P_E|$ and hence memory-consumption and compilation time, making path profiling applicable to modern JIT compilers. However, path cutting decreases also the average path-length π, leading to more frequent updates that increase the execution-time and code-size overhead associated to the instrumentation. Consider for example the path counting strategy and the execution of basic blocks <1, 2, 6, 7, 11> shown in Figure 19. Without path cutting, instrumentation code (inserted at node 11) would increment only the counter associated with path <1, 2, 6, 7, 11>. With path cutting, instrumentation code increments three counters. In particular, instrumentation code inserted at nodes 2, 7, and 11 would increment the counters associated with path <1, 2>, <6, 7>, and <11>, respectively. In Section 6, we show that on widely used benchmarks, path cutting leads to a reasonable trade-off between memory consumption and execution-time overhead.

8 RELATED WORK

Zheng et al. [125] show how compiler optimizations influence event occurrences and, due to this reason, source- and bytecode-level instrumentation cannot be used to accurately count event occurrences that may be altered by different compiler phases, such as allocated objects or method invocations. To mitigate this issue, they define an API and a technique to make the compiler aware of the inserted instrumentation code, which can be moved or removed with the corresponding event occurrence. As a result, a bytecode-level profiler that uses their API can avoid accounting event occurrences that do not take place after compilation. Unfortunately, such a technique addresses only a limited set of optimizations, i.e., method inlining, stack allocation, and lock optimizations. Moreover, it cannot track event occurrences inserted by a compiler phase. Our approach does not suffer from the same limitations.

Another limitation of the above technique is that it does not allow instrumenting event types that are available in the IR but do not have a corresponding bytecode representation. To overcome this limitation, the authors propose another approach [124] that combines bytecode- and compiler-IR instrumentation. Their framework exposes interfaces that correspond to some IR-level event types that the bytecode profiler can reference to define the instrumentation code. The framework properly inserts the instrumentation code at the correct position in the compilation pipeline. Even though their method enables the collection of IR-level event types (including memory-barriers, safepoints, and deoptimizations [126]), it may cause performance degradation due to the instrumentation code perturbing the subsequent compiler optimization phases. Our approach is less

prone to this issue thanks to the use of markers that will be converted to instrumentation code only at the end of the compilation pipeline. Furthermore, the approach of Zheng et al. [126] does not allow one to directly analyze and modify the IR to track additional event types and perform optimizations such as path profiling, in contrast to our technique.

Jikes RVM [3] offers an API [53] to count event types by inserting instrumentation phases into its compilation pipeline. Such instrumentation phases insert high-level "count event" instructions that will be later lowered into actual instrumentation code. However, unlike our approach, Jikes RVM inserts all the instrumentation phases at the end of the high tier [43], preventing one from tracking intermediate instructions at the middle of the compilation pipeline. Moreover, event counters capture the effects of high-tier optimizations but not the effects of subsequent phases that may be perturbed by "count event" instructions. While we are interested in tracking several frequent event types, Jikes RVM offers only three built-in event counters, i.e., method invocation, yieldpoint, and instruction counters. Finally, the lowering phase of event counters does not implement path profiling, possibly leading to a prohibitively high runtime overhead.

Compiler-IR instrumentation is a well known approach. However, we are not aware of techniques exposing APIs that allow users to specify event types of interest at the compiler-IR level, as our work enables. Gprof [46] uses compiler-IR instrumentation to collect execution profiles. AddressSanitizer [104] and ThreadSanitizer [105, 106] use a compiler-IR instrumentation module and a runtime library to detect memory errors and data races, respectively. In contrast to our methodology, such tools solve very specific issues and cannot be employed to extensively analyze compiler behaviour by collecting runtime metrics.

Other tools perform the instrumentation at the compiler-IR level by inserting instrumentation phases into the compilation pipeline [12, 40, 102, 115]. In particular, Preuss [81] uses an instrumentation phase to perform path profiling and monitor program executions. In contrast to our approach, such instrumentation phases do not allow users to specify and count event types of interest. For instance, Preuss' approach allows only to count path executions within single functions. Moreover, such approaches do not reduce perturbations and do not allow one to inspect compiler optimizations.

