Observable Dynamic Compilation

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Yudi Zheng

under the supervision of
Prof. Walter Binder

May 2017

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Nathaniel Nystrom	Università della Svizzera Italiana, Switzerland
Prof. Thomas Gross	ETH Zürich, Switzerland
Prof. Andreas Krall	TU Wien, Austria

Dissertation accepted on 11 May 2017

Research Advisor	PhD Program Director
Prof. Walter Binder	**Prof. Michael Bronstein**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Yudi Zheng
Lugano, 11 May 2017

Abstract

Managed language platforms such as the Java Virtual Machine rely on a dynamic compiler to achieve high performance. Despite the benefits that dynamic compilation provides, it also introduces some challenges to program profiling.

Firstly, profilers based on bytecode instrumentation may yield wrong results in the presence of an optimizing dynamic compiler, either due to not being aware of optimizations, or because the inserted instrumentation code disrupts such optimizations. To avoid such perturbations, we present a technique to make profilers based on bytecode instrumentation aware of the optimizations performed by the dynamic compiler, and make the dynamic compiler aware of the inserted code.

We implement our technique for separating inserted instrumentation code from base-program code in Oracle's Graal compiler, integrating our extension into the OpenJDK Graal project. We demonstrate its significance with concrete profilers. On the one hand, we improve accuracy of existing profiling techniques, for example, to quantify the impact of escape analysis on bytecode-level allocation profiling, to analyze object life-times, and to evaluate the impact of method inlining when profiling method invocations. On the other hand, we also illustrate how our technique 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.

Secondly, the lack of profiling support at the intermediate representation (IR) level complicates the understanding of program behavior in the compiled code. This issue cannot be addressed by bytecode instrumentation because it cannot precisely capture the occurrence of IR-level operations. Binary instrumentation is not suited either, as it lacks a mapping from the collected low-level metrics to higher-level operations of the observed program. To fill this gap, we present an easy-to-use event-based framework for profiling operations at the IR level.

We integrate the IR profiling framework in the Graal compiler, together with our instrumentation-separation technique. We illustrate our approach with a profiler that tracks the execution of memory barriers within compiled code. In

addition, using a deoptimization profiler based on our IR profiling framework, we conduct an empirical study on deoptimization in the Graal compiler. We focus on situations which cause program execution to switch from machine code to the interpreter, and compare application performance using three different deoptimization strategies which influence the amount of extra compilation work done by Graal. Using an adaptive deoptimization strategy, we manage to improve the average start-up performance of benchmarks from the DaCapo, ScalaBench, and Octane suites by avoiding wasted compilation work. We also find that different deoptimization strategies have little impact on steady-state performance.

Contents

Chapter 1

Introduction

In this chapter, we motivate the need for accurately observable dynamic compilation (Section 1.1) and give an overview of the contributions of this dissertation (Section 1.2). We also give an outline of the dissertation (Section 1.3) and list the underlying publications (Section 1.4).

1.1 Motivation

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 dynamic optimizing compiler. State-of-the-art dynamic compilers, such as the optimizing compiler in the Jikes RVM [18, 4] or Graal [77], apply online feedback-directed optimizations [91, 7] to the program according to profiling information gathered during program execution.

Although such a profile may not properly characterize the program behavior in the subsequent execution phase, modern dynamic compilers aggressively optimize hot methods by making assumptions on the future program behavior based on the behavior observed so far. In the case where such assumptions turn out to be wrong, the managed runtime system is forced to de-optimize, i.e., to fall back to interpretation (or to code produced by a baseline compiler, respectively). Subsequently, the method may get optimized and compiled again, based on updated profiling information. In long-running programs, most execution time is typically spent in highly optimized compiled code.

Common optimizations performed by dynamic compilers include method inlining [5] and stack allocation of objects based on (partial) escape analysis [22,

93], 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 is not aware of the optimizations performed by the dynamic compiler. Prevailing profilers based on bytecode instrumentation suffer from three serious limitations: (1) *over-profiling* of code that is optimized (and in the extreme case completely removed) by the dynamic compiler, (2) *perturbation* of the compiler optimizations due to the inserted instrumentation code, and (3) *inability* to exactly profile low-level events at the dynamic compiler's intermediate representation (IR) level, such as e.g. deoptimization actions or the execution of memory barriers.

Binary instrumentation is also often adopted in developing profiling tools. One of the challenges to applying binary instrumentation techniques on managed runtime systems is how to identify dynamically compiled code. Prevailing solutions to this problem are twofold. Tools such as DynamoRIO [16] mark the executable memory pages non-writable and intercept any write to these pages. When deploying compiled code, the managed runtime system attempts to modify these pages and thus triggers a previously registered instrumentation callback. The drawback of such a solution is that it lacks the metadata of the compiled code, which is essential for constructing calling-context information. Alternatively, tools such as Pin [67] require a callback within the managed runtime system to intercept the deployment of generated machine code. In both cases, it is difficult to map low-level events (e.g., memory access) to higher-level events (e.g., field access of a particular type) because of information loss. Consequently, the collected profile is often not actionable at the source-code level.

These problems of profiling in the presence of dynamic compilation can be tackled in two ways. On the one hand, to address the aforementioned limitations with over-profiling and perturbation of optimizations in bytecode instrumentation, the dynamic compiler is expected to distinguish between the base-program code and the inserted instrumentation code. The base-program code is compiled in the usual fashion, undergoing the same optimizations as if the inserted code was not there, while the inserted code is adjusted to preserve its purpose with respect to the optimized base-program code.

On the other hand, the gap between bytecode instrumentation and binary instrumentation can be filled by employing instrumentation at the IR level, hence-

forth called *IR instrumentation*. For instance, reasoning about deoptimization behavior in a managed runtime system is impossible via bytecode instrumentation and extremely tedious via binary instrumentation. But it can be easily achieved using IR instrumentation, because a deoptimization action appears as a single IR unit during compilation. Yet, the following challenges need to be addressed: (1) the infrastructure shall maintain a mapping from an IR unit to its originating bytecode; (2) unlike bytecode or machine code, each IR unit has a life cycle, which covers only part of the compilation passes employed by the compiler; (3) the inserted instrumentation code (and its callees) shall not trigger any IR instrumenation (i.e., an IR instrumentation shall only apply to the base-program code); (4) because an IR instrumentation may greatly inflate the emitted machine code, optimizations are needed to improve efficiency and compactness of the inserted code.

1.2 Contributions

In this dissertation, we make two major contributions to profiling in the presence of dynamic compilation: (i) accurate profiling with bytecode instrumentation; (ii) profiling at the IR level.

1.2.1 Accurate Profiling with Bytecode Instrumentation

We define a delimitation API to enable explicit marking of the inserted code such that the dynamic compiler can distinguish between the base-program code and the inserted code. The base-program code is compiled in the usual fashion, undergoing the same optimizations as if the inserted code was not there, while the inserted code is adjusted to preserve its purpose with respect to the optimized base-program code.

We present a solution for a method-based dynamic compiler using a graph-based IR. The dynamic compiler analyzes the IR and identifies the boundaries between the base-program code and the inserted code. We unlink the inserted-code sub-graphs (ICGs) from the IR and keep them separately as IR annotations referencing the base-program nodes that either precede or follow the ICGs in the control-flow graph. We let the dynamic compiler process the IR containing only the base-program code. For each IR node operation, we perform a reconciling operation on the corresponding ICG to preserve its semantics throughout the transformations performed by the compiler. Towards the end of dynamic compilation, the ICGs are spliced back into the base-program IR.

