Accurate Profiling in the Presence of Dynamic Compilation

Yudi Zheng* Lubomír Bulej*† Walter Binder*

* Università della Svizzera italiana (USI), Faculty of Informatics, Switzerland
† Charles University, Faculty of Mathematics and Physics, Czech Republic
firstname.lastname@usi.ch

Abstract

Many profilers based on bytecode instrumentation yield wrong results in the presence of an optimizing dynamic compiler, either due to not being aware of optimizations such as stack allocation and method inlining, or due to the inserted code disrupting such optimizations. To avoid such perturbations, we present a novel technique to make any profiler implemented at the bytecode level aware of optimizations performed by the dynamic compiler. We implement our approach in a state-of-the-art Java virtual machine and demonstrate its significance with concrete profilers. We quantify the impact of escape analysis on allocation profiling, object lifetime analysis, and the impact of method inlining on callsite profiling. We illustrate how our approach enables new kinds of profilers, such as a profiler for non-inlined callsites, and a testing framework for locating performance bugs in dynamic compiler implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Compilers, Optimization; D.2.5 [*Software Engineering*]: Testing and Debugging — Testing tools

General Terms Algorithms, Languages, Measurement, Performance

Keywords Dynamic compilers; profiling; bytecode instrumentation

1. Introduction

Many programming languages are implemented on top of a managed runtime system, such as the Java Virtual Machine (JVM) or the .NET CLR, featuring an optimizing dynamic (just-in-time) compiler. Programs written in those languages are first interpreted (or compiled by a baseline compiler), whereas frequently executed methods are later compiled by the optimizing dynamic compiler. State-of-the-art dynamic compilers, such as the optimizing compiler in the Jikes RVM [2, 9] or Graal [28], apply online feedback-directed optimizations [4, 36] to the program according to profiling information gathered during program execution. In long-running programs, most execution time is typically spent in highly optimized compiled code.

Common optimizations performed by dynamic compilers include method inlining [3] and stack allocation of objects based on (partial) escape analysis [10, 37], amongst others. Such optimizations result in compiled machine code that does not perform certain operations present at the bytecode level. In the case of inlining, method invocations are removed. In the case of stack allocation, heap allocations are removed and pressure on the garbage collector is reduced.

Many profiling tools are implemented using bytecode instrumentation techniques, inserting profiling code into programs at the bytecode level. However, because dynamic compilation is transparent to the instrumented program, a profiler based on bytecode instrumentation techniques is not aware of the optimizations performed by the dynamic compiler. Prevailing profilers based on bytecode instrumentation suffer from two serious limitations: (1) *over-profiling* of code that is optimized (and in the extreme case completely removed) by the dynamic compiler, and (2) *perturbation* of the compiler optimizations due to the inserted instrumentation code.

On the one hand, the dynamic compiler is not aware of the bytecode instrumentation. It will optimize the instrumented methods; it may (re)move instructions that are being profiled, but it will not (re)move the associated profiling code. Hence, the profile will not exactly correspond to the execution of the optimized program. In the case of a method invocation profiler, the profile may include spurious method invocations. In the case of an object allocation profiler, the profile may show more heap allocations than took place.

On the other hand, code instrumented by a profiler may affect the optimization decisions of the dynamic compiler, i.e., the presence of profiling makes the profiled program behave differently. For example, the increased size of an instrumented method may prevent its inlining into callers, because method body sizes are used in typical method inlining heuristics.

OOPSLA '15, October 25–30, 2015, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/2814270.2814281

As another example, passing the reference of an application object to the profiling logic makes the object escape and therefore forces heap allocation of the object, independently of whether the object escapes in the original method or not. In general, when profiling a program, the *observer effect* [27], i.e., perturbations of low-level dynamic metrics (such as hardware or operating system performance counters), cannot be avoided. However, it is possible to avoid perturbations of dynamic optimizations by making the dynamic compiler aware of the inserted profiling code.

As a consequence of the aforementioned two limitations, profiles produced by tools based on bytecode instrumentation are often misleading [20, 27]. This problem has been witnessed by many researchers; for example, tools for object lifetime analysis [17, 31] and for modeling garbage collection behavior based on program traces [24] suffer from significant inaccuracies because they fail to capture the impact of (partial) escape analysis and stack allocation. Moreover, using bytecode instrumentation, it is generally impossible to profile the effectiveness of dynamic compiler optimizations.

In this paper, we introduce a novel technique to make profilers implemented with bytecode instrumentation techniques aware of the optimization decisions of the dynamic compiler, and to make the dynamic compiler aware of inserted profiling code. Our technique enables profilers which collect dynamic metrics that (1) correspond to an execution of the base program without profiling (w.r.t. the applied compiler optimizations), and (2) properly reflect the impact of dynamic compiler optimizations. The contributions of this paper are fourfold.

1. We present a new technique to make profilers aware of dynamic compiler optimizations and to avoid perturbation of the optimizations (Section 3). We implement our technique in Oracle's Graal compiler [28] and provide a set of query intrinsics for retrieving the optimization decisions within inserted profiling code.
2. We present profilers to explore the impact of (partial) escape analysis and stack allocation on heap usage (Section 4.1) and object lifetime (Section 4.2), and to explore the impact of method inlining on callsite profiling (Section 4.3), demonstrating that our approach helps improve the accuracy of existing bytecode-instrumentation-based tools.
3. We present tools to identify inlining opportunities (Section 5.1), and to study the impact of method inlining considering varying levels of calling context (Section 5.2), demonstrating that our approach enables new tools that can help further improve the optimizations performed by dynamic compilers.
4. We introduce a new framework for testing the results of dynamic compiler optimizations at runtime, which helps, e.g., to discover optimizations that become ineffective due to unwanted interferences among different optimization phases (Section 5.3). Our framework has already helped the developers of Graal to locate and fix performance bugs in their compiler.

Before presenting our approach in Section 3, we discuss the relevant background on dynamic compilation, intermediate code representation, and instrumentation in Section 2. There is no separate related-work section; instead, the discussion of related work is distributed throughout the other sections to improve the flow of the paper. The case studies presented in Sections 4 and 5 serve primarily to demonstrate the applicability and benefits of our approach in diverse scenarios. An assessment of the strengths and limitations of our approach can be found in Section 6.

2. Background

Dynamic compilation is adopted in many programming language implementations [1]. In comparison with static compilation, it has two major advantages. First, dynamic compilation allows for a platform-independent code representation (e.g., Java bytecode). Second, in some cases, the code generated by a dynamic compiler may outperform statically compiled code, because the dynamic compiler can optimize aggressively by making certain assumptions on the future program behavior based on profiling information collected in the preceding execution. In the unfortunate case where such assumptions fail, the managed runtime system switches back to executing unoptimized code; this is called de-optimization [18]. Subsequently, the code may get optimized and compiled again.

Optimizing dynamic compilers perform optimizations based on static code analysis, such as *Escape Analysis (EA)* [7, 8, 10, 21, 22]. EA analyzes where references to new objects flow, in order to determine their dynamic scope. If the scope of an allocated object is method-local, the compiler may allocate it on the stack or apply scalar replacement by "breaking up" the object. *Partial Escape Analysis (PEA)* [37] extends the traditional escape analysis by checking whether an object escapes for individual branches. PEA allows for postponing heap allocation until an object escapes. Consequently, heap allocation may occur in a different location than in the unoptimized program. For execution paths where an object does not escape, heap allocation can be avoided.

Optimizing dynamic compilers perform online feedback-directed optimizations [4], such as profile-directed inlining of virtual call sites [2, 4, 12, 16, 19]. SELF-93 [19] introduces *type feedback*, which requires the managed runtime system to profile the receiver type of virtual call sites. The collected receiver type profile is later applied in the dynamic compiler to perform guarded inlining. While inlining such a call site, the dynamic compiler may either preserve the expensive dynamic dispatch for the minority of receiver types, or inline all possible call targets with the assumption that the profiled receiver types cover all potential use scenarios. If

an unexpected receiver type is encountered at runtime, the compiled code is de-optimized and will be profiled again for receiver types.

Intermediate Representations (IRs) are used by many modern compilers, as IR graphs are well suited for implementing compiler optimizations as graph transformations, before emitting machine code [1]. In a dynamic compiler, different levels of IR may be applied for different kinds of optimizations. For instance, Jikes RVM's optimizing compiler applies a high-level IR, a low-level IR, and a machine-specific IR, for performing general optimizations, managed runtime-specific optimizations, and machine-specific optimizations, respectively.

Additional forms of IR are used for speeding up local and global compiler optimizations. For instance, the *Program Dependence Graph (PDG)* combines both control-flow and data dependencies to express the program semantics [14]. Click [11] extended PDG by introducing nodes that are not necessarily fixed at a specific position in the control flow. The dynamic compiler must ensure a valid *schedule* for the IR graph (i.e., a serialization of the graph). The IR graph must not contain any data dependency edge where the *to* node is not reachable from the *from* node. Click's graph representation uses *Phi* nodes to represent ϕ-functions. These nodes have multiple input data values and output a single selected data value according to the control-flow edge.

Instrumentation is commonly used to observe the runtime behavior of a program. In a managed runtime system that applies feedback-directed optimization, the dynamic compiler may automatically instrument the compiled code, in order to collect profiling information for subsequent optimization; e.g., Jikes RVM features such an optimization system [2]. Moreover, instrumentation is widely used for implementing dynamic analyses such as for tracing, debugging, or profiling. Typically, the inserted instrumentation code emits some events, which may be simply dumped or consumed by an analysis at runtime.

Instrumentations for dynamic program analysis typically target either a pre-compilation program representation (i.e., source code or bytecode) or the post-compilation program representation (i.e., machine code). On the one hand, instrumenting the former representation often impairs accuracy of an analysis [20]. On the other hand, instrumenting the latter representation makes it difficult to map low-level events (e.g., memory access) to higher-level events at the level of the used programming language (e.g., field access on an instance of a particular type). One possible solution is to perform instrumentation directly within the dynamic compiler. For instance, Jikes RVM's optimizing compiler allows for the integration of additional compiler phases dedicated to specific instrumentation tasks [5]. The drawback of such a solution is that it requires deep knowledge of the dynamic compiler's implementation and of the IRs it uses. Furthermore, inserted instrumentation code may still perturb the subsequent compiler phases.

3. Approach

The aforementioned problems with over-profiling and perturbation of optimizations are due to the inability of the dynamic compiler to distinguish between the inserted profiling/analysis code and the base program code, and due to the inability of the inserted code to adapt to the optimizations performed by the dynamic compiler.