Some compilers such as GCC [36] and the LLVM compiler infrastructure [58] provide convenient APIs to insert user-defined compiler phases (called passes) [38, 64]. Such APIs ease developers in implementing our technique and analyzing other languages and runtimes without the need of modifying the internal compiler implementation.

The implementation of our profiler leverages the efficient path profiling algorithm proposed by Ball and Larus [9] to count paths and reduce the execution time overhead, enabling the collection of several metrics in a single workload execution. Ball [7] discusses a related strategy to perform efficient event counting. However, that strategy maintains basic-block and/or edge profiles [8] instead of path profiles and is not suitable for collecting a large set of different event types, unlike ours. Ammons et al. [4] extend the efficient path-profiling algorithm to exploit **hardware performance counters (HPCs)** with flow and context-sensitive profiling. Our technique can be easily extended to report HPC metrics and context for measurements, enabling more in-depth compiler optimization analyses.

Bond and McKinley [14] combine instrumentation and sampling to perform efficient path and edge profiling in JIT compilers. In particular, they implement their technique in Jikes RVM, using instrumentation to track dynamic path executions and sampling to mitigate the path counting overhead. Even though their sampling technique lowers execution-time overhead, efficient instrumentation leads to more accurate profiles. Moreover, in state-of-the-art JIT compilers such as Graal, their technique may not be applicable due to the large number of paths and hence require a modification of the algorithm.

Eyerman et al. [33] study the effect of compiler optimizations on superscalar processors by performing interval analysis [32, 34, 55, 113] and collecting clock cycles associated to low-level event types, such as branch and cache misses. Later, similarly to our approach, they enable and disable compiler optimization phases to study their effects on such metrics. However, the metrics obtained by their approach represent only the final code emitted by the compiler and do not provide accurate information on the specific event types and how different compiler optimization phases influence each other.

Replay JIT compilation [41, 72] is an approach that aims at improving JIT-compiler debuggability. Using this approach, while compiling a method, the compiler stores all the input code it processes in a log file. Later, the developer can re-compile the code using the log file, also enabling debugging options to print diagnostic data. On the one hand, replay compilation reduces non-determinism and improves experiment repeatability. However, in contrast to our approach, it does not provide execution metrics that can be analyzed to understand performance speedup or slowdown.

9 CONCLUSION

In this section, we first give our concluding remarks (Section 9.1) and then we discuss some ideas for future work (Section 9.2).

9.1 Concluding Remarks

We present a novel technique for profiling program event types, which lowers perturbation on compiler optimizations. Our approach marks locations of interest in the compiler's IR at the phases in the compilation pipeline after which no subsequent optimizations affect the corresponding event types. For these locations, instrumentation code will be inserted only at the end of the compilation pipeline, with the aim of reducing the chances of affecting compiler optimizations. Our technique can efficiently and accurately profile bytecode-level event types that map to bytecode instructions and compiler-internal event types that do not explicitly map to source code or bytecode instructions such as guards or safepoints.

We provide a generic API for customizing the set of event types of interest. Then, we implement our methodology in a counting profiler for Graal, which is able to track 43 event types simultaneously. We propose two path-profiling strategies that significantly reduce the instrumentation overhead. The first strategy, called path decoding, is suitable for on-line profiling while the second strategy, called path counting, further reduces profiling overhead when real-time event count updates are not a requirement. Moreover, we propose a modification to the standard path-profiling algorithm (called path cutting) to make path profiling applicable to modern JIT compilers. We show that our technique helps compiler developers analyze and debug compiler behaviour—we describe two use cases in which we employed the profiler to identify the reasons behind a performance speedup, and the causes of an unexpected performance slowdown introduced by a compiler optimization.

We evaluate our approach, confirming that the two proposed path-profiling strategies with path cutting help reduce the instrumentation overhead w.r.t. a naive approach called direct event counting. Our results show that path counting and path decoding introduce an average execution-time overhead of 1.18× and 2.11×, respectively, while direct event counting leads to an average execution-time overhead of 3.55×. Moreover, we show that path counting leads to a lower compilation-time and code-size overhead w.r.t. direct event counting.