We allow the inserted code to query (and adapt to) the dynamic compiler's decisions. The queries are represented as invocations of special methods that are recognized and handled by the compiler similarly to intrinsics. We call these special methods query intrinsics, and whereas normal compiler intrinsics expand to a sequence of machine instructions, query intrinsics expand to an IR sub-graph with one or more IR nodes. When the compiler encounters a query-intrinsic invocation node (when the ICGs are spliced back into the base-program IR), it executes a corresponding handler function, providing it with the invocation context of the intrinsic. The handler returns an IR sub-graph, which replaces the IR node representing the intrinsic invocation.

We implement the proposed approach in Oracle's Graal compiler [77], and integrate our extension into the corresponding OpenJDK project, such that profiler developers in industry and academia directly benefit from our work. We apply the approach in different scenarios. We present a profiler to explore the impact of (partial) escape analysis and stack allocation on heap usage and object lifetime, demonstrating that our approach helps improve the accuracy of existing bytecode instrumentation-based tools. We present a profiler to study the impact of method inlining considering varying levels of calling context, demonstrating that our approach enables new tools that can help further improve the optimizations performed by dynamic compilers. We also introduce a new framework for testing the results of dynamic compiler optimizations at runtime. Our framework has already helped the developers of Graal to locate and fix performance bugs in their compiler.

1.2.2 Profiling at the IR Level

We present an easy-to-use event-based framework for profiling of IR-level operations. Our framework provides an abstraction for each kind of IR node, namely IR event. We allow the developer to subscribe to an IR event by registering an IR callback method. For each IR node of interest, our framework instantiates the corresponding IR event along with the IR callback method and performs several optimizations of the inserted code. We explain our IR profiling framework using a memory-barrier profiler as running example.

We also implement a deoptimization profiler based on IR profiling. We characterize the deoptimization causes in the code produced by Graal for various benchmark suites using different programming languages. We find that only a small fraction of deoptimization sites is triggered ($\sim$2%), and most cause reprofiling ($\sim$98%). Based on the findings, we implement a conservative deoptimization strategy for the Graal compiler that defers the invalidation of the compiled code

until enough deoptimizations are observed, and an adaptive deoptimization strategy which switches among various deoptimization actions according to a precise deoptimization profile. We evaluate the performance of both deoptimization strategies and compare them to the default strategy used by Graal. The result shows that the conservative strategy, which is based on the inaccurate deoptimization profile collected by the HotSpot VM, may cause extra compilation work due to a first recompilation without reprofiling and a (potential) subsequent recompilation after reprofiling. On the other hand, the adaptive strategy, which relies on a more accurate deoptimization profile, provides benefits for both static and dynamic programming languages.

1.3 Dissertation Outline

This dissertation is structured as follows:

- Chapter 2 discusses the state of the art in the areas of dynamic compilation, deoptimization, intermediate code representation, instrumentation, profiling techniques, as well as profile accuracy and usefulness.

- Chapter 3 introduces our approach to make inserted profiling code explicit to the dynamic compiler. Our approach also allows the inserted code to query runtime path decisions in the optimized compiled code. The new technique enables the collection of accurate profiles that faithfully represent the execution of a base program without profiling (w.r.t. the applied optimizations).

- Chapter 4 introduces an event-based framework for profiling IR-level operations. Our approach also enables the composition of IR-level profiling and bytecode-level profiling.

- Chapter 5 presents an empirical study of the deoptimization behavior in benchmarks executing on a VM using Graal. The chapter also presents two alternative deoptimization strategies and performance evaluation results, comparing the two strategies with Graal's default deoptimization strategy.

- Chapter 6 concludes the dissertation and outlines future research directions inspired by this work.

1.4 Publications

This dissertation is based on the following papers. The work on accurate bytecode profiling (Chapter 3) has been published at OOPSLA '15:

- Y. Zheng, L. Bulej, and W. Binder. Accurate Profiling in the Presence of Dynamic Compilation. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2015), Pittsburgh, PA, USA, October 2015. ACM Press, ISBN 978-1-4503-3689-5, pp. 433–450. Distinguished paper award; paper with artifact.

The empirical study on deoptimization (Chapter 5) has been accepted at ECOOP '17:

- Y. Zheng, L. Bulej, and W. Binder. An Empirical Study on Deoptimization in the Graal Compiler. 31st European Conference on Object-Oriented Programming (ECOOP 2017), Barcelona, Spain, June 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Chapter 2

State of the Art

In this chapter we discuss the the state of the art in the areas of dynamic compilation and deoptimization (Section 2.1), intermediate code representation (Section 2.2), instrumentation (Section 2.3), profiling techniques (Section 2.4), as well as profile accuracy and usefulness (Section 2.5).

2.1 Dynamic Compilation and Deoptimization

Dynamic compilation is adopted in many programming-language implementations [3]. 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 [47]. Subsequently, the code may get optimized and compiled again.

Optimizing dynamic compilers perform online feedback-directed optimizations [7], such as profile-directed inlining of virtual call sites [4, 7, 25, 43, 48]. Self [48] 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.

Dynamic deoptimization as a way to transfer execution from compiled code to the interpreted code was introducted in the Self system to facilitate full source-level debugging of optimized code [47]. Self also introduced techniques such as on-stack replacement, which since then have been adopted and improved by others [82, 33, 58, 52].

In general, deoptimization switches to a less optimized execution mode, i.e., interpreted execution, or execution of machine code generated by a baseline compiler. In Self, deoptimization was primarily used to defer compilation and to execute uncommon code in the interpreter. In the HotSpot VM, especially with Graal enabled, deoptimization represents a key recovery mechanism for speculative optimizations.

Being more interested in the use of deoptimization in the implementation of speculative optimizations, we trace their origins to partial and deferred compilation in Self [20]. To reduce compilation time, program code that was predicted to execute infrequently was compiled only as a stub which invoked the compiler when a particular code path was first executed, thus deferring the compilation of uncommon code paths until they were actually needed. Many of the techniques found in Self, such as adaptive compilation, dynamic deoptimization, and speculative optimizations using deoptimization, were later adopted by Java [82, 58]. Further improvements to the HotSpot VM target selective compilation [59, 19, 60, 61], phase-based recompilation [40], and feedback-directed optimization [101, 7, 110, 94]. Google's V8 JavaScript engine also contains an optimizing compiler that heavily employs speculative optimizations based on the profiled type information [38]. Unlike HotSpot VM, when an assumption in the optimized code fails, V8 bails out into the code generated by a base compiler instead of the interpreter.

Despite the role of deoptimization in the implementation of speculative optimizations, we are not aware of any study that characterizes the actual deoptimization behavior of programs compiled by a speculating dynamic compiler, and the impact of the deoptimizations on the compiled code. This does not mean that deoptimization does not receive any attention. In recent work [102], Wimmer et al. present a VM implementation technique that allows a deoptimization triggered in aggressively optimized code to resume execution in (deoptimized) machine code generated by the same compiler at a different optimization level. In contrast to an interpreter or baseline compiler, both of which rely on a fixed stack-frame layout, using a single compiler allows using an optimized stack layout for both the optimized and deoptimized code. This approach helps reduce the complexity of a VM implementation, because neither an interpreter nor a baseline compiler

are needed.