The key idea of our approach is therefore to make the compiler aware of the two kinds of code, and treat them differently. For the base program code, the goal is to let the dynamic compiler process it in the usual fashion, making optimization decisions and performing optimizations as if the inserted code was not there. For the inserted code, the goal is to preserve its purpose and semantics by adapting it in response to the optimizations performed by the dynamic compiler on the base program code.

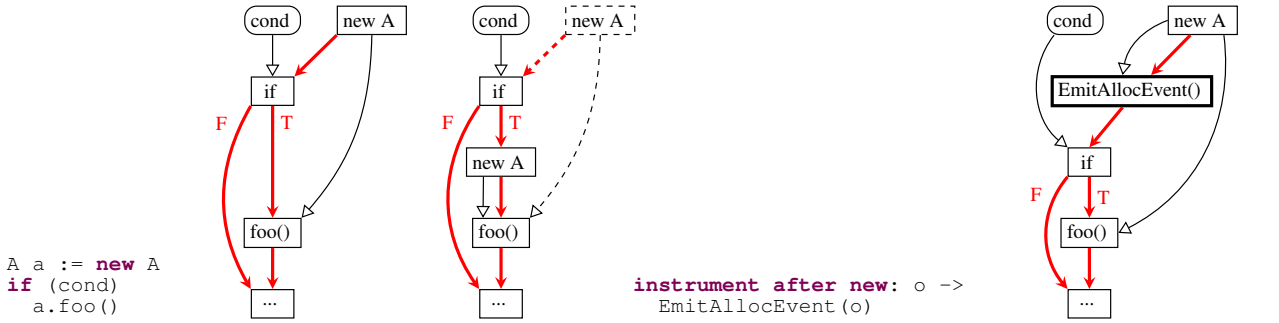
In this section we present our approach in detail. We start with an example illustrating how instrumentation perturbs an allocation optimization (Section 3.1), followed by a high-level overview of our approach (Section 3.2). It comprises several steps which we then present in detail (Sections 3.3–3.6), progressively amending the running example to illustrate the effects of our approach.

3.1 Running Example

Consider the snippet of pseudo code in Figure 1a. The code allocates an instance of class A and then, only if a condition evaluates to true, invokes foo() on the newly allocated object. To compile this code, the dynamic compiler first builds a high-level IR for the code, represented by the graph shown in Figure 1b. Before lowering the high-level IR to machine-code representation, the compiler performs optimizations on the IR, possibly reordering the code (while preserving its semantics). For example, if PEA determines that the allocation only escapes in the then-branch of the conditional, the compiler may move the allocation there, as illustrated in Figure 1c.

If we instrument the program code to trace object allocations using the pseudo-code shown in Figure 1d, every allocation will be followed by an invocation of the EmitAllocEvent() method with the newly allocated object as an argument. The corresponding IR is shown in Figure 1e. When the compiler attempts to optimize the instrumented program, it will determine that the newly allocated object always escapes into the event-emitting method (assuming it is not inlined), which will cause the object to be always allocated on the heap, even if the conditional in a particular method invocation evaluates to false. The inserted instrumentation code thus perturbs an optimization the compiler would otherwise perform on the uninstrumented program.

To avoid perturbation, we would want the compiler to perform the optimization as if the program was not instrumented.



(a) Base program **(b)** Base program IR graph **(c)** After PEA **(d)** Instrumentation code **(e)** IR graph with instrumentation

Figure 1: Instrumentation intercepting object allocations perturbing PEA due to dependency on the allocated object. In the IR graphs, rectangles represent fixed IR nodes, rounded rectangles represent floating IR nodes, thick red edges represent control flow, and thin black edges with hollow arrows represent data flow. Eliminated IR elements (nodes, control-flow edges, data-flow edges) are drawn using dashed lines.

To preserve the intent of the instrumentation, we would also want the EmitAllocEvent() method to follow the movement of the allocation into the then-branch of the conditional.

3.2 Algorithm Overview

Our approach has been formulated for a method-based dynamic compiler using a graph-based IR in the Static Single Assignment (SSA) form, with optimizations implemented as IR graph transformations.

When the dynamic compiler builds the IR of the method being compiled, we identify the boundaries between the base program code and the inserted code, and unlink the inserted code from the base program IR, creating inserted code subgraphs (ICGs) associated with base program nodes. We then let the dynamic compiler work on the base program IR while tracking the operations it performs on the IR graph nodes. If the compiler performs an operation on a node with an associated ICG, we perform a *reconciling operation* on the corresponding ICG to preserve its semantics throughout the transformations performed by the compiler. When the compiler finishes optimizing the base program IR, we splice the ICGs back into the base program IR—before it is lowered to machine-code level.

To ensure that the semantics of the base program is not changed by the inserted code, the ICGs must satisfy the following properties:

1. An ICG must have exactly one entry and exactly one exit.
2. An ICG must have exactly one predecessor[1] and exactly one successor before being extracted from the IR.
3. An ICG must not have any outgoing data-flow edges into the base program IR, i.e., the base program code must not depend on any values produced within an ICG.

[1] This may require inserting a dummy "start" node at the beginning of each method being compiled, and dummy "merge" nodes at control flow merge points. Modern compilers such as Graal do this automatically.

There may be data-flow edges originating in the base program, caused by the inserted code inspecting the base program state. While these are legal, they would normally anchor the IR nodes belonging to the inserted code to a particular location in the base program IR, effectively preventing optimizations involving code motion. To avoid this perturbation, we consider all data-flow edges originating in the base-program and targeting ICGs to be *weak data-flow edges*. These edges will be ignored by the dynamic compiler working on the base program IR, but taken into account when performing the reconciling operations on the ICGs. After splicing the ICGs back into the base program IR, the weak data-flow edges will resume their normal semantics. We now review the individual steps of our approach.

3.3 Extracting ICGs

To distinguish between the base program code and the inserted code, we rely on explicit marking of the boundaries of the inserted code. This is achieved by enclosing the inserted code between invocations of a pair of methods from the delimitation API shown in Table 1. Invocations of these methods can be recognized at the IR level, and consequently used to identify the ICG boundaries.

Depending on the base program behavior the inserted code aims to intercept, an ICG can be associated either with the predecessor or the successor base program node, or anchor to its original location in the control flow graph (CFG). Because the relative position of an ICG with respect to the base program nodes cannot be determined automatically, this information needs to be made explicit in form of an argument to the invocation of the instrumentationBegin() method. When HERE is passed as the argument, meaning that the inserted code is anchored to its original location, we create a placeholder node inserted in place of the ICG, and associate the ICG with the placeholder. To avoid any perturbation caused by the placeholders, the dynamic compiler is modified to disregard

Method	Description
instrumentationBegin	Marks the beginning of a block of inserted code. It requires an argument indicating how to determine the position of the instrumentation in the control flow graph with respect to the base program nodes. The supported values are PRED, SUCC, and HERE. The first two values indicate that the position of the instrumentation is relative either to the predecessor or to the successor base program node in the control flow graph. The last value indicates that the position of the instrumentation in the control flow graph is fixed, i.e., it does not depend on any base-program node.
instrumentationEnd	Marks the end of a block of inserted code.

Table 1: Delimitation API methods for explicit marking of inserted profiling code.

An IR graph is a tuple $\langle N, C, D\rangle$, where:

- N denotes the set of IR nodes in the base program IR graph. Initially, it also contains the IR nodes of the inserted code.
- C denotes the set of control-flow edges in the base program IR graph. Initially, it also contains the control-flow edges involving the inserted code.
- D denotes the set of data-flow edges in the base program IR graph. Initially, it also contains the data-flow edges involving the inserted code.

An ICG is a tuple $\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle$, where:

- a_I denotes the base program node the ICG is associated with.
- p_I denotes the node representing the constant argument passed to instrumentationBegin.
- N_I denotes the set of IR nodes in the ICG.
- C_I denotes the set of control flow edges in the ICG.
- D_I denotes the set of data flow edges in the ICG.
- W_I denotes the set of weak data flow edges from the base program to the ICG.

Other important data structures:

- $\mathbb{I}$ denotes the set of ICGs, initially empty.

Figure 2: Data structures used in ICG-related algorithms. The scope of $N, C, D, \mathbb{I}$ is the context of dynamic compilation of a single method. Changes to these sets will be visible in subsequent compilation passes.

them in all optimization heuristics. Unless the basic block containing the placeholder is eliminated, the placeholder, and hence the associated ICG, cannot be optimized away by the compiler.

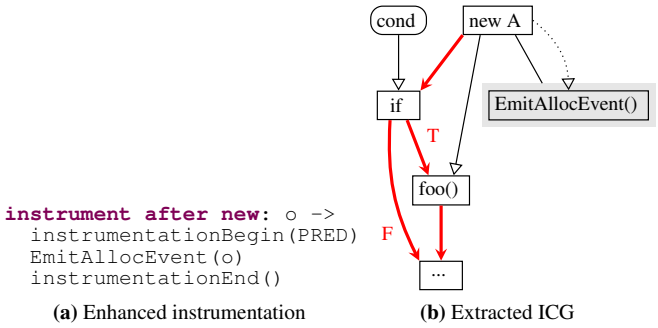
The procedure for extracting ICG is specified in Algorithm 1. The algorithm employs the data structures defined in Figure 2, and its key part is the identification of ICG nodes.