9.2 Future Work

The two proposed implementations (i.e., path decoding and path counting) described in this article do not allow collecting runtime values associated to the profiled event types. For example, the implementations allow counting the number of method invocations (invokes) but not to collect,

dump, and analyze the runtime values of their arguments, because they are expressed in different IR nodes than those representing the event type. As part of our future work, we plan to extend the implementations to support this feature. This would allow one to target more use cases, such as the collection of program-execution traces.

To lower the difficulty of finding the target phase after which inserting markers for a given event type, we plan to model the compilation pipeline and use metadata to automatically infer the target phase and insert the instrumentation phase there, without the developer intervention.

Finally, the proposed profiler can bolster future research of compiler optimizations, assist the analysis of programs and benchmarks workloads, or serve as a performance-regression-detection tool.

REFERENCES

[1] 2023. Akka Website. https://akka.io/.

[2] 2023. Twitter Finagle. https://twitter.github.io/finagle/guide/Quickstart.html.

[3] Bowen Alpern, Steve Augart, Stephen Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn McKinley, Mark Mergen, Eliot Moss, Ton Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44 (2005), 399–418.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling *(PLDI'97)*. 85–96.

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining *(DYNAMO'00)*. 52–64.

[6] Phil Bagwell. 2001. *Ideal Hash Trees*. (2001). Technical report, October 2001. http://infoscience.epfl.ch/record/64398.

[7] Thomas Ball. 1994. Efficiently counting program events with support for on-line queries. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1399–1410.

[8] Thomas Ball and James Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (1994), 1319–1360.

[9] Thomas Ball and James Larus. 1996. Efficient path profiling *(MICRO'96)*. 46–57.

[10] Matteo Basso, Andrea Rosà, Luca Omini, and Walter Binder. 2023. Java vector API: Benchmarking and performance analysis *(CC'23)*. 1–12.

[11] Matteo Basso, Eduardo Rosales, Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. Accurate fork-join profiling on the Java virtual machine *(Euro-Par'22)*. 35–50.

[12] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. 2016. *XRay: A Function Call Tracing System*. Technical Report. 1–8 pages.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis *(OOPSLA'06)*. 169–190.

[14] Michael David Bond and Kathryn McKinley. 2005. Continuous path and edge profiling *(MICRO'05)*. 11–140.

[15] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring garbage collection with dynamic generations for HotSpot big data applications *(ISMM'17)*. 2–13.

[16] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Jose Serrano, Vugranam Sreedhar, Harini Srinivasan, and John Whaley. 1999. The Jalapeño dynamic optimizing compiler for Java *(JAVA'99)*. 129–141.

[17] William Chen, Pohua Chang, Thomas Conte, and Wen-Mei Hwu. 1993. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput.* 42, 9 (1993), 1045–1057.

[18] Cliff Click. 1995. Global code motion/global value numbering *(PLDI'95)*. 246–257.

[19] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation *(IR'95)*. 35–49.

[20] Cliff Click and John Rose. 2002. Fast subtype checking in the HotSpot JVM *(JGI'02)*. 96–107.

[21] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Frank Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13 (1991), 451–490.

[22] DaCapo Project. 2018. The DaCapo Benchmark Suite. http://dacapobench.sourceforge.net/.

[23] Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. 2015. Techniques and applications for guest-language safepoints *(ICOOOLPS'15)*. 8:1–8:10.

[24] David Detlefs and Ole Agesen. 1999. Inlining of virtual methods *(ECOOP'99)*. 258–278.