2.2 Intermediate Code Representations

Intermediate representations (IRs) are used by many modern compilers, as IR graphs are well suited for implementing compiler optimizations via graph transformations, before emitting machine code [3]. In a dynamic compiler, different levels of IR may be applied for different kinds of optimizations. For instance, Jikes RVM's optimizing compiler uses 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 [32], and is extended by introducing nodes that are not necessarily fixed at a specific position in the control flow [23]. 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. To represent ϕ-functions, Click et al. introduce a new node type that has multiple input data values and outputs a single selected data value according to the control-flow edge, namely *Phi* node [23].

2.3 Instrumentation

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 [4]. 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).

Regarding bytecode instrumentations, libraries such as ASM [17], BCEL [24], and Javassist [21] provide APIs for manipulating Java class files. Additionally,

Javassist offers a source code-level API that enables program manipulation without any knowledge of the details of Java bytecode. Soot [99] supports multiple intermediate code representations for analyzing and instrumenting Java source files or class files. WALA [50] integrates various static analyses, which can be employed in their bytecode instrumentation library Shrike and dynamic load-time instrumentation library Dila. RoadRunner [35] and Chord [75] encapsulate Java bytecode instrumentation into events, and allow composition of analyses that intercept these events. AspectJ [54] enables aspect-oriented programming in Java, which can be used for instrumenting Java programs. DiSL [70] is a domain-specific aspect language dedicated to bytecode instrumentation. ShadowVM [69] is an extension of DiSL that enables asynchronous analysis in a separate process.

Concerning binary instrumentation, DynamoRIO [16] is a dynamic code manipulation system that performs binary code transformation while a program is executing. Pin [67] provides a high-level, call-based instrumentation API for intercepting both architecture-independent and architecture-specific events. Valgrind [76] offers built-in shadow value support for constructing heavyweight binary analysis tools.

However, prevailing tools relying of bytecode or binary instrumentation suffer from several limitations. On the one hand, instrumenting the pre-compilation program representation often impairs accuracy of an analysis [53]. On the other hand, instrumenting the post-compilation 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 [8]. 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 compilation phases.

2.4 Profiling Techniques

The importance of contextual information in profiles has been explored since the early 80's. Graham et al. extend the UNIX command *prof* that collects elapsed time per method to the complete call graph [39]. Thanks to the contextual information, they implement the tool *gprof* that enables the propagation of method execution times across call sites. Pettis and Hansen present an application of *gprof*, but slightly change its approach by replacing recompilation with a modified linker for

inserting profiling code [83]. They use the profile to drive the code positioning of the methods and the basic blocks, and to guide the code splitting according to hotness of the code snippets. Both techniques benefit from the locality of reference and help reduce the cache miss rate. As the call graph maintains only one level of calling context, Ammons et al. introduce the *Calling Context Tree* (CCT), which captures multiple levels of calling contexts and does not suffer from the memory overhead of a complete dynamic call tree [1].

For reducing the runtime overhead of instrumentation-based profiling, a profiler may periodically sample instead of collecting data continuously. Whaley presents the *Partial Calling Context Tree* (PCCT) that supports the storage of incomplete calling context information [100]. The PCCT approach applies periodic sampling and incomplete stack traversals, and the author's evaluation confirms reduced runtime overhead. In order to recognize the visited stack frames, the PCCT approach marks a bit in the return address, which requires support from the platform for parsing the return address correctly. Arnold and Sweeney propose a platform-independent implementation named *Approximating Calling Context Tree* (ACCT). ACCT replaces the return address in every stack frame with the address of a profiling method named *trampoline* [9]. After collecting the profile, the trampoline transfers the control back to the caller. Froyd et al. optimize ACCT by inserting the trampoline only to the very top stack frame and shifting it on return [36]. Zhuang et al. apply bursty sampling on ACCT [111]. Their approach predicts the method calls/returns during a burst using a history-based predictor, and applies the prediction to disable redundant bursts. In order to reduce the number of mis-predictions, their approach re-enables a small portion of bursts by an adaptive re-enable ratio.

A more general framework for applying counter-based sampling on instrumentation is introduced by Arnold and Ryder [8]. This framework inserts checks that decrease a global counter and branch to the duplicated instrumented code when the counter reaches a given threshold. The checks are placed on method entries and on loop back edges, such that between checks the program never executes an unbounded number of instructions. To distribute the checks, ensuring an upper bound of executed instructions between two subsequent checks, one can apply Feeley's balanced polling strategy [31]. This strategy will insert checks at arbitrary positions. However, as a modern compiler often inserts thread yield points on method entries and on loop back edges, adding checks at these locations will not significantly perturb the performance of the program. Hirzel and Chilimbi extend this framework to allow collecting samples in bursts before switching back to uninstrumented code [45]. Since the bursty sampling technique can span procedure boundaries, profilers using that technique can collect more accurate

profiles. Arnold and Grove follow the idea of bursty sampling, but they use a timer as the triggering mechanism [6]. Their approach also introduces the sampling *stride*, which results in skipping several sampling points before actually taking every sample. Moret et al. introduce Polymorphic Bytecode Instrumentation (PBI), allowing dynamic dispatch amongst multiple versions of method bodies [72]. Depending on the code version computed upon method entry, the control will be passed to the corresponding method body. The aforementioned sampling frameworks could be implemented using PBI with an extension that supports branching to different code version at arbitrary locations.

Path profiling heavily involves static analyses to reduce runtime overhead. Ball and Larus present an efficient path profiling approach that assigns each path a dedicated number for indexing a counter array, and apply the *maximum spanning tree algorithm* to identify the instrumentation locations [10]. Since the algorithm only works on graphs without cycles, the authors truncate the back edges and divide a single trace into several paths. Bond and McKinley extend this approach by applying a simplified Arnold-Grove bursty sampling [15]. Instead of sampling with a *stride*, their approach only skips the first sampling points and continuously takes a certain number of samples. In addition, the authors propose a smart numbering algorithm, whose goal is to reduce the runtime calculation of the dedicated number for hot paths. They also conduct a profile-guided profiling approach, which applies the profile to determine a hot path, and relocates the instrumented code.

Profiling of languages hosted on managed runtime systems has been well studied. Sweeney et al. modify the Jikes RVM to generate traces of hardware events for Java programs [95]. Hauswirth et al. extend this work by combining software performance monitors at the application, virtual machine, and operating system levels [42]. Their extension enables the reasoning of whole-system performance of programs executed in a virtual machine.

In recent years, concrete profiling tools are implemented via modifying the dynamic compiler of a runtime system. Xu et al. propose approaches to detect high-overhead copy activities [107] and low-utility data structures [108] by modifying the dynamic compiler of IBM's commercial J9 VM [49]. Yan et al. propose a profiling technique to track the propagation of object references [109]. Xu et al. present a tool for finding allocation sites that create reusable data structures [105], and a tunable technique to profile object lifetime [106]. Fang et al. present an approach to amplify memory-related performance problems [30]. All these tools are based on modifications to the dynamic compilers of the Jikes RVM.

2.5 Profile Accuracy and Usefulness

To measure the usefulness of a profiler, researchers discuss about the benefit gained by using the profile. Pettis and Hansen evaluate the impact of code positioning on performance, and the reduction of executed branches [83]. Ball and Larus evaluate path profiling on executed path lengths [10]. Ammons et al. evaluate CCT profiling by applying it to measure the cache misses on paths and procedures [1] in the SPEC95 suite [92].