For each node b_I corresponding to an invocation of instrumentationBegin(), we collect all IR nodes reachable from b_I via the CFG into a set of ICG nodes, until we encounter a node corresponding to an invocation of instrumentationEnd() (Line 5–9). Next, we add to the set of ICG nodes also nodes that represent data values used exclusively within the ICG, i.e., nodes that are not involved in any control flow and that are only involved in the data-flow among existing ICG nodes (Line 10–14). With the set of ICG nodes identified, we collect the control-flow edges, data-flow edges, and *weak data-flow edges* between ICG nodes into their respective sets (Line 15–17). We then remove ICG nodes and ICG edges from the base program IR graph (Line 18–36).

```
1  procedure ExtractICGs
2    foreach b_I ∈ N | (b_I is a callsite invoking
3                       instrumentationBegin) do
4      N_I ← {b_I}
5      repeat
6        N_I ← N_I ∪ {v ∈ N | v ∉ N_I
7            ∧ (∃u ∈ N_I | ⟨u,v⟩ ∈ C∧(u is not
8                a callsite invoking instrumentationEnd))}
9      until N_I not changed
10     repeat
11       N_I ← N_I ∪ {u ∈ N | u ∉ N_I
12           ∧({v | ⟨v,u⟩ ∈ C ∨ ⟨u,v⟩ ∈ C} = ∅)
13           ∧({v | ⟨u,v⟩ ∈ D} ⊆ N_I)}
14     until N_I not changed
15     C_I ← {⟨u,v⟩ ∈ C | u ∈ N_I ∧ v ∈ N_I}
16     D_I ← {⟨u,v⟩ ∈ D | u ∈ N_I ∧ v ∈ N_I}
17     W_I ← {⟨u,v⟩ ∈ D | u ∉ N_I ∧ v ∈ N_I}
18     N ← N − N_I
19     let p_I ∈ {u | ⟨u,b_I⟩ ∈ D}
20     let e_I = callsite in N_I invoking instrumentationEnd
21     if p_I = PRED then
22       let a_I ∈ {u | ⟨u,b_I⟩ ∈ C}
23       C ← {⟨u,v⟩ ∈ C | u ∉ N_I ∧ v ∉ N_I}
24             ∪ {⟨a_I,v⟩ | ⟨e_I,v⟩ ∈ C}
25     else if p_I = SUCC then
26       let a_I ∈ {v | ⟨e_I,v⟩ ∈ C}
27       C ← {⟨u,v⟩ ∈ C | u ∉ N_I ∧ v ∉ N_I}
28             ∪ {⟨u,a_I⟩ | ⟨u,b_I⟩ ∈ C}
29     else  // p_I = HERE
30       a_I ← new node created as the placeholder
31       N ← N ∪ {a_I}
32       C ← {⟨u,v⟩ ∈ C | u ∉ N_I ∧ v ∉ N_I}
33             ∪ {⟨u,a_I⟩ | ⟨u,b_I⟩ ∈ C}
34             ∪ {⟨a_I,v⟩ | ⟨e_I,v⟩ ∈ C}
35
36     D ← (D − D_I) − W_I
37     𝕀 ← 𝕀 ∪ {⟨a_I,p_I,N_I,C_I,D_I,W_I⟩}
38   end
```

Algorithm 1: Extract ICGs from the base program IR graph.

To preserve a valid CFG, we reconnect the predecessor and the successor nodes of the ICG, either directly (Line 21–28) or via a placeholder node created for an ICG (Line 29–34).



(a) Enhanced instrumentation **(b)** Extracted ICG

Figure 3: Enhanced instrumentation and the IR graph of the base program after extracting the instrumentation as an ICG. The gray rectangle in the IR graph represents an extracted ICG (the delimitation API invocations are omitted for clarity), solid black lines without arrows represent the association between an ICG and a base program node, and the dotted edges with hollow arrows represent a *weak data-flow edge*.

Finally, we add a tuple representing an ICG into a set of ICGs (Line 37).

To put the concepts presented so far into the context of our example, consider again the program snippet shown in Figure 1a. The original instrumentation code from Figure 1d is now surrounded by invocations of the delimitation API methods, as shown in Figure 3a. In contrast to Figure 1e, the corresponding IR graph in Figure 3b shows the instrumentation as an ICG associated with the allocation node preceding the ICG in the base program IR.

3.4 Reconciling Operations on ICGs

The optimizations performed by the dynamic compiler can be expressed as transformations on the IR graph, which can be split into simpler graph-mutating operations, such as node elimination, value replacement, expansion, cloning, and movement. To preserve the purpose of the inserted code residing in ICGs associated with the base program IR nodes, we perform reconciling operations on the ICGs in response to the graph operations performed on the base program IR. We now review each of the reconciling operations defined in Algorithm 2.

Node elimination. Removes a node from the IR graph. This operation is primarily used to eliminate dead code. The corresponding reconciling operation is to remove the associated ICG (Line 3–4).

Value node replacement. Replaces the origin node in a data flow edge with another node. This operation is involved in many optimizations, e.g., constant folding. The corresponding reconciling operation is to update all affected *weak data-flow edges* in all ICGs to use the replacement node as the new source of a value (Line 5–7).

Input : *op*: node operation applied to the base program IR graph by the compiler

1 **procedure** *ReconcileICGs(op)*
2 **switch** *op* **do**
3 **case** *elimination of* n*:*
4 $\mathbb{I} \leftarrow \{\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I} \mid a_I \neq n\}$
5 **case** *value replacement of* $n \rightarrow n_r$*:*
6 $\mathbb{I} \leftarrow \bigcup_{\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I}} \{\langle a_I, p_I, N_I, C_I, D_I,$
7 $\{\langle u, v\rangle \in W_I \mid u \neq n\} \cup \{\langle n_r, v\rangle \mid \langle n, v\rangle \in W_I\}\rangle\}$
8 **case** *expansion of* $n \rightarrow$
9 *subgraph with entry* b_{sub} *and exit* e_{sub}*:*
10 **foreach** $\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I} \mid a_I = n$ **do**
11 **if** $p_I = PRED$ **then**
12 $\mathbb{I} \leftarrow (\mathbb{I} - \langle a_I, p_I, N_I, C_I, D_I, W_I\rangle)$
13 $\cup \{\langle e_{sub}, p_I, N_I, C_I, D_I, W_I\rangle\}$
14 **else if** $p_I = SUCC$ **then**
15 $\mathbb{I} \leftarrow (\mathbb{I} - \langle a_I, p_I, N_I, C_I, D_I, W_I\rangle)$
16 $\cup \{\langle b_{sub}, p_I, N_I, C_I, D_I, W_I\rangle\}$
17 **end**
18 **case** *cloning of* $n \rightarrow n_c$*:*
19 **foreach** $\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I} \mid a_I = n$ **do**
20 $N'_I \leftarrow$ clone N_I with mapping function $\mathbf{map}_{clone}$
21 $C'_I \leftarrow \bigcup_{\langle u,v\rangle \in C_I} \{\langle \mathbf{map}_{clone}(u), \mathbf{map}_{clone}(v)\rangle\}$
22 $D'_I \leftarrow \bigcup_{\langle u,v\rangle \in D_I} \{\langle \mathbf{map}_{clone}(u), \mathbf{map}_{clone}(v)\rangle\}$
23 $W'_I \leftarrow \bigcup_{\langle u,v\rangle \in W_I} \{\langle u, \mathbf{map}_{clone}(v)\rangle\}$
24 $\mathbb{I} \leftarrow \mathbb{I} \cup \{\langle n_c, \mathbf{map}_{clone}(p_I), N'_I, C'_I, D'_I, W'_I\rangle\}$
25 **end**
26 **case** *movement of* n*:*
27 // nothing to be done
28 **endsw**

Algorithm 2: Reconcile ICGs for node operations in the base program IR graph.

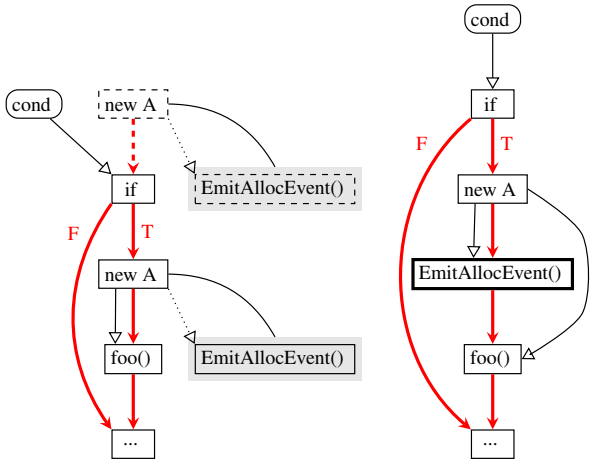
Node expansion. Expands a node into a subgraph which replaces the original node in the CFG. This operation is typically used to implement IR lowering [35], and often followed by a value node replacement operation. The corresponding reconciling operation is to re-associate the ICGs with either the entry or the exit node of the subgraph replacing the original node (Line 8–17).

Node cloning. Duplicates an IR node. This operation is often used in transformations implementing, e.g., loop peeling, loop unrolling, or tail duplication. The newly created node is usually immediately moved to a different location in the CFG, and the original node is sometimes eliminated. In any case, the corresponding reconciling operation is to clone the associated ICG and attach it to the newly created IR node (Line 18–25).

Node movement. Relocates a node to a different location in the CFG. This operation usually follows a cloning operation, because the clone needs to be moved to a new location. It can be used as a standalone operation to implement, e.g., loop-invariant code motion. Node movement is a change

Method	Description	Default
	Static query intrinsics	
isMethodCompiled	Returns true if the enclosing method has been compiled by the dynamic compiler.	false
isMethodInlined	Returns true if the enclosing method is inlined.	false
getRootName	Returns the name of the root method for the current compilation task. If the enclosing method is inlined, it returns the name of the method into which it was inlined.	"unknown"
	Dynamic query intrinsics	
getAllocationType	Returns the kind of heap allocation for a directly preceding allocation site. In HotSpot, the possible return values are {HEAP, TLAB}, representing a direct heap allocation (slow path), or a TLAB allocation (fast path). If the allocation site was eliminated, the method returns a special error value.	ERROR
getLockType	Returns the runtime lock type for a directly preceding lock site. In HotSpot, the possible return values are {BIASED, RECURSIVE, CAS}, representing the different locking strategies.	ERROR

Table 2: Query intrinsics available to the developer of bytecode instrumentation.



(a) After reconciling **(b)** Final IR graph

Figure 4: Base program IR with the associated ICGs after moving the allocation into the then-branch of the conditional, and after splicing the IGCs back into the base program IR. Dashed lines represent eliminated nodes and edges.

that does not affect the relative position between the moved IR node and the associated ICG. Consequently, no special reconciling operation is needed—the ICG implicitly "follows" the associated IR node around.

To illustrate the effect of the reconciling operations, we now return to our running example. In Figure 3b we left off with the instrumentation code extracted into an ICG. We assume that as a result of PEA (which disregards the ICG), the dynamic compiler decides to move the allocation into the then-branch of the conditional. To perform this transformation, the compiler first clones the allocation node, which triggers a reconciling operation resulting in the cloning of the associated ICG (this also involves updating all the IR edges to use the newly created allocation node). The compiler then moves the cloned IR node to the new location in the then-branch, and eliminates the original allocation node, triggering the elimination of the original ICG. The IR graph resulting from these operations is shown in Figure 4a.

3.5 Querying Compiler Decisions

An important aspect of our approach is that it allows the inserted code to query and adapt to the dynamic compiler's decisions. The queries are represented by invocations of special methods that are recognized and handled by the compiler similarly to intrinsics. We call these special methods *query intrinsics*, and whereas typical compiler intrinsics expand to a sequence of machine code instructions, query intrinsics expand to an IR subgraph comprising one or more IR nodes. Depending on the type of the replacement subgraph, we distinguish between static and dynamic query intrinsics.