[25] David Dice. 2001. Implementing fast Java monitors with relaxed-locks *(USENIX JVM'01)*. 79–90.
[26] Dave Dice. 2006. Biased Locking in HotSpot. https://blogs.oracle.com/dave/biased-locking-in-hotspot.
[27] David Dice, Mark Moir, and William Scherer III. 2003. Quickly reacquirable locks (patent). US7814488B1 (2003), 1–19. https://patents.google.com/patent/US7814488B1/en.
[28] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation *(APPLC'13)*. 1–9.
[29] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler *(VMIL'13)*. 1–10.
[30] Gilles Marie Duboscq. 2016. Combining Speculative Optimizations with Flexible Scheduling of Side-Effects. https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-9708.
[31] Alexis Engelke and Martin Schulz. 2020. Instrew: Leveraging LLVM for high performance dynamic binary instrumentation *(VEE'20)*. 172–184.
[32] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James Smith. 2006. A performance counter architecture for computing accurate CPI components *(ASPLOS XII)*. 175–184.
[33] Stijn Eyerman, Lieven Eeckhout, and James Smith. 2008. Studying compiler optimizations on superscalar processors through interval analysis *(HiPEAC'08')*. 114–129.
[34] Stijn Eyerman, James Smith, and Lieven Eeckhout. 2006. Characterizing the branch miss prediction penalty *(ISPASS'06)*. 48–58.
[35] R. A. Fisher. 1925. *Statistical Methods for Research Workers*. Oliver & Boyd (Edinburgh).
[36] Free Software Foundation. 2021. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.
[37] Free Software Foundation. 2021. Passes and Files of the Compiler. https://gcc.gnu.org/onlinedocs/gccint/Passes.html.
[38] Free Software Foundation. 2021. Plugin Pass (GNU Compiler Collection (GCC) Internals). https://gcc.gnu.org/onlinedocs/gccint/Plugins-pass.html#Plugins-pass.
[39] Alex Garthwaite, David Dice, and Derek White. 2005. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification *(VEE'05)*. 24–34.
[40] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios Nikolopoulos, and Martin Schulz. 2017. REFINE: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed *(SC'17)*. 29:1–29:14.
[41] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java performance evaluation through rigorous replay compilation *(OOPSLA'08)*. 1–18.
[42] GitHub. 2019. Graal Issue 1541. https://github.com/oracle/graal/issues/1541.
[43] GitHub. 2021. JikesRVM OptimizationPlanner. https://github.com/JikesRVM/JikesRVM/blob/38b21f5a663016dbf43771cb2d231f74db0a01c6/rvm/src/org/jikesrvm/compilers/opt/driver/OptimizationPlanner.java#L285.
[44] Google. 2021. An Overview of the TurboFan Compiler. https://docs.google.com/presentation/d/1H1lLsbclvzyOF3IUR05ZUaZcqDxo7_-8f4yJoxdMooU/edit#slide=id.g18ceb14729_0_135.
[45] Google. 2022. TurboFan. https://v8.dev/docs/turbofan.
[46] Susan Lois Graham, Peter Bernard Kessler, and Marshall Kirk Mckusick. 1982. Gprof: A call graph execution profiler *(SIGPLAN'82)*. 120–126.
[47] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. 2004. Java just-in-time compiler and virtual machine improvements for server and middleware applications *(VM'04)*. 1–12.
[48] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A benchmark for software transactional memory *(EuroSys'07)*. 315–324.
[49] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3 (2020), 49:1–49:29.
[50] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization *(PLDI'92)*. 32–43.
[51] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback *(PLDI'94)*. 326–336.
[52] Urs Hölzle. 1993. A fast write barrier for generational garbage collectors. *OOPSLA/ECOOP'93 Workshop on Garbage Collection in Object-Oriented Systems*. 1–6.
[53] Jikes™ RVM project. 2021. Profiling Applications with Jikes RVM. https://www.jikesrvm.org/UserGuide/ProfilingApplicationsWithJikesRVM/index.html#x10-990008.
[54] John Rose. 2011. Java Enhancement Proposal 243: Java-Level JVM Compiler Interface. https://openjdk.java.net/jeps/243.
[55] Tejas Karkhanis and James Smith. 2004. A first-order superscalar processor model *(ISCA'04)*. 338.
[56] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-driven conditional instruction prefetching with coalescing *(MICRO'20)*. 146–159.