To measure the accuracy of a profiler, the methodology differs depending on the profiling technique. In the context of sampling profiling, researchers naturally compare the result with the full profile (or perfect profile, exact profile) that is collected by an instrumentation-based profiler [8, 45, 6, 111, 15, 73, 64, 103]. For instance, Arnold and Ryder evaluate the accuracy of an edge profiler using the overlap percentage that accumulates the minimum of the normalized edge counters in the sampling profile and the full profile [8]. These approaches assume that metrics collected by an instrumentation-based profiler are stable across runs such that they can serve as references for the sampling profiles.

In the context of timing profiling, researchers propose different measurement methods on accuracy, because the instrumentation greatly disrupts the timing profile and hence the full profile cannot serve as the baseline. Whaley evaluates PCCT profiling by measuring the correlation between profiles collected in successive runs [100]. Froyd et al. compare the profile result with a low-overhead sampling profiler, assuming that the low-overhead profiler generates a closer match to the actual program behavior [36]. Mytkowicz et al. observe that commonly used Java sampling profilers often disagree with each other, and cannot attribute the increased execution time to the inserted code in their causality analysis [74]. Therefore, they propose the concept of *actionable* profile, meaning that applying the profile yields the expected output, such as blaming the inserted code in the causality analysis, or speeding up real-world applications.

Unlike Mytkowicz's concern about feasibility of measuring profile accuracy, research on the relationship between profile accuracy and profile usefulness questions the correlation between them. Duesterwald and Bala deny the positive correlation between the percentage of profiled flows and the possible performance improvement gained by hot path prediction [29]. Compared to traditional path profiling, they aggressively assume that the *Next Executing Tail* (NET) is a hot path. Their evaluation results confirm that earlier path prediction based on an inaccurate profile may lead to better performance. Langdale and Gross calculate normalized cycle counts as usefulness metric and several accuracy metrics on the basic-block count profiles across runs [62]. They apply the *Spearman Rank*

Correlation Coefficient to measure the correlation between profile accuracy metrics and profile usefulness metric, and the result shows non-significant correlation. Wu et al. conduct a similar study on the correlation between the sampling rate and the feedback-driven program optimization benefits [103]. They analyze the failure and attribute the problem to zero-count errors and inconsistency errors that generally exist in sampling profiling. By statistically rectificating these errors in a profile using a full profile for training, the authors increase the usefulness of the sampled profiles. The above studies quantify the correlation between profile accuracy and the profile usefulness in different aspects.

Apart from Wu's work, Levin et al. propose an approach for fixing the sampling profile off-line [64]. This approach adjusts the edge profile to respect a generalized flow conservation rule that the net flow to a vertex is zero, except for the sources and the sinks. After the adjustment, the sampling edge profile better overlaps the full profile. Regarding online fix for sampling profilers, many approaches introduce randomness on the sampling period such that the correlation between sampling points and the executed code is reduced. Anderson et al. introduce a randomized interrupt period by writing a pseudo-random value into a dedicated performance counter [2]. Binder applies the same idea on Java counter-based sampling profiling, and confirms an increased accuracy [11]. Arnold and Grove skip a random number of sampling points in their bursty sampling framework, to ensure an equal chance on each sampling point [6]. Tian et al. propose randomized inspection-instrumentation to maximize coverage across multiple runs [98]. Mytkowicz et al. blame the bias in existing Java profilers that take samples only at thread yield points, and suggest using standard UNIX signals to pause the application and take samples [74].

Dmitriev proposes JFluid for addressing the huge-overhead problem in an instrumentation-based profiler [27]. This approach modifies the JVM to support dynamic bytecode instrumentation, which enables the injecting and removing of instrumentations on-the-fly. The overhead of an instrumentation-based profiler is reduced by allowing a selective profiling at runtime. Hofer et al. propose partial safepoints and incremental stack tracing for improving the performance of CCT construction in a sampling-based profiler [46]. Compared to a sampling profiler implemented using the Java Virtual Machine Tool Interface (JVMTI) [79], their techniques significantly reduce the overhead without affecting the accuracy of the profiles.

Tian et al. use the optimization level in Jikes RVM [18] for evaluating profile accuracy [98]. They propose a continuous learning framework that applies cross-run profiles stored in a database to guide the dynamic program optimizations. Instead of recompiling methods at a higher optimization level as the original

solution in the Jikes RVM, their framework directly predicts the optimization level and compiles the method. The accuracy is measured by comparing the optimization level prediction and the actual final decision dumped upon program exit.

Chapter 3

Accurate Bytecode Profiling

In this chapter, we present our approach to make inserted profiling code explicit to the dynamic compiler. We start with the motivation of this technique in Section 3.1. In Section 3.2 we present our approach in detail. Section 3.3 and Section 3.4 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 3.5.

3.1 Motivation

Many profiling tools are implemented using bytecode instrumentation techniques. However, profiles produced by these tools are often misleading [74, 53]. 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. 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 [74], 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.

This problem of inaccurate profile has been witnessed by many researchers; for example, tools for object lifetime analysis [44, 84] and for modeling garbage collection behavior based on program traces [66] suffer from significant inaccuracies because they fail to capture the impact of escape analysis and stack allocation. Moreover, using bytecode instrumentation, it is generally impossible to profile the effectiveness of dynamic compiler optimizations.

We introduce a 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. We implement our technique in Oracle's Graal compiler [77] and provide a set of query intrinsics for retrieving the optimization decisions within inserted profiling code.

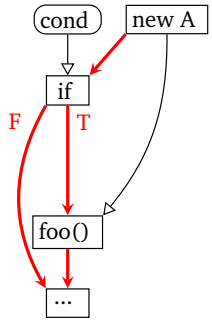
3.2 Approach

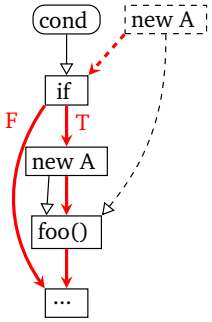
The aforementioned problems with over-profiling and perturbation of optimizations are due to the inability of the dynamic compiler to distinguish between the inserted profiling/analysis code and the base program code, and due to the inability of the inserted code to adapt to the optimizations performed 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 (Subsection 3.2.1), followed by a high-level overview of our approach (Subsection 3.2.2). It comprises several steps which we then present in detail (subsections 3.2.3–

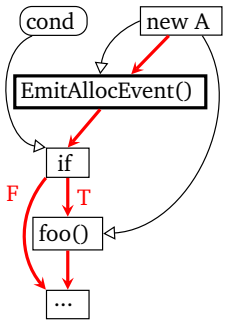
```
A a := new A
if (cond)
  a.foo()
```





(a) Base program (b) Base program IR graph (c) After PEA

```
instrument after new: o ->
  EmitAllocEvent(o)
```



(d) Instrumentation code (e) Instrumentation code

Figure 3.1. Instrumentation intercepting object allocations perturbing partial escape analysis (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.

3.2.6), progressively amending the running example to illustrate the effects of our approach.

3.2.1 Running Example

Consider the snippet of pseudo code in Figure 3.1a. 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 3.1b.

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). One of the employed optimization is escape analysis [13, 14, 22, 56, 57, 71]. 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)* [93] 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 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 3.1c.

If we instrument the program code to trace object allocations using the pseudocode shown in Figure 3.1d, every allocation will be followed by an invocation of the $\mathrm{EmitAllocEvent}()$ method with the newly allocated object as an argument. The corresponding IR is shown in Figure 3.1e. 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. To enable instrumentation that intends to intercept actual occurred program behavior, we would also want the $\mathrm{EmitAllocEvent}()$ method to follow the movement of the allocation into the then-branch of the conditional.

3.2.2 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.