The *static query intrinsics* expand to constant value nodes, which reflect static (compile-time) decisions of the dynamic compiler. Examples include the name of the compiling root method, or whether a method is compiled. In practice, the inserted code would typically use the *static query intrinsics* to limit the profiling scope. For instance, the inserted code can query whether its containing method is compiled, which allows enclosing all profiling code in a guarded block enabled only for compiled methods. This in turn allows collecting metrics only for the execution of compiled methods.

The *dynamic query intrinsics* expand to ϕ-function nodes. Depending on which runtime path is taken during program execution, the ϕ-function node selects a distinct constant value representing the path. This is useful when a compiler expands a base program IR-node into a subgraph containing multiple code paths that are selected at runtime. For instance, the inserted code can query whether an object was allocated in a thread-local allocation buffer (TLAB) or directly on the heap, or what kind of locking was used with a particular lock.

The query intrinsics recognized by the compiler represent an API that provides an instrumentation developer with the means to determine both compile-time and runtime compiler decisions, and allows creating an instrumentation that adapts accordingly. An overview of the methods making up the API is shown in Table 2.

3.6 Splicing ICGs

Towards the end of the dynamic compilation, we splice the ICGs back into the base program IR, as shown in Algorithm 3. For each ICG, we first evaluate all query intrinsics and replace the corresponding nodes with the resulting IR subgraph (Line 3–10). We then remove the invocations of the delimitation API methods (Line 12–14), and splice the ICG into the base program IR graph. Depending on the constant argument passed to the instrumentationBegin() method, which is either PRED, SUCC, or HERE, we insert the ICG after the associated node, (Line 15–19), before the associated node (Line 20–24), or in place of the associated placeholder node (Line 25–31). Finally, we convert the *weak data-flow edges* back to normal data flow edges (Line 32–39). If the originating node for a *weak data-flow edge* is not available in the resulting IR graph, it will be replaced with a default value corresponding to its type (Line 36–39).

Coming back to our running example, the result of splicing the ICGs back into the base program IR is shown in Figure 4b, with the invocation of the EmitAllocEvent() method relocated to the then-branch of the conditional.

4. Improving Existing Tools

One of the use cases for our approach is improving the existing profilers and tools based on bytecode instrumentation. These tools allow observing program execution at the bytecode level, but they fail to provide insight into execution at the level of compiled code. Profiling at the bytecode level tends to overprofile certain operations compared to the execution of compiled code, because modern JVMs will try to optimize them. The inserted instrumentation code tends to perturb certain optimizations, further reducing accuracy of the results.

Our approach allows improving the existing tools by enabling observation of program execution at the level of compiled code (but still using bytecode instrumentation) and by avoiding optimization perturbations arising from increased method sizes due to the inserted instrumentation code. We illustrate the benefits of our approach on three case studies covering allocation profiling (Section 4.1), object-lifetime profiling (Section 4.2), and callsite profiling (Section 4.3).

4.1 Impact on Allocation Profiling

Allocation profiling is generally used to identify allocation hotspots, because these may be associated with high garbage collection (GC) overheads. Commonly used profilers such as Netbeans profiler [30], Eclipse TPTP [39], JProfiler [38], and hprof [29] all support this kind of analysis. However, these profilers rely on bytecode instrumentation to track all object and array allocations, which perturbs the dynamic compiler's optimizations, and ultimately results in over-profiling [20, 27]. An allocation hotspot profiler may thus draw attention to places with high allocation rates (or amounts of allocated

```
1  procedure SpliceICGs()
2    foreach ⟨a_I, p_I, N_I, C_I, D_I, W_I⟩ ∈ 𝕀 do
3      foreach n_I ∈ N_I | (n_I is a query intrinsic) do
4        ⟨n_eval, N_eval, D_eval⟩ ← Evaluate(n_I, a_I)
5        N_I ← (N_I − {n_I}) ∪ N_eval
6        C_I ← {⟨u,v⟩ ∈ C_I | u ≠ n_I ∧ v ≠ n_I}
7              ∪ {⟨u,v⟩ | ⟨u,n_I⟩ ∈ C_I ∧ ⟨n_I,v⟩ ∈ C_I}
8        D_I ← {⟨u,v⟩ ∈ D_I | u ≠ n_I}
9              ∪ {⟨n_eval,v⟩ | ⟨n_I,v⟩ ∈ D_I} ∪ D_eval
10     end
11
12     let b_I = callsite in N_I invoking instrumentationBegin
13     let e_I = callsite in N_I invoking instrumentationEnd
14     N ← N ∪ (N_I − {b_I, p_I, e_I})
15     if p_I = PRED then
16       C ← {⟨u,v⟩ ∈ C | u ≠ a_I}
17             ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
18             ∪ {⟨a_I,v⟩ | ⟨b_I,v⟩ ∈ C_I}
19             ∪ {⟨u,v⟩ | ⟨u,e_I⟩ ∈ C_I ∧ ⟨a_I,v⟩ ∈ C}
20     else if p_I = SUCC then
21       C ← {⟨u,v⟩ ∈ C | v ≠ a_I}
22             ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
23             ∪ {⟨u,a_I⟩ | ⟨u,e_I⟩ ∈ C_I}
24             ∪ {⟨u,v⟩ | ⟨u,a_I⟩ ∈ C ∧ ⟨b_I,v⟩ ∈ C_I}
25     else  // p_I = HERE
26       N ← N − {a_I}
27       C ← {⟨u,v⟩ ∈ C | u ≠ a_I ∧ v ≠ a_I}
28             ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
29             ∪ {⟨u,v⟩ | ⟨u,a_I⟩ ∈ C ∧ ⟨b_I,v⟩ ∈ C_I}
30             ∪ {⟨u,v⟩ | ⟨u,e_I⟩ ∈ C_I ∧ ⟨a_I,v⟩ ∈ C}
31
32     D ← D ∪ {⟨u,v⟩ ∈ D_I | v ≠ b_I}
33     foreach ⟨u,v⟩ ∈ W_I do
34       if NodeIsAvailable(u,v) then
35         D ← D ∪ {⟨u,v⟩}
36       else
37         d ← the default value of type of u
38         N ← N ∪ {d}
39         D ← D ∪ {⟨d,v⟩}
40     end
41   end
42
43 function Evaluate(n_I, a_I)
     return a subgraph ⟨res, N_sub, D_sub⟩ representing the evaluation result of the query intrinsic n_I. res denotes
44   the value of the evaluated query intrinsic, N_sub denotes all nodes (including res) in the subgraph, and
     D_sub denotes all data-flow edges in the subgraph.
45
46 predicate NodeIsAvailable(u, v)
     return true if u has not been eliminated and can be
47   scheduled before v.
```

Algorithm 3: Splice ICGs into the base program IR graph.

```
instrument after new: o ->
  instrumentationBegin(PRED)
  if (isMethodCompiled())
    EmitHeapAllocEvent()
  else
    EmitInterpreterAllocEvent()
  instrumentationEnd()

  instrumentationBegin(HERE)
  EmitBytecodeAllocEvent()
  instrumentationEnd()
```

Figure 5: Pseudo-code of the instrumentation used by the allocation profiler to track object allocations.

memory) which in reality may have only a negligible impact on the GC overhead.

Also, previously published results on workload characterization [23, 32, 33] capture a workload's allocation behavior, but fail to differentiate between allocations that can be optimized away and allocations that are more costly because the GC will have to take care of the garbage later. In such a case, a summary quantification of the amount of overprofiled allocations may improve the results and enable more realistic characterization of allocation behavior in the future.

To gauge the potential for allocation over-profiling, we developed an allocation profiler which uses bytecode instrumentation to track object allocations. The instrumentation, shown in Figure 5, uses the delimitation API to make itself visible to the dynamic compiler. It uses two instrumentation blocks, one associated with the allocation, which will be executed only when an actual allocation occurs, and one anchored to the place where the allocation occurs at bytecode level. The former instrumentation block makes use of the isMethodCompiled() intrinsic to distinguish between interpreted-mode allocation and compiled-mode allocation. Below, we report on stack allocations calculated as the difference between the bytecode-level allocations (counted via EmitBytecodeAllocEvent()) and actual allocations (counted via EmitHeapAllocEvent() and EmitInterpreterAllocEvent()).

We profiled selected benchmarks[2] from the DaCapo 9.12 suite [6] on a multi-core platform[3], and report results for the 1st (startup) and the 15th (steady state) benchmark iteration[4]. The proportion of allocation types is shown in Figure 6, with the actual values shown in Table 3. During the startup iteration, in which the benchmark code is in the least optimized form, the proportion of stack allocations ranges from zero

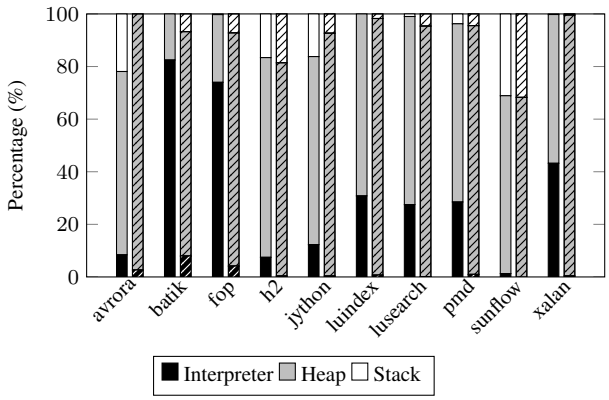


Figure 6: Proportions of allocation types for the 1st (without ▨ pattern) and the 15th (with ▨ pattern) benchmark iteration.

(batik, luindex) to 31.09% (sunflow), and is 9.1% on average (arithmetic mean). The proportion of stack allocations in the steady-state iteration ranges from zero (avrora) to 31.73% (sunflow), and is 8.29% on average.

Table 3 also shows the results in terms of allocated memory. For the startup iteration, the proportion of stack-allocated memory ranges from zero (batik, luindex) to 26.22% (sunflow), and is 7.22% on average. For the steady-state iteration, the proportion of stack-allocated memory ranges from zero (avrora) to 26.72% (sunflow), and is 6.06% on average. The overprofiling percentage generally tends to be lower for the amount of allocated memory compared to the number of allocations, because the result for the amount of allocated memory is naturally weighted by the allocated object sizes.

We note a striking difference in the numbers of startup and steady-state stack allocations for the avrora and jython benchmarks. Further profiling[5] reveals that the differences are due to each benchmark's initialization phase[6].