[57] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot client compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1 (2008), 7:1–7:32.
[58] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation *(CGO'04)*. 75–87.
[59] Doug Lea. 2000. A Java fork/join framework. In *Java Grande*. 36–43.
[60] Jan-Patrick Lehr. 2016. Counting performance: Hardware performance counter and compiler instrumentation *(IWOMP'16)*. 2187–2198.
[61] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations *(ManLang'18)*. 1–13.
[62] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A cost model for a graph-based intermediate-representation in a dynamic compiler *(VMIL'18)*. 26–35.
[63] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): Code duplication to enable compiler optimizations *(CGO'18)*. 126–137.
[64] LLVM Project. 2018. Writing an LLVM Pass. https://releases.llvm.org/5.0.2/docs/WritingAnLLVMPass.html.
[65] LLVM Project. 2021. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html#transform-passes.
[66] Allen Malony, Daniel Reed, and Harry Wijshoff. 1992. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems* 3, 4 (1992), 433–450.
[67] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at your own risk: The Java unsafe API in the wild *(OOPSLA'15)*. 695–710.
[68] Raphael Mosaner. 2020. Machine learning to ease understanding of data driven compiler optimizations *(SPLASH Companion 2020)*. 4–6.
[69] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. 2007. Understanding measurement perturbation in trace-based data *(IPDPS'07)*. 1–6.
[70] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth Kent. 2020. Insights into WebAssembly: Compilation performance and shared code caching in node.Js *(CASCON'20)*. 163–172.
[71] Martin Odersky and Adriaan Moors. 2009. Fighting bit rot with types (experience report: Scala collections) *(LIPIcs)*. 427–451.
[72] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. 2006. Replay compilation: Improving debuggability of a just-in-time compiler *(OOPSLA'06)*. 241–252.
[73] Oracle. 2022. GraalVM Repository at GitHub. https://github.com/oracle/graal.
[74] Oracle. 2022. Ideal Graph Visualizer. https://docs.oracle.com/en/graalvm/enterprise/20/docs/tools/igv/.
[75] Oracle. 2022. Java Platform, Standard Edition Java Flight Recorder Runtime Guide. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/toc.htm.
[76] Oracle. 2022. JVMCI JDK 8 Repository at GitHub. https://github.com/usi-dag/graal-jvmci-8.
[77] Oracle. 2022. Oracle Developer Studio. https://www.oracle.com/tools/developerstudio/.
[78] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot server compiler *(JVM'01)*. 1–13.
[79] Orion Papadakis, Foivos Zakkak, Nikos Foutris, and Christos Kotselidis. 2020. You can't hide you can't run: A performance assessment of managed applications on a NUMA machine *(MPLR'20)*. 80–88.
[80] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.* 12, 5 (2002), 393–434.
[81] Adam Preuss. 2021. Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure. https://llvm.org/pubs/2010-12-Preuss-PathProfiling.pdf.
[82] Aleksandar Prokopec. 2016. Pluggable scheduling for the reactor programming model *(AGERE'16)*. 41–50.
[83] Aleksandar Prokopec. 2017. Analysis of concurrent lock-free hash tries with constant-time operations. *ArXiv e-prints* (2017). arXiv:1712.09636.
[84] Aleksandar Prokopec. 2017. Encoding the building blocks of communication *(Onward! 2017)*. 104–118.
[85] Aleksandar Prokopec. 2018. Cache-tries: Concurrent lock-free hash tries with constant-time operations *(PPoPP'18)*. 137–151.
[86] Aleksandar Prokopec. 2018. Efficient lock-free removing and compaction for the cache-trie data structure *(Euro-Par'18)*. 575–589.
[87] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2013. Lock-free resizeable concurrent tries *(LCPC'13)*. 156–170.
[88] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A generic parallel collection framework *(Euro-Par'11)*. 136–147.