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.

3.2.3 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 3.1. 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

[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.

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 3.1. Delimitation API methods for explicit marking of inserted profiling code.

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 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 3.2, 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 4–8). 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 9–12). 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

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 3.2. 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.

respective sets (Line 13–15). We then remove ICG nodes and ICG edges from the base program IR graph (Line 16–31). To preserve a valid CFG, we reconnect the predecessor and the successor nodes of the ICG, either directly (Line 19–24) or via a placeholder node created for an ICG (Line 25–29). Finally, we add a tuple representing an ICG into a set of ICGs (Line 32).

To put the concepts presented so far into the context of our example, consider again the program snippet shown in Figure 3.1a. The original instrumentation code from Figure 3.1d is now surrounded by invocations of the delimitation API methods, as shown in Figure 3.3a. In contrast to Figure 3.1e, the corresponding IR graph in Figure 3.3b shows the instrumentation as an ICG associated with the allocation node preceding the ICG in the base program IR.

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

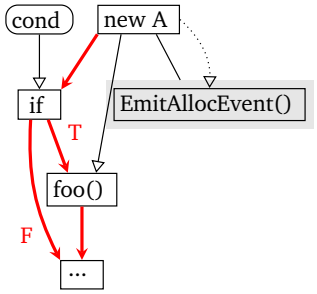
Algorithm 1: Extract ICGs from the base program IR graph.

3.2.4 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

```
instrument after new: o ->
  instrumentationBegin(PRED)
  EmitAllocEvent(o)
  instrumentationEnd()
```

(a) Enhanced instrumentation



(b) Extracted ICG

Figure 3.3. 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.

operations, such as node elimination, value replacement, expansion, cloning, and movement. These node operations are sufficient to implement all important dynamic compiler optimizations. If the compiler employs other node operations, additional reconciling operation needs to be defined. 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).

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 [90], 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 8–14).

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:* **do**
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* $n \rightarrow n_r$*:* **do**
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* $n \rightarrow$ *subgraph with entry* b_{sub} *and exit* e_{sub}*:* **do**
9 **foreach** $\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I} \mid a_I = n$ **do**
10 **if** $p_I = PRED$ **then**
11 $\mathbb{I} \leftarrow (\mathbb{I} - \langle a_I, p_I, N_I, C_I, D_I, W_I\rangle) \cup \{\langle e_{sub}, p_I, N_I, C_I, D_I, W_I\rangle\}$
12 **else if** $p_I = SUCC$ **then**
13 $\mathbb{I} \leftarrow (\mathbb{I} - \langle a_I, p_I, N_I, C_I, D_I, W_I\rangle) \cup \{\langle b_{sub}, p_I, N_I, C_I, D_I, W_I\rangle\}$
14 **end**
15 **case** *cloning of* $n \rightarrow n_c$*:* **do**
16 **foreach** $\langle a_I, p_I, N_I, C_I, D_I, W_I\rangle \in \mathbb{I} \mid a_I = n$ **do**
17 $N_I' \leftarrow$ clone N_I with mapping function $\mathbf{map}_{clone}$
18 $C_I' \leftarrow \bigcup_{\langle u,v\rangle \in C_I} \{\langle \mathbf{map}_{clone}(u), \mathbf{map}_{clone}(v)\rangle\}$
19 $D_I' \leftarrow \bigcup_{\langle u,v\rangle \in D_I} \{\langle \mathbf{map}_{clone}(u), \mathbf{map}_{clone}(v)\rangle\}$
20 $W_I' \leftarrow \bigcup_{\langle u,v\rangle \in W_I} \{\langle u, \mathbf{map}_{clone}(v)\rangle\}$
21 $\mathbb{I} \leftarrow \mathbb{I} \cup \{\langle n_c, \mathbf{map}_{clone}(p_I), N_I', C_I', D_I', W_I'\rangle\}$
22 **end**
23 **case** *movement of n:* **do**
24 // nothing to be done
25 **end**

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

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 15–22).

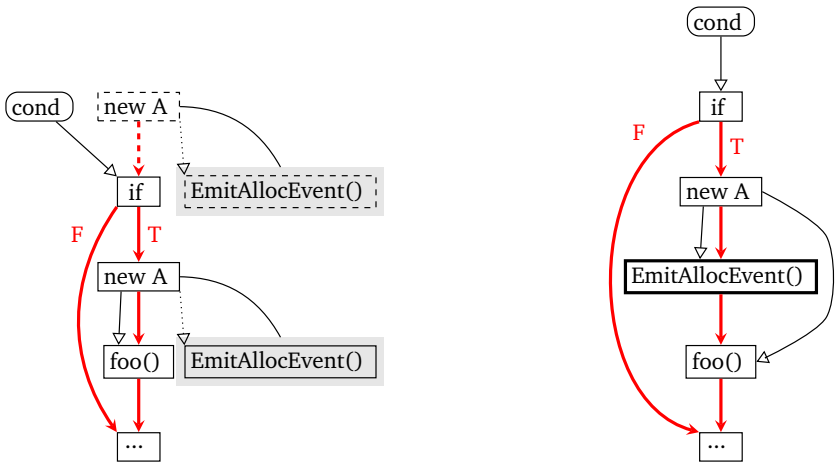
Node movement. Relocates a node to a different location in the CFG. This operation usually follows a cloning operation, because the clone needs to be moved to a new location. It can be used as a standalone operation to implement,

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 3.2. Query intrinsics available to the developer of bytecode instrumentation.

e.g., loop-invariant code motion. Node movement is a change 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 3.3b 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



(a) After reconciling (b) Final IR graph

Figure 3.4. 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.

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 3.4a.

3.2.5 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

1 **function** *Evaluate*(n_I, a_I)
  return a subgraph $\langle res, N_{sub}, D_{sub} \rangle$ representing the evaluation result of the query intrinsic n_I. res denotes 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.
2
3
4 **predicate** *NodeIsAvailable*(u, v)
5 **return true** if u has not been eliminated and can be scheduled before v.

Algorithm 3: Sub-routines used in the splicing algorithm (Algorithm 4).

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 3.2.

3.2.6 Splicing ICGs

Towards the end of the dynamic compilation, we splice the ICGs back into the base program IR, as shown in Algorithm 4. For each ICG, we first evaluate all query intrinsics and replace the corresponding nodes with the resulting IR subgraph (Line 3–8). We then remove the invocations of the delimitation API methods (Line 9–11), 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 12–14), before the associated node (Line 15–17), or in place of the associated

```
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} ∪ {⟨u,v⟩ | ⟨u,n_I⟩ ∈ C_I ∧ ⟨n_I,v⟩ ∈ C_I}
7        D_I ← {⟨u,v⟩ ∈ D_I | u ≠ n_I} ∪ {⟨n_eval,v⟩ | ⟨n_I,v⟩ ∈ D_I} ∪ D_eval
8      end
9      let b_I = callsite in N_I invoking instrumentationBegin
10     let e_I = callsite in N_I invoking instrumentationEnd
11     N ← N ∪ (N_I − {b_I, p_I, e_I})
12     if p_I = PRED then
13       C ← {⟨u,v⟩ ∈ C | u ≠ a_I} ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
14             ∪ {⟨a_I,v⟩ | ⟨b_I,v⟩ ∈ C_I} ∪ {⟨u,v⟩ | ⟨u,e_I⟩ ∈ C_I ∧ ⟨a_I,v⟩ ∈ C}
15     else if p_I = SUCC then
16       C ← {⟨u,v⟩ ∈ C | v ≠ a_I} ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
17             ∪ {⟨u,a_I⟩ | ⟨u,e_I⟩ ∈ C_I} ∪ {⟨u,v⟩ | ⟨u,a_I⟩ ∈ C ∧ ⟨b_I,v⟩ ∈ C_I}
18     else // p_I = HERE
19       N ← N − {a_I}
20       C ← {⟨u,v⟩ ∈ C | u ≠ a_I ∧ v ≠ a_I} ∪ {⟨u,v⟩ ∈ C_I | u ≠ b_I ∧ v ≠ e_I}
21             ∪ {⟨u,v⟩ | ⟨u,a_I⟩ ∈ C ∧ ⟨b_I,v⟩ ∈ C_I} ∪ {⟨u,v⟩ | ⟨u,e_I⟩ ∈ C_I ∧ ⟨a_I,v⟩ ∈ C}
22
23     D ← D ∪ {⟨u,v⟩ ∈ D_I | v ≠ b_I}
24     foreach ⟨u,v⟩ ∈ W_I do
25       if NodeIsAvailable(u,v) then
26         D ← D ∪ {⟨u,v⟩}
27       else
28         d ← the default value of type of u
29         N ← N ∪ {d}
30         D ← D ∪ {⟨d,v⟩}
31     end
32   end
```