Both the number of stack allocations and the amount of memory allocated on the stack potentially affect the ranking of the allocation hotspots in the resulting profile. Without our approach, an allocation hotspot profiler may report hot allocation sites that will be optimized away by the dynamic compiler, rendering the output of the profiler "un-actionable".

4.2 Impact on Object Lifetime Analysis

Another application of allocation profiling is to collect information on memory-related behavior of programs, which helps

[2] We excluded the tomcat, tradebeans, and tradesoap benchmarks due to well known issues (see http://sf.net/p/dacapobench/bugs/70/ and http://sf.net/p/dacapobench/bugs/68/). We also excluded eclipse due to its incompatibility with Java 8.

[3] Intel Xeon E5-2680 2.7GHz with 8 cores, 64 GB of RAM, CPU frequency scaling and Turbo mode disabled, Oracle JDK 1.8.0_20 b26 Hotspot Server VM (64-bit), running on Ubuntu Linux Server 64-bit version 12.04.5 64-bit.

[4] The profiler introduces an average overhead (i.e., geometric mean for DaCapo) of 9% for exact profiling, and 3% for sampling with a rate of 1/1000. Our approach does not introduce any noticeable additional runtime overhead (see Section 6 for a discussion of performance issues).

[5] By using the getRootName() intrinsic, we can extend the identification of an allocation site with the name of the root method into which the allocation site is inlined.

[6] An allocation site at cck.text.StringUtil.convertToHex inlined to avrora.monitors.PacketMonitor$Mon.renderPacket performs most of the stack allocations in the first iteration of avrora; various allocation sites in org.python.antlr.PythonParser dominate the stack allocations in the first iteration of jython. By excluding these, the average proportion of stack allocations in the first iteration of jython is 5.68%.

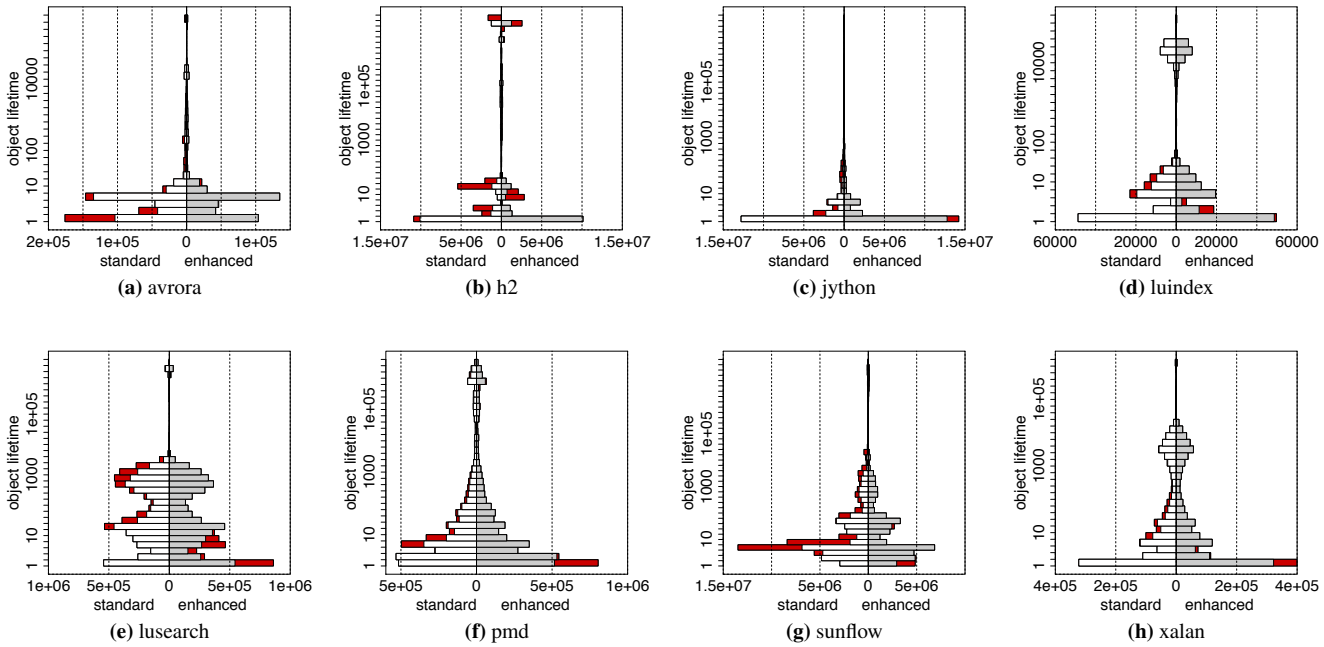


Figure 7: Back-to-back histogram of object lifetime distributions for the first benchmark iteration. The lifetimes are obtained using ET with the standard and the enhanced instrumentations. The X-axis denotes the object counts per bin. The differences between bins are marked in red. The results for the batik and the fop benchmarks are excluded due to less significant differences.

Benchmark	*1st iteration*				*15th iteration*			
	stack #	%	stack mem.	%	stack #	%	stack mem.	%
avrora	111 676	21.88	3.6M	17.78	3	0	96	0
batik	0	0	0	0	19 438	6.76	0.8M	4.47
fop	2548	0.26	82K	0.16	70 823	7.22	2.4M	4.74
h2	411 038	16.61	12M	9.19	460 251	18.58	14.3M	10.9
jython	3 819 122	16.22	207M	16.03	983 843	7.33	47.7M	6.34
luindex	0	0	0	0	2468	1.74	79K	1.49
lusearch	55 717	0.99	1.8M	0.67	262 144	4.64	8.4M	3.16
pmd	136 788	3.75	3.3M	1.88	173 994	4.45	4.3M	2.28
sunflow	18 718 463	31.09	691M	26.22	19 101 856	31.73	704M	26.72
xalan	3024	0.23	0.2M	0.28	5600	0.43	0.3M	0.46
average		9.1		7.22		8.29		6.06

Table 3: Number of stack allocation and stack-allocated memory (in bytes) per benchmark, along with their proportions.

in GC algorithm design and development. For instance, the Merlin algorithm [17] maintains allocation and last-reachable timestamps for each object, and calculates the lifetime as the difference of the two timestamps when an object is garbage-collected. Compared to a brute-force approach that repeatedly forces a whole-heap GC, the Merlin algorithm provides more accurate and fine-grained results with less overhead, especially when integrated in a VM such as Jikes RVM where the object header or GC can be easily changed [40]. For other VM implementations, researchers typically use instrumentation-based dynamic analysis tools such as ElephantTracks (an implementation of the Merlin algorithm) [31].

The instrumentation needed to gather the information required by the Merlin algorithm is inherently heavy-weight, because it needs to track object allocations, object usage, and reference updates. Consequently, the inserted code significantly impacts the ability and willingness of the dynamic compiler to optimize the instrumented code. Specifically, the (significantly) increased method sizes prevent inlining, while the object tracking causes all references to escape, thus forcing all object allocations to happen on the heap (including those that could be converted to stack allocations).

This causes two problems. The first is that we are unable to observe the application's original allocation behavior—we observe more allocations, and we may observe them in different order compared to an uninstrumented program. For example, the Merlin algorithm and its variant found in ElephantTracks use an allocated-amount counter or method invocation/return counter as a logical clock, both of which will be influenced by optimizing allocations away or by moving the allocation code around. The second problem is that the increased number of heap allocations causes the GC to behave differently from what would be observed with an uninstrumented program.

In performance engineering research, the inaccurate allocation information and the differences in GC behavior in response to slight changes in GC-relevant workload make it extremely difficult to model and predict the impact of GC on application performance [24]. While there is a need for accurate object-lifetime profiles, existing tools such as ElephantTracks cannot provide it.

Overprofiling of the number of object allocations influences the distribution of object lifetimes. Also, even if objects

are not allocated on the stack, the allocation code may have been moved around by the dynamic compiler [37], influencing object lifetimes based on various logical clocks.

To quantify the impact of these observer effects on the object lifetimes, we profiled the same set of DaCapo benchmarks as in the previous case study using an implementation of the Merlin algorithm[7]. We developed two different variants of the instrumentation necessary to track object allocations, object usage, and reference updates. The first variant, which represents the baseline, is a standard instrumentation, in which the inserted code for tracking object allocations and object usage always emits all events and passes all object references to the profiler. The second variant takes advantage of our approach, and explicitly marks the instrumentation using the delimitation API from Table 2.

When tracking the target of an object-related operation, the instrumentation will receive an actual reference if an object was heap-allocated, or null if it was stack-allocated. Consequently, the object allocation and object usage events are only emitted for heap-allocated objects. We use an atomic logical clock represented by the cumulative amount of allocated memory, which is advanced regardless of the allocation type to enable comparison between results from the two versions[8].

To capture the increasing influence of the dynamic compiler, we collect allocation traces for the first benchmark iteration and compare the distributions of object lifetimes obtained using the two instrumentation variants.

The back-to-back histograms in Figure 7 show the object lifetime distribution obtained without and with being aware of stack allocations, respectively. The results for avrora show that stack allocations shrink the proportion of objects in the shorter-lifetime bin of the histogram. The results for xalan are interesting, because the benchmark has virtually no stack allocations (0.23% in the 1st iteration, and 0.43% in the 15th iteration). We can observe a significant increase in the number of very short-lived objects as they move to the shortest-lifetime bin. This is because PEA attempts to coalesce allocations and postpone them until the objects escape [37]. The results for sunflow demonstrate this kind of behavior in the case of a benchmark with a significant proportion of stack-allocated objects. We can observe that the differences in the histograms exhibit both shifting from bins corresponding to longer-lived objects to bins corresponding to shorter-lived objects, as well as proportional shrinking of some bins.

These changes in the distribution of object lifetime are significant. While they do not influence bytecode-level workload characterization results, which are only concerned with allocations, any research that depends on object lifetime profiles obtained by tools such as ElephantTracks is potentially influenced. For example, simulation of generational garbage collector behavior is particularly sensitive to inaccurate inputs. The over-profiled allocations artificially increase the rate at which the young-generation space fills up, and trigger simulated minor collections sooner than expected. This in turn influences the rate at which objects mature, affecting the simulation of tenured-generation collections. Because the GC is a non-linear system, this then derails the predictions of major collections [24]. Using the enhanced instrumentation allows obtaining accurate information about the memory-related behavior of a program, thus improving simulation results.

4.3 Impact on Callsite Profiling

Dynamic compilers in modern VM implementations aggressively inline methods at hot call sites [2, 4, 12] to eliminate the method invocation overhead, to expand the scope for other intraprocedural optimizations, and to enable specialization of the inlined code [19]. At polymorphic callsites the target method is determined by the receiver type and usually requires dynamic dispatch, which hinders inlining. However, when the number of target methods is very small, inlining can be still done with appropriate guards in place. In workload characterization research, callsite profiling is used to identify callsites [32, 34] that are suitable for inlining or inline caching, which requires collecting information on the actual number of target methods and on the distribution of receiver types.