[89] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots *(PPoPP'12)*. 151–160.
[90] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An optimization-driven incremental inline substitution algorithm for just-in-time compilers *(CGO'19)*. 164–179.
[91] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making collection operations optimal with aggressive JIT compilation *(SCALA'17)*. 29–40.
[92] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, channels, and event streams for composable distributed programming *(Onward!'15)*. 171–182.
[93] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking suite for parallel applications on the JVM *(PLDI'19)*. 31–47.
[94] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: A modern benchmark suite for parallel applications on the JVM *(SPLASH Companion'19)*. 11–12.
[95] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. On evaluating the renaissance benchmarking suite: Variety, performance, and complexity. *CoRR* abs/1903.10267 (2019).
[96] Andrea Rosà and Walter Binder. 2018. Optimizing type-specific instrumentation on the JVM with reflective supertype information. *Journal of Visual Languages & Computing* 49 (2018), 29–45.
[97] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2017. Accurate reification of complete supertype information for dynamic analysis on the JVM *(GPCE'17)*. 1–13.
[98] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2018. Analyzing and optimizing task granularity on the JVM *(CGO'18)*. 27–37.
[99] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2019. Analysis and optimization of task granularity on the Java virtual machine. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 19 (2019), 47 pages.
[100] Eduardo Rosales, Matteo Basso, Andrea Rosà, and Walter Binder. 2023. Profiling and optimizing Java streams. *The Art, Science, and Engineering of Programming* 7, 3 (2023), 1–43.
[101] Kenneth Russell and David Detlefs. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing *(OOPSLA'06)*. 263–272.
[102] Tao Schardl, Tyler Denniston, Damon Doucet, Bradley Kuszmaul, I-Ting Angelina Lee, and Charles Leiserson. 2017. The CSI framework for compiler-inserted program instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2 (2017), 43:1–43:25.
[103] Robert William Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM* 20, 9 (1977), 647–654.
[104] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker *(USENIX ATC'12)*. 1–28.
[105] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice *(WBIA'09)*. 62–71.
[106] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic race detection with LLVM compiler *(RV'11)*. 110–114.
[107] Jaroslav Sevcik. 2016. Turbofan IR. https://docs.google.com/presentation/d/1Z9iIHojKDrXvZ27gRX51UxHD-bKf1QcPzSijntpMJBM/edit#slide=id.p.
[108] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the high road to a low level. *ACM Trans. Archit. Code Optim.* 12, 2 (2015), 20:1–20:25.
[109] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An experimental study of the influence of dynamic compiler optimizations on scala performance *(SCALA'13)*. 9:1–9:8.
[110] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java *(CGO'14)*. 165–174.
[111] Steve Blackburn. 2018. DaCapo Issue 68. http://sf.net/p/dacapobench/bugs/68/.
[112] Steve Blackburn. 2020. DaCapo Issue 70. http://sf.net/p/dacapobench/bugs/70/.
[113] Tarek M. Taha and Scott Wills. 2008. An instruction throughput model of superscalar processors. *IEEE Trans. Comput.* 57, 3 (2008), 389–403.
[114] Nathan Tallent, John Mellor-Crummey, and Michael Fagan. 2009. Binary analysis for measurement and attribution of program performance. *SIGPLAN Not.* 44, 6 (2009), 441–452.
[115] Ronny Tschüter, Johannes Ziegenbalg, Bert Wesarg, Matthias Weber, Christian Herold, Sebastian Döbel, and Ronny Brendel. 2017. An LLVM instrumentation plug-in for score-P *(LLVM-HPC'17)*. 2:1–2:8.

[116] David Ungar. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes* 9, 3 (1984), 157–167.
[117] Mark Wegman and Frank Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210.
[118] Matthew Edwin Weingarten, Theodoros Theodoridis, and Aleksandar Prokopec. 2022. Inlining-benefit prediction with interprocedural partial escape analysis *(VMIL'22)*. 13–24.
[119] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter Bernard Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29.
[120] Thomas Würthinger. 2014. Graal and Truffle: Modularity and separation of concerns as cornerstones for building a multipurpose runtime *(MODULARITY'14)*. 3–4.
[121] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2008. Visualization of program dependence graphs *(CC'08)*. 193–196.
[122] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[123] Peng Zhao and José Nelson Amaral. 2004. To inline or not to inline? Enhanced inlining decisions *(LCPC'04)*. 405–419.
[124] Yudi Zheng. 2017. *Observable Dynamic Compilation* (Doctoral dissertation). 1–111.
[125] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate profiling in the presence of dynamic compilation *(OOPSLA'15)*. 433–450.
[126] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2017. An empirical study on deoptimization in the Graal compiler. In *ECOOP'17 (LIPIcs, Vol. 74)*. 30:1–30:30.
[127] Matija Šipek, Dino Muharemagić, Branko Mihaljević, and Aleksander Radovan. 2020. Enhancing performance of cloud-based software applications with GraalVM and Quarkus *(MIPRO'20)*. 1746–1751.

Received 17 January 2022; revised 29 November 2022; accepted 8 March 2023