Algorithm 4: Splice ICGs into the base program IR graph.

placeholder node (Line 18–22). Finally, we convert the *weak data-flow edges* back to normal data flow edges (Line 23–30). 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 27–30). For example, if the instrumentation intends to count "new" bytecode but at the same time constructs

a data dependency towards the allocated instance, the eventual object may be stack allocated and hence replaced with the "null" value.

Coming back to our running example, the result of splicing the ICGs back into the base program IR is shown in Figure 3.4b, with the invocation of the $\mathrm{EmitAllocEvent}()$ method relocated to the then-branch of the conditional.

3.3 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 (Subsection 3.3.1), object-lifetime profiling (Subsection 3.3.2), and callsite profiling (Subsection 3.3.3).

3.3.1 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 [81], Eclipse TPTP [97], JProfiler [96], and hprof [78] 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 overprofiling [74, 53]. An allocation hotspot profiler may thus draw attention to places with high allocation rates (or amounts of allocated memory) which in reality may have only a negligible impact on the GC overhead.

Also, previously published results on workload characterization [87, 65, 85] 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

```
1 instrument after new: o ->
2   instrumentationBegin(PRED)
3   if (isMethodCompiled())
4     EmitHeapAllocEvent()
5   else
6     EmitInterpreterAllocEvent()
7   instrumentationEnd()
8
9   instrumentationBegin(HERE)
10  EmitBytecodeAllocEvent()
11  instrumentationEnd()
```

Figure 3.5. Pseudo-code of the instrumentation used by the allocation profiler to track object allocations.

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 3.5, 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 [12] 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 3.6, with the actual values shown in Table 3.3. During the startup iteration,

[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 the eclipse benchmark 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 3.5 for a discussion of performance issues).

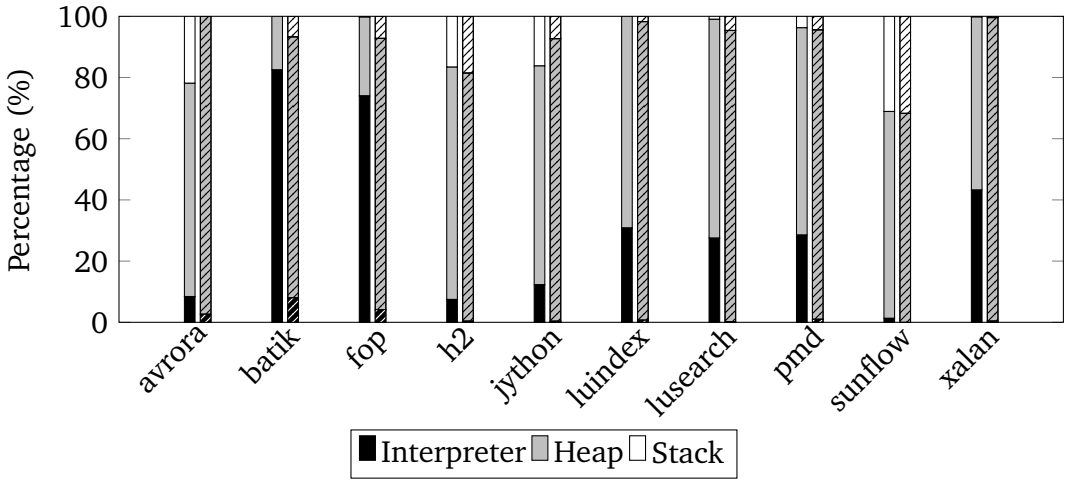


Figure 3.6. Proportions of allocation types for the 1st (without ▨ pattern) and the 15th (with ▨ pattern) benchmark iteration.

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

in which the benchmark code is in the least optimized form, the proportion of stack allocations ranges from zero (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 3.3 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".

3.3.2 Impact on Object Lifetime Analysis

Another application of allocation profiling is to collect information on memory-related behavior of programs, which helps in GC algorithm design and development. For instance, the Merlin algorithm [44] 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 [106]. For other VM implementations, researchers typically use instrumentation-based dynamic analysis tools such as ElephantTracks (an implementation of the Merlin algorithm) [84].

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

[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%.

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.

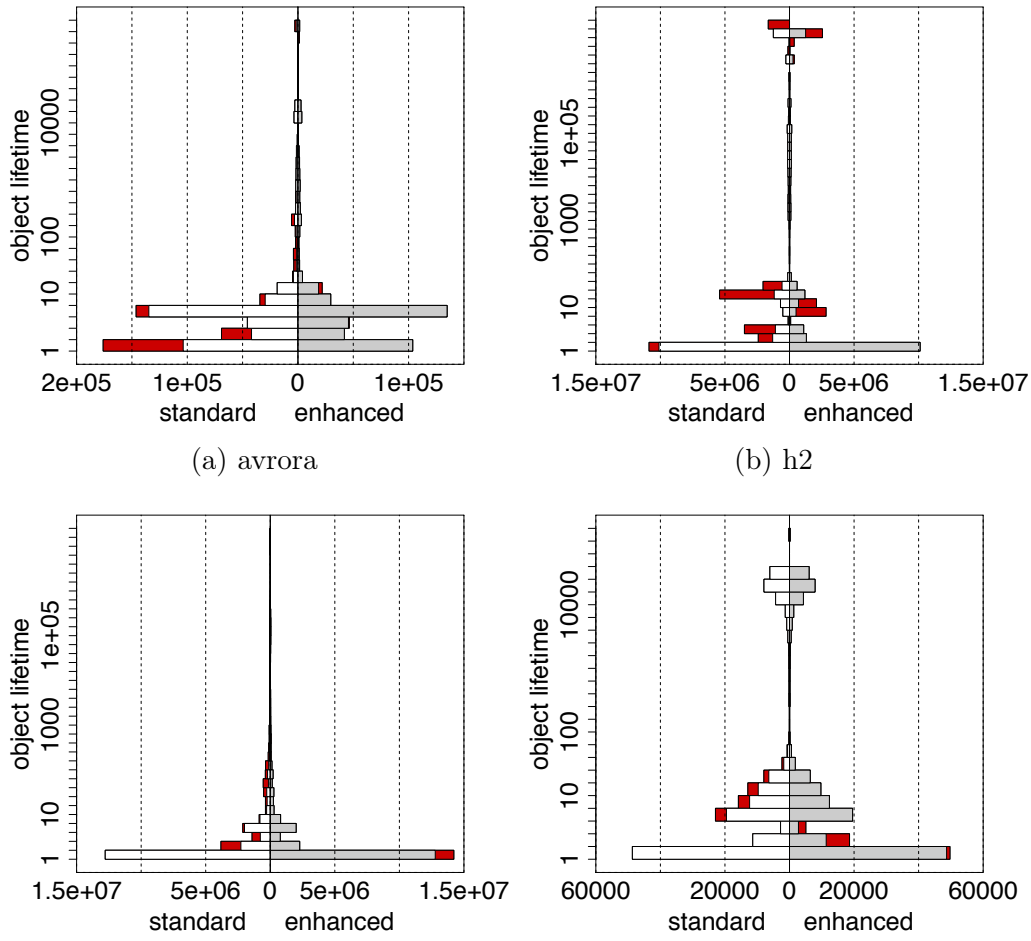
In performance engineering research, the inaccurate allocation information and the differences in GC behavior in response to slight changes in GC-relevant workload make it extremely difficult to model and predict the impact of GC on application performance [66]. While there is a need for accurate object-lifetime 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 [93], 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 3.2.