For the Java benchmarks from the DaCapo suite, Sewe et al. [34] report that on average $97.8\,\%$ of callsites are monomorphic, and account for $91.5\,\%$ of method invocations. However, from this information we can only infer how much a JVM could optimize at these callsites, not what it actually does, i.e., that most of these callsites are actually inlined by a modern JVM. Yet with a classic instrumentation-based callsite profiler it is impossible to distinguish between an inlined and a non-inlined callsite, because the inlining behavior is not observable at the bytecode level.

Our approach allows building a profiler that can determine whether a callsite was inlined or not, enabling analysis of the inlining behavior for a particular JVM. Specifically, if we are interested in understanding or improving the inlining policy, we would prefer to profile callsites that were not inlined. In this context, the callsite information provided by a classic callsite profiler can be considered significantly overprofiled, and is therefore of little use.

Similar to allocation profiling presented earlier, bytecode instrumentation has a tendency to disturb optimizations—in this case inlining. This is because the inserted code (often excessively) increases methods sizes and compilers generally avoid inlining large methods. Therefore, even if we can determine whether a callsite has been inlined or not, the

[7] We adapted ElephantTracks to run on OpenJDK and Graal. We also excluded Graal classes and their dependencies from instrumentation.

[8] Both versions of ElephantTracks introduce an average overhead of a factor of 17 during startup, and of a factor of 79 and 68, respectively, during steady-state execution. The speedup observed in the version based on our approach is due to omission of events corresponding to stack-allocated objects. While these overhead factors may seem high, they are common for tools using such a heavy-weight instrumentation as ElephantTracks [31].

Benchmark	Bytecode-level Profiler	Non-inlined Callsite Profiler			
		Perturbed		Accurate	
	Method calls	Method calls	%	Method calls	%
avrora	619 416 614	81 417 434	13.14	56 855 805	9.18
batik	19 882 555	2 302 771	11.58	1 957 338	9.84
fop	34 145 814	3 802 440	11.14	3 241 928	9.49
h2	881 803 337	146 743 342	16.64	125 981 105	14.29
jython	424 639 480	50 547 787	11.90	32 425 816	7.64
luindex	95 286 242	12 375 687	12.99	7 165 321	7.52
lusearch	377 707 756	29 581 448	7.83	27 714 126	7.34
pmd	118 230 869	16 192 709	13.70	13 499 182	11.42
sunflow	1 973 544 191	222 996 162	11.30	67 157 052	3.40
xalan	301 667 030	25 613 673	8.49	20 232 863	6.71
total	4 846 323 888	591 573 453	12.21	356 230 536	7.35

Table 4: The number of method invocation aggregated over all profiled callsites. The total number of invocations is produced by a bytecode-level callsite profiler, while the numbers of invocations at non-inlined callsites are produced by a similar profiler using our approach. The data for the "perturbed" and "accurate" columns correspond to data collected with the respective dynamic compiler variant.

inlining decision for a particular callsite may have been perturbed by the instrumentation, resulting in loss of accuracy with respect to the execution of the base program without instrumentation.

To quantify the aforementioned two types of overprofiling, we developed a callsite profiler using bytecode instrumentation which counts the number of method invocations at each callsite. Without our approach, the profiler naïvely collects the total number of method invocations at each callsite, similar to what an existing callsite profiler would do. Using our approach, the inserted instrumentation code is associated with the call site, and will emit an event only when a callsite is not inlined.

To evaluate the potential loss of accuracy caused by perturbing the inlining optimization, we use two variants of the dynamic compiler. The first compiler variant, referred to as "perturbed", includes the size of the inserted code in the calculated method size, causing the instrumentation to influence inlining decisions. The second compiler variant, referred to as "accurate", disregards the size of the inserted code, which is what we normally do in our approach.

We profiled 15 iterations of the same set of DaCapo benchmarks as in Section 4.1 using the two compiler variants[9]. We report results for the 15th iteration (steady state) in Table 4, showing the total number of method invocations, and the number of non-inlined method invocations collected by the profiler using the "perturbed" and "accurate" variants of the dynamic compiler.

Compared to our (accurate) approach to profiling non-inlined callsites, the classic callsite profiler overprofiles $92.65\,\%$ of method calls. The instrumentation significantly perturbs the inlining optimization, increasing the number of method calls at non-inlined callsites by $4.86\,\%$.

Our approach thus enables more accurate characterization of JVM workloads and allows analyzing the inlining policy of a particular JVM using bytecode instrumentation. By combining accurate callsite profiling with calling context profiling, our approach also enables accurate stack depth profiling, which is again a metric commonly found in workload characterization research [23, 32, 34].

5. Enabling New Tools

The ability to profile program execution at both the bytecode level as well as at the level of compiled code enables construction of new tools that were previously impossible to build using bytecode instrumentation. We illustrate this on three additional case studies. In the first case study, we use the aforementioned callsite profiler to identify the causes for not inlining potentially hot callsites (Section 5.1). In the second case study, we use a calling-context-aware receiver-type profiler to explore the potential benefits of using calling-context information to resolve target methods at non-inlined polymorphic callsites (Section 5.2). In the third case study, we present a compiler testing framework (Section 5.3) relying on the ability to observe program behavior at the level of compiled code.

5.1 Identifying Inlining Opportunities

When tuning the inlining strategy to suit a particular program, we can instruct the dynamic compiler to print all inlining decisions[10]. The log usually includes the inlining decision for each compiled call site, along with reasons for not inlining specific call sites. If we rely on certain methods to be inlined, the log allows checking whether the expected inlining happened or failed and for what reason.

We could also use the log to identify additional optimization opportunities, but the logs produced by existing VMs do not help in deciding whether the reasons for not inlining a particular call site are worth analyzing. This is because there is not enough additional information (e.g. hotness) related to a particular call site, and also because the logs may contain duplicate entries for methods that were either inlined from different root methods or recompiled.

This is unfortunate, because if, for example, a VM developer or a researcher decides to include calling context into traditional receiver-type profiling to enable more inlining opportunities, the information missing in the inlining log prevents him or her to quickly gauge the actual potential for such optimization at individual call sites before committing to implementing it in the interpreter and dynamic compiler.

To make the inlining decision log more useful, we complement the log with the information about hotness of the non-inlined call sites collected in Section 4.3. We filtered

[9] The profiler introduces an average overhead of 49% for exact profiling, and 16% for sampling with a rate of $1/1000$.

[10] Enabled by -G:Log=InliningDecisions in Graal.

the resulting augmented inlining log, looking for inlining opportunities at polymorphic call sites. For each benchmark, we identified the hottest call site that was not inlined due to polymorphism-related reasons, as shown in Table 5. In general, the reason for not inlining these call sites is that they target too many types. The specific reason #1 (*no methods remaining after filtering less frequent methods*) means that the number of receiver types exceeds a preset limit[11] and that the frequency of the profiled types is below a preset threshold. The specific reason #2 (*relevance-based*) means that the compiler wanted to inline one or more targets, but the total size of these targets is over a certain limit.

A generally observed phenomenon is that a call site may have different distributions of dynamic receiver types in different calling contexts. For instance, a call site in a Java class library method often has library receiver types if invoked internally, but many other receiver types if invoked from application code. Because the receiver-type profiling in the interpreted mode lacks calling-context information, the negative inlining-decision results shown in Table 5 are based on information from all calling contexts. The compiler does not inline the target even for call sites for which the receiver type could be resolved within a particular calling context. This leads to a hypothesis that including calling context in receiver-type profiling may help in resolving the receiver type for some of the non-inlined call sites.

5.2 Calling-context Aware Receiver-type Profiler

To test the hypothesis, we extended the previous non-inlined callsite profiler to combine the existing calling-context profiling with a receiver-type profiling. With each non-inlined call site, the profiler associates a receiver type profile for each of 0, 1, 2, and 3 levels of calling context. The profiler therefore produces a distribution of dynamic receiver types for each non-inlined call site and calling-context level, with level 0 (i.e., no calling-context information) representing the baseline receiver-type profile.

We collected the context-sensitive receiver-type profiles for the selected DaCapo benchmarks[12], and for each call site, we determined whether it is possible to resolve the receiver to a single type using the additional calling-context information. We then calculated the number of invocations that could be resolved while considering 1, 2, and 3 levels of calling context at each call site. The results are shown in Figure 8.

While the results vary with each benchmark, we note that there are several benchmarks where the calling-context-sensitive receiver-type profiling could improve inlining. In particular, we observe that adding a single level of calling-context information to the receiver-type profile for jython allows resolving the receiver type in 60.7% of invocations at the non-inlined call sites. Adding a second level of calling-context information increases this to 72.2%. This suggests that calling-context-aware receiver-type profiling may be beneficial for the implementation of dynamic-language interpreters.

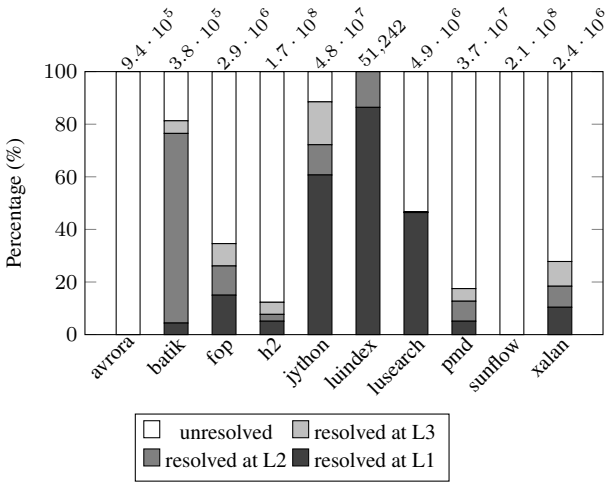


Figure 8: The percentages of invocations at non-inlined call sites for which the receiver type can be resolved using 1, 2, or 3 levels of calling context. Above each bar is the total number of invocations at non-inlined callsites.

Consistent with these findings, the most significant peak performance speedup in a recent trace-based JIT compiler for Java was achieved for jython (factor of 1.59) [15]. Performing the above analysis based on the currently available inlining decision logs is simply not possible. Using our approach, we were able to quickly create a tool that helps a compiler developer direct his or her efforts, instead of blindly following intuition.