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

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



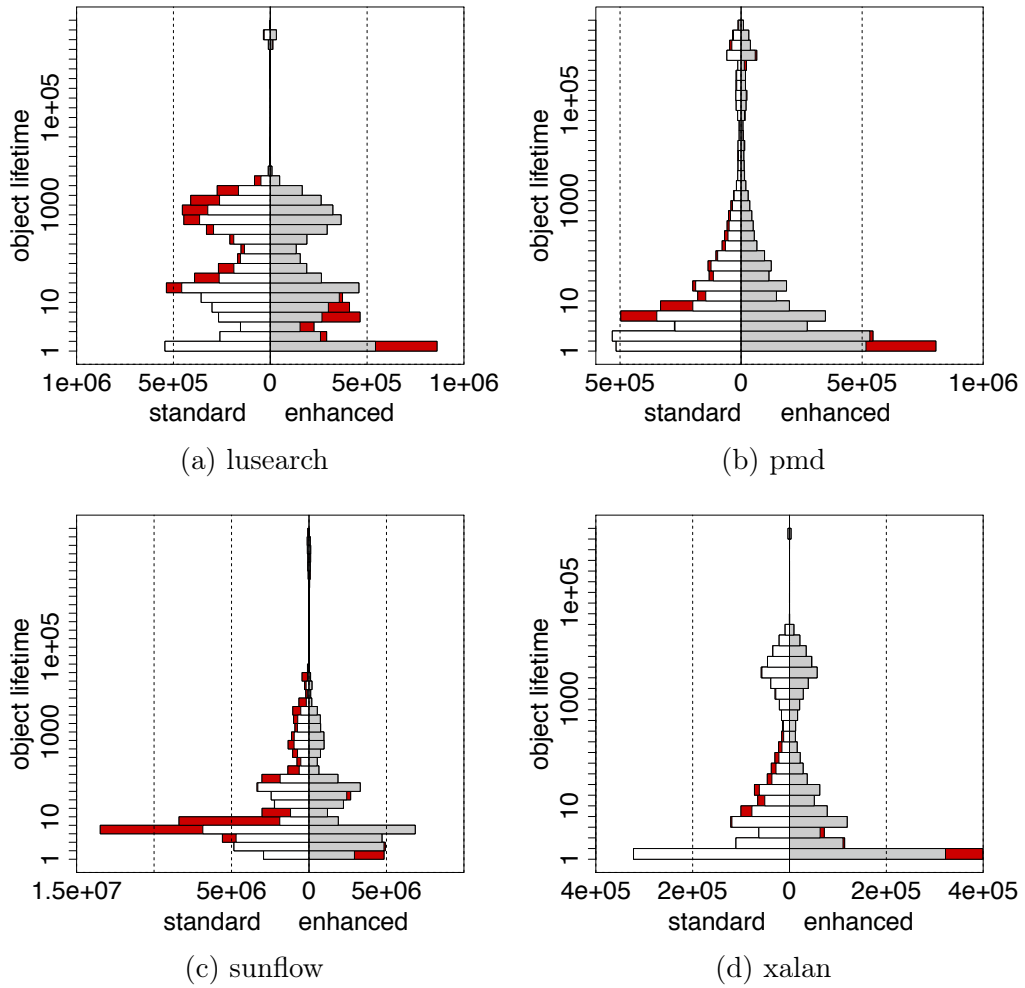
(a) avrora (b) h2

(c) jython (d) luindex

Figure 3.7. 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.

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].

[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 [84].



(a) lusearch (b) pmd

(c) sunflow (d) xalan

Figure 3.8. Back-to-back histogram of object lifetime distributions for the first benchmark iteration (continued). The results for the batik and the fop benchmarks are excluded due to less significant differences.

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 3.7 and Figure 3.8 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 [93]. 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 [66]. Using the enhanced instrumentation allows obtaining accurate information about the memory-related behavior of a program, thus improving simulation results.

3.3.3 Impact on Callsite Profiling

Dynamic compilers in modern VM implementations aggressively inline methods at hot call sites [4, 7, 25] to eliminate the method invocation overhead, to expand the scope for other intraprocedural optimizations, and to enable specialization of the inlined code [48]. 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 [88, 85] 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. [88] 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 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 Subsection 3.3.1 using the two compiler variants[9]. We report results for the 15th iteration (steady state) in Table 3.4, 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

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

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 3.4. 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.

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 [88, 65, 85].

3.4 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 (Subsection 3.4.1). 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 (Subsection 3.4.2). In the third case study, we present a compiler testing framework (Subsection 3.4.3) relying on the ability to observe program behavior at the level of compiled code.

3.4.1 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 Subsection 3.3.3. We filtered 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 3.5. 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

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

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

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	Comp...Maker.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 3.5. 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).

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 3.5 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.

3.4.2 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

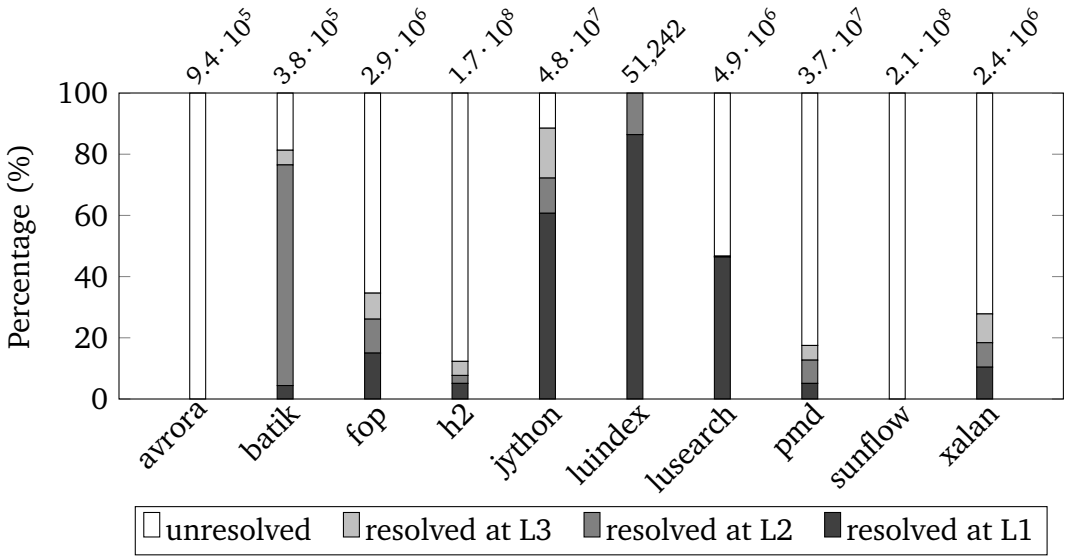


Figure 3.9. The percentages of invocations at non-inlined call sites for which the receiver type can be resolved using 1, 2, or 3 levels of calling context. Above each bar is the total number of invocations at non-inlined callsites.