5.3 Compiler Testing Framework

Unit testing is considered best practice in any serious software development project. When developing a dynamic compiler, the various optimizations operating at the IR level are perfect candidates for unit testing—to ensure that the optimizations never produce incorrect code. However, compilers always perform many optimizations, often repeatedly and in multiple passes, because transformations performed by some optimizations may enable other optimizations to yield better results. Therefore, in addition to individual optimizations, the developers should be also able to test whether the expected synergy between optimizations actually occurs. Yet such tests are difficult to write at the IR level; it may be difficult to specify what the input and the result should be.

It is considerably easier to test the synergy between specific optimizations by executing the compiled code and checking if it produces an expected output. Currently, this approach is only able to detect incorrect results. Results that are merely suboptimal, because some of the expected optimizations did not happen, or because the expected synergy between optimizations did not materialize, will not be detected.

[11] Determined by the -XX:TypeProfileWidth option.

[12] The profiler introduces an average overhead of factor 68 for exact profiling, and 66% for sampling with a rate of $1/1000$.

Benchmark	*Hottest non-inlined call site*		*Invocation Counter*	*Reason for not inlining*
	Caller	BCI		
avrora	DefaultMCU$Pin.write	29	179 553	#1
batik	CSSEngine.getComputedStyle	81	45 226	#1
fop	CompoundPropertyMaker.makeCompound	41	97 916	#2
h2	Select.queryFlat	123	6 848 343	#2
jython	PyObject.__getattr__	2	2 994 315	#1
luindex	IndexOutput.writeVInt	17	14 227	#2
lusearch	IndexSearcher.search	25	259 440	#2
pmd	SimpleJavaNode.childrenAccept	29	3 311 669	#1
sunflow	Geometry.intersect	28	41 155 041	#2
xalan	AbstractSAXParser.characters	59	708 840	#2

Table 5: Hottest call sites not inlined due to polymorphism-related reasons: #1: no methods remaining after filtering less frequent methods, #2: relevance-based. The call sites are represented by the enclosing method name as well as the bytecode index (BCI).

We can improve assertion-based testing using our approach, allowing the developers to specify the test input and the expected results using normal (Java) code instead of having to craft instances of the post-optimization IR. The test input (target code), on which the optimizations should be performed, is compiled and executed on the JVM, and when compiled by the dynamic compiler, it exposes the compiler decisions made at critical locations. A test case would then assert decisions to be made at concrete locations, e.g., expecting a particular allocation to be converted to a stack allocation, or expecting that certain callsites will be inlined.

Based on this idea, we built a testing framework to simplify testing of compiler optimizations and the combinations thereof. A high-level overview of the framework is shown in Figure 9. The input to the framework consists of the target code, the test case, and (optionally) the profiler code. The target code is normally compiled and executed. For simple test cases, the target code will typically use the methods of the compiler decision query API (c.f. Table 2) directly. For test cases requiring complex target code, the test will typically use a profiler, i.e., a specialized dynamic analysis focused at a specific optimization. The profiler will instrument the target code automatically when it is loaded by the JVM. The test case triggers the execution of the target code and captures the expected results in the form of assertions. For simple target code, or generally when it is possible to determine the expected result value exactly, the assertions will test for that value. For target code in form of large programs (such as the benchmarks from the DaCapo suite), determining the expected result value exactly may be difficult, therefore the assertions may expect the result value to be in a certain range.

The test execution has three major phases: warmup, profiling, and validation. During the warmup phase, the target code executes in the interpreted mode, mainly to collect internal profile information needed by the dynamic compiler. When the target code is compiled by the dynamic compiler, a trigger in the target code switches the test execution into profiling phase, which exercises the optimized target code and collects information on the decisions made by the dynamic compiler.

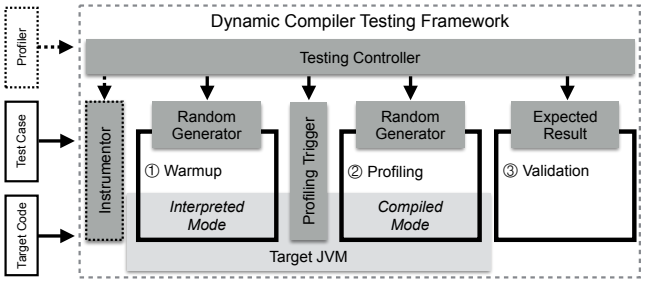


Figure 9: Overview of the dynamic compiler testing framework

The target code may exercise more than one code path, e.g., by using a random number generator to select certain code paths with predefined probability, which is especially useful for synthetic target code. Finally, during the validation phase, the results from the actual profile are compared with the expected values to determine the test result. If the target program is probabilistic, the comparison of the profiles should take into account the expected probabilities.

Our testing framework is built as an abstract base test class on top of JUnit. During test setup, the framework triggers the execution of the warmup operation, which causes the target code to be loaded by the JVM. If a profiler is used, the target code will be automatically instrumented. During warmup, the target methods use the `isMethodCompiled()` intrinsic as a guard to avoid profiling the target methods during interpreted execution. While warming up the target code, the test setup code polls a test-specific monitor to determine whether more warmup activity is needed. After the target code is warmed up, the test execution progresses to the profiling phase for a fixed number of target method invocations before advancing to the validation phases which checks the results. The code for both the profiling and the validation phases is located in a single test method identified by the `@Test` annotation.

To demonstrate the usage of the testing framework, we present an example intended to test the results of PEA. The code in Figure 10 shows the abstract base class providing the necessary support for the dynamic compiler test cases

```
1  /** Framework base class. */
2  abstract class JITTestCase extends TestCase {
3    /** Repeatedly invoke warmup()
4        until isWarmedUp() returns true. */
5    @Before
6    public final void warmUpTarget() { ... }
7
8    /** Returns TRUE with the given probability. */
9    protected boolean likely(double probability) { ... }
10
11   /** Warmup operation. */
12   protected abstract void warmup();
13
14   /** Returns TRUE if more warmup is needed. */
15   protected abstract boolean isWarmedUp();
16 }
17
18 /** The test case: Partial Escape Analysis. */
19 class PEATestCase extends JITTestCase {
20
21   static final double PROB = 0.8;
22   static final int ITERATIONS = 10000;
23   static final double EPSILON = 0.02;
24   static int cnt = 0;
25
26   protected void warmup() {
27     A.foo(likely(PROB));
28   }
29
30   protected boolean isWarmedUp() {
31     return cnt > 0;
32   }
33
34   @Test
35   public void testPartialEscape() {
36     cnt = 0;
37
38     for (int i = 0; i < ITERATIONS; i++) {
39       A.foo(likely(PROB));
40     }
41
42     assertEquals(((double) cnt)/ITERATIONS,
43       PROB, EPSILON);
44   }
45 }
46
47 /** The target code. */
48 class A {
49   static void foo(boolean invokeBar) {
50     A a = new A();
51
52     DelimitationAPI.instrumentationBegin(PRED);
53     if (CompilerDecision.isMethodCompiled()) {
54       PEATestCase.cnt++;
55     }
56     DelimitationAPI.instrumentationEnd();
57
58     if (invokeBar) {
59       a.bar();
60     }
61   }
62
63   /** This method will not be inlined. */
64   void bar() { ... }
65 }
```

Figure 10: Example of a simple Partial Escape Analysis test.

(Line 1–16), the actual test case (Line 18–45), and the target code (Line 47–65) with the inlined profiling code surrounded by invocations of delimitation API methods (Line 52–56), which can be delegated to a dedicated profiler.

Using a similar test, we discovered a subtle bug[13] where a later optimization pass reverted the optimization done by Graal's PEA. Some bugs may only appear when a certain combination of optimizations is applied at the same time. For example, instead of invoking the target method directly, a test case can invoke another method to repeatedly invoke the original target method within a for loop. This may trigger the application of both inlining and loop unrolling optimizations.

[13] Fix can be found at http://hg.openjdk.java.net/graal/graal-compiler/rev/1f4c9729c9f0

We discovered another subtle bug[14] when the previous test case was positioned in a loop. In line with best practices, such test cases should become part of a project's test suite to avoid regression in future versions.

6. Discussion

Below we discuss the benefits and limitations about our approach.

Applicability and ease of use. Our approach can be easily integrated into existing tools and profilers for improved accuracy. For example, in the case of the allocation profiler, one only needs to wrap the original instrumentation code with invocations of the delimitation API methods to avoid over-profiling of allocations.

Our approach also simplifies implementation of new profilers that focus on the runtime behavior of code optimized by the dynamic compiler. The query intrinsics API allows the profiler to determine the actual runtime path taken by specific operations. For instance, an allocation profiler can find out where an object is allocated on the heap (i.e., in the TLAB or in the eden space). A lock profiler can query the runtime behavior upon lock acquisition (e.g., recursive locking, biased locking, epoch expired).

Previously, writing such profilers required direct modification of the dynamic compiler to insert the appropriate profiling logic, which in turn required certain familiarity with the VM internals. Each tool required a tool-specific VM version, and making changes to a tool forced a recompilation of the VM. In contrast, our approach allows developers to create such profilers with widely used bytecode instrumentation techniques. Even though certain familiarity with compiler optimizations is assumed, our approach does not significantly change the level of abstraction the tool developer deals with, compared to having to deal with VM implementation details of production-level VMs.

Our approach is also applicable in other contexts, not just instrumentation code; the queries can be made directly in the source code of the base program, as shown in Figure 10.

Improved profiler accuracy. In general, tools based on bytecode instrumentation inflate the base-program code and introduce additional dependencies on base-program objects. This perturbs the results of analyses such as PEA, and prevents optimizations such as inlining, scalar replacement, or code motion in general, that would otherwise be performed on the base-program code. Our approach eliminates this problem for all such cases, while requiring only minor changes to existing tools. Consequently, our approach improves the accuracy of profilers that are inherently susceptible to this kind of observer effect. We demonstrate and quantify this improvement in Section 4, where the ability to accurately observe the allocation behavior of applications running on a state-of-the-art

[14] Fix can be found at http://hg.openjdk.java.net/graal/graal-compiler/rev/c215dec9d3cf

VM is extremely important, e.g., for modeling the impact of garbage collection on application performance [24].

Obviously, our approach is not a general solution to all kinds of observer effects. We only avoid perturbations of the dynamic compiler's optimization decisions, and make the profiler aware of the applied optimizations. When it comes to external metrics such as wall clock time of method executions, profilers will still suffer from the observer effect due to the execution overhead of the instrumentation code, but that can be mitigated by sampling techniques.

Testing dynamic compiler optimizations. Finding a performance bug in a dynamic compiler is difficult, because there is nothing obviously wrong but an optimization not being activated. Reporting such a bug is also difficult, because the circumstances may be complex and difficult to describe, which subsequently impairs the ability of a compiler developer to reproduce the bug. However, having a test case that showcases a bug is an entirely different matter. The testing framework based on our approach allows writing test cases that help reproducing and locating such performance bugs in a dynamic compiler. The test cases can be also used to document the expected behavior of specific optimizations, e.g., inlining decisions at specific call sites, and in general optimizations that use different runtime paths to improve common-case performance. The framework can be also used by developers who care about performance of an application running in a VM with a dynamic compiler. They can write performance test cases to check whether certain parts of the application are optimized as expected.

Since our approach focuses on dynamically compiled code, the testing framework requires an initial warmup phase to exercise the base program code in interpreted (or baseline-compiled) mode. In comparison to unit testing that primarily checks functional correctness, our approach requires longer test execution time.

Performance impact. Our approach does not introduce any overhead to the execution of the analysis code, nor does it aggravate the (in)efficiency inherent to a particular analysis. However, in some circumstances, our approach may improve the performance of a particular profiler by filtering out unnecessary profiling sites, such as stack allocations in the case of the allocation profiler. Our approach is fully compatible with the sampling technique, which helps improve the performance of heavy profilers.

Our approach may affect dynamic compilation time. On the one hand, it requires additional compilation phases to extract, reconcile, and splice the ICGs back into the IR. On the other hand, it decreases the complexity of the base-program IR by extracting the ICGs. In practice, the possible extra compilation overhead introduced by our technique is negligible compared to the overhead caused by many instrumentation-based profilers.

Implementation. The proposed approach, including the query intrinsics API, is implemented in Oracle's Graal compiler [13] version 0.6. In terms of code changes, the difference between the base Graal and our implementation is 1448 insertions and 8 deletions. The implementation currently relies on the explicit marking of the inserted instrumentation code. To relieve the developer from having to care about marking the boundaries of inserted instrumentation code, we have modified the DiSL [25, 26] instrumentation framework to automatically insert the necessary delimitation API invocations. Both the implementation of our approach and the modified DiSL can be found at http://dag.inf.usi.ch/downloads/.

Our current implementation targets execution of code produced by an optimizing dynamic compiler, i.e., the query intrinsics receive special handling only when the code is compiled. The interpreter is left unmodified, therefore when executing in interpreted mode, the query intrinsics return defaults that are adequate for the interpreter. In general, certain optimizations may be performed by a baseline compiler. The support for handling the query intrinsics needs to be implemented in these compilers depending on the optimizations they perform. The implementation effort would be reduced in runtimes employing a single optimizing compiler with support for different optimization levels, such as Jikes RVM.

7. Conclusion

Prevailing profilers based on bytecode instrumentation often yield inaccurate profiles, because neither such profilers are aware of the optimizations performed by the dynamic compiler, nor the dynamic compiler is aware of the inserted instrumentation code. That is, the dynamic metrics conveyed in the profiles usually do not accurately reflect the execution of the application without profiling.

We present a new approach to make inserted profiling code explicit to the dynamic compiler and to allow the inserted code to query runtime path decisions in the optimized compiled code, enabling the collection of accurate profiles that faithfully represent the execution of a base program without profiling (w.r.t. the applied optimizations). We demonstrate the benefits and the applicability of our approach with case studies in different scenarios.

On the one hand, our approach allows improving the accuracy of existing profilers, which typically only need minor modifications to wrap the instrumentation with invocations of the delimitation API methods. This application is supported by allocation profiling, object lifetime analysis, and callsite profiling.

On the other hand, our approach enables new kinds of profilers that allow for gathering information on the effectiveness of dynamic compiler optimizations. This application is supported by inlining profiling and calling-context-aware receiver-type profiling, which show that our analysis-agnostic approach makes it easy for, e.g., language implementers to quickly create analyses that help in deciding which optimiza-

tions are worth pursuing, or what would be the effect of particular context information in certain optimizations.

Finally, our approach eases the development of performance testing tools, supporting both compiler implementers and application developers who rely on particular dynamic compiler optimizations for critical parts of their application code. This application is supported by the compiler testing framework, which presents a novel framework that enables writing simple test cases for individual compiler optimizations without interfering with the way the dynamic compiler combines these optimizations. This allows discovering performance bugs in the dynamic compiler that frequently occur due to the interplay of different optimizations, and that are generally difficult to detect, reproduce, and fix. Thanks to our testing framework, we discovered and reported two performance bugs in Graal that were subsequently fixed by the Graal team.

Acknowledgments

The research presented in this paper was supported by Oracle (ERO project 1332), by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (contract ACP2-GA-2013-605442), and by the Charles University institutional funding (project SVV-2015-260222). We especially thank Thomas Würthinger and Lukas Stadler for their support with Graal.

References

[1] Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.: A survey of adaptive optimization in virtual machines. Proceedings of the IEEE 93(2), 449–466 (2005)

[2] Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.F.: Adaptive optimization in the Jalapeño JVM. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 47–65. OOPSLA '00, ACM (2000)

[3] Arnold, M., Fink, S., Sarkar, V., Sweeney, P.F.: A comparative study of static and profile-based heuristics for inlining. In: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization. pp. 52–64. DYNAMO '00, ACM (2000)

[4] Arnold, M., Hind, M., Ryder, B.G.: Online feedback-directed optimization of Java. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 111–129. OOPSLA '02, ACM (2002)

[5] Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 168–179. PLDI '01, ACM (2001)

[6] Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications. pp. 169–190. OOPSLA '06, ACM (2006)

[7] Blanchet, B.: Escape analysis for object-oriented languages: Application to Java. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 20–34. OOPSLA '99, ACM (1999)

[8] Blanchet, B.: Escape analysis for Java: Theory and practice. ACM Trans. Program. Lang. Syst. 25(6), 713–775 (2003)

[9] Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño dynamic optimizing compiler for Java. In: Proceedings of the ACM 1999 Conference on Java Grande. pp. 129–141. JAVA '99, ACM (1999)

[10] Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 1–19. OOPSLA '99, ACM (1999)

[11] Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations. pp. 35–49. IR '95, ACM (1995)

[12] Detlefs, D., Agesen, O.: Inlining of virtual methods. In: Proceedings of the 13th European Conference on Object-Oriented Programming. pp. 258–278. ECOOP '99, Springer-Verlag (1999)

[13] Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., Mössenböck, H.: An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In: Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages. pp. 1–10. VMIL '13, ACM (2013)

[14] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)

[15] Häubl, C., Wimmer, C., Mössenböck, H.: Trace transitioning and exception handling in a trace-based JIT compiler for Java. ACM Trans. Archit. Code Optim. 11(1), 6:1–6:26 (2014)

[16] Hazelwood, K., Grove, D.: Adaptive online context-sensitive inlining. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 253–264. CGO '03, IEEE Computer Society (2003)

[17] Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanović, D.: Generating object lifetime traces with Merlin. ACM Trans. Program. Lang. Syst. 28(3), 476–516 (2006)

[18] Hölzle, U., Chambers, C., Ungar, D.: Debugging optimized code with dynamic deoptimization. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. pp. 32–43. PLDI '92, ACM (1992)

[19] Hölzle, U., Ungar, D.: Reconciling responsiveness with performance in pure object-oriented languages. ACM Trans. Program. Lang. Syst. 18(4), 355–400 (1996)

[20] Kell, S., Ansaloni, D., Binder, W., Marek, L.: The JVM is not observable enough (and what to do about it). In: Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages. pp. 33–38. VMIL '12, ACM (2012)

[21] Kotzmann, T., Mössenböck, H.: Escape analysis in the context of dynamic compilation and deoptimization. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. pp. 111–120. VEE '05, ACM (2005)

[22] Kotzmann, T., Mössenböck, H.: Run-Time support for optimizations based on escape analysis. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 49–60. CGO '07, IEEE Computer Society (2007)

[23] Li, W.H., Singer, J., White, D.: JVM-Hosted Languages: They talk the talk, but do they walk the walk? In: Proc. Intl. conf. on Principles and Practices of Programming on the Java platform:Virtual machines, languages, and tools. pp. 101–112. PPPJ '13, ACM (2013)

[24] Libič, P., Bulej, L., Horky, V., Tůma, P.: On the limits of modeling generational garbage collector performance. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. pp. 15–26. ICPE '14, ACM (2014)

[25] Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., Tuma, P.: Introduction to dynamic program analysis with DiSL. Sci. Comput. Program. 98, 100–115 (2015)

[26] Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: A domain-specific language for bytecode instrumentation. In: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development. pp. 239–250. AOSD '12, ACM (2012)

[27] Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Evaluating the accuracy of Java profilers. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 187–197. PLDI '10, ACM (2010)

[28] OpenJDK: Graal project. http://openjdk.java.net/projects/graal/

[29] Oracle: hprof. http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html

[30] Oracle: Netbeans profiler. https://profiler.netbeans.org

[31] Ricci, N.P., Guyer, S.Z., Moss, J.E.B.: Elephant Tracks: Portable production of complete and precise GC traces. In: Proceedings of the 2013 International Symposium on Memory Management. pp. 109–118. ISMM '13, ACM (2013)

[32] Sarimbekov, A., Stadler, L., Bulej, L., Sewe, A., Podzimek, A., Zheng, Y., Binder, W.: Workload Characterization of JVM Languages. Software: Practice and Experience (2015), http://dx.doi.org/10.1002/spe.2337

[33] Sewe, A., Mezini, M., Sarimbekov, A., Ansaloni, D., Binder, W., Ricci, N., Guyer, S.Z.: new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In: Proc. Intl. symp. on Memory Management. pp. 97–108. ISMM '12, ACM (2012)

[34] Sewe, A., Mezini, M., Sarimbekov, A., Binder, W.: Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 657–676. OOPSLA '11, ACM (2011)

[35] Simon, D., Wimmer, C., Urban, B., Duboscq, G., Stadler, L., Würthinger, T.: Snippets: Taking the high road to a low level. ACM Trans. Archit. Code Optim. 12(2), 20:20:1–20:20:25 (2015)

[36] Smith, M.D.: Overcoming the challenges to feedback-directed optimization (keynote talk). In: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization. pp. 1–11. DYNAMO '00, ACM (2000)

[37] Stadler, L., Würthinger, T., Mössenböck, H.: Partial escape analysis and scalar replacement for Java. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 165:165–165:174. CGO '14, ACM (2014)

[38] ej technologies: JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html

[39] The Eclipse Foundation: eclipse tptp. https://eclipse.org/tptp/

[40] Xu, G.: Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. pp. 111–130. OOPSLA '13, ACM (2013)