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 3.9.

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.

Consistent with these findings, the most significant peak performance speedup

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

in a recent trace-based JIT compiler for Java was achieved for jython (factor of 1.59) [41]. 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.

3.4.3 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.

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 3.10. 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 3.2) directly. For test

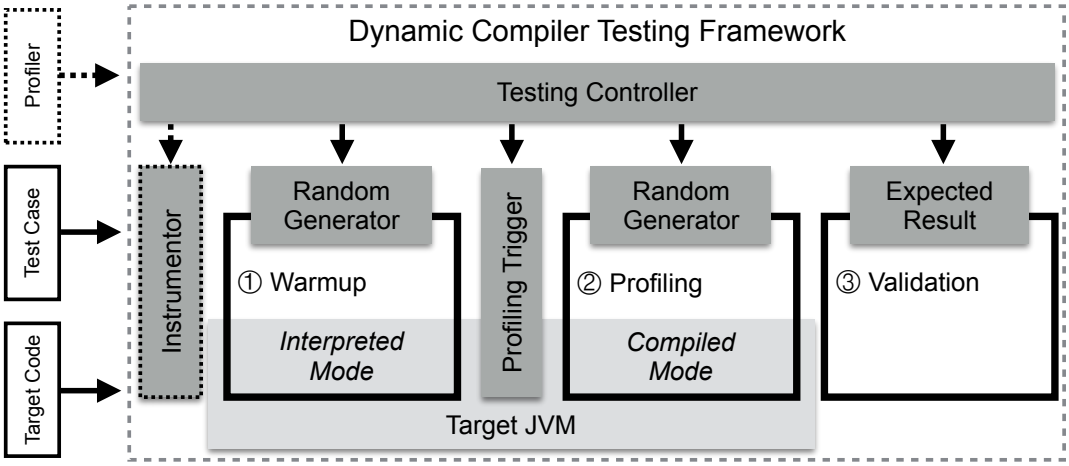


Figure 3.10. Overview of the dynamic compiler testing framework

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. 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

```
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   /** Returns TRUE with the given probability. */
8   protected boolean likely(double probability) { ... }
9   /** Warmup operation. */
10  protected abstract void warmup();
11  /** Returns TRUE if more warmup is needed. */
12  protected abstract boolean isWarmedUp();
13 }
14
15 /** The test case: Partial Escape Analysis. */
16 class PEATestCase extends JITTestCase {
17
18   static final double PROB = 0.8;
19   static final int ITERATIONS = 10000;
20   static final double EPSILON = 0.02;
21   static int cnt = 0;
22
23   protected void warmup() { A.foo(likely(PROB)); }
24   protected boolean isWarmedUp() { return cnt > 0; }
25
26   @Test
27   public void testPartialEscape() {
28     cnt = 0;
29     for (int i = 0; i < ITERATIONS; i++) {
30       A.foo(likely(PROB));
31     }
32     assertEquals(((double) cnt)/ITERATIONS, PROB, EPSILON);
33   }
34 }
35
36 /** The target code. */
37 class A {
38
39   static void foo(boolean invokeBar) {
40     A a = new A();
41     DelimitationAPI.instrumentationBegin(PRED);
42     if (CompilerDecision.isMethodCompiled()) {
43       PEATestCase.cnt++;
44     }
45     DelimitationAPI.instrumentationEnd();
46     if (invokeBar) { a.bar(); }
47   }
48   /** This method will not be inlined. */
49   void bar() { ... }
50 }
```

Figure 3.11. Example of a simple Partial Escape Analysis test.

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 3.11 shows the abstract base class providing the necessary support for the dynamic compiler test cases (Line 1–13), the actual test case (Line 15–34), and the target code (Line 36–50) with the inlined profiling code surrounded by invocations of delimitation API methods (Line 41–45), 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. 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.

3.5 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

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

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

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 3.11.

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 3.3, where the ability to accurately observe the allocation behavior of applications running on a state-of-the-art VM is extremely important, e.g., for modeling the impact of garbage collection on application performance [66].

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 [28]. 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 [70, 68] instrumentation framework to automatically insert the necessary delimitation API invocations.

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.

3.6 Summary

In this chapter 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 optimizations 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.

Chapter 4

Intermediate-Representation Profiling

In this chapter, we present our approach to profile IR-level operations. We start with the motivation of this approach in Section 4.1. We then discuss the general technical challenges and the framework design in Section 4.2. We present our approach in Section 4.3 and demonstrate its applicability in Section 4.4. An assessment of the strengths and limitations of our approach can be found in Section 4.5.

4.1 Motivation

Instrumentation is a commonly adopted technique for developing profilers. Prevailing instrumentation techniques targeting programs running on the JVM can be divided into bytecode instrumentation and binary instrumentation. Yet, both techniques suffer from limitations that cannot be easily addressed.

On the one hand, tools based on bytecode instrumentation cannot precisely capture the occurrence of IR-level operations such as memory barriers or deoptimizations. One may mimic the profiling of IR-level operations by intercepting their originating bytecodes. For example, the FastTrack data-race detector instruments each volatile field-access bytecode to emit memory-barrier events [34]. It addresses the problem of false positives of prevailing data-race detectors such as Eraser [86] on programs that rely on barrier synchronization. With the assistance of our accurate bytecode profiling technique presented in Chapter 3, the inserted code will be adapted to compiler optimizations. However, this approach may result in an incomplete profile, as it is not possible to intercept IR-level operations that do not have an associated originating bytecode. For example, the compiler

may insert a memory barrier upon exit of an inlined constructor, for ensuring safe publication. Another example is the triggering of deoptimization, which does not correspond to any specific bytecode pattern.

On the other hand, tools based on binary instrumentation often lack a mapping from the collected profile of program behavior at the binary level to higher-level operations. Because of various compiler optimizations, the emitted binary code may be significantly different from the input bytecode in terms of code structure. Moreover, information may be lost when lowering a high-level operation to a lower-level operation. For example, it is difficult to map a memory access to the access of a field of a particular type. As a consequence, the resulting profile is often not actionable at the source-code level.

To fill the gap between bytecode instrumentation and binary instrumentation, we introduce a technique to profile IR-level operations used during dynamic compilation. We present an easy-to-use event-based framework, which allows a profiler to register callback methods that will be invoked when the emitted code of the IR-level operation is executed. Our technique enables the combination of IR-level profiling with bytecode-level instrumentation. We implement our approach to IR-level profiling in Oracle's Graal compiler, together with the accurate bytecode-level profiling technique discussed in Chapter 3.

4.2 Framework Design

Before presenting the details of our IR profiling framework, we discuss several challenges that need to be addressed.

Life cycle of IR nodes. Unlike bytecode or machine code that are static and fixed, an IR node has a life cycle during compilation, because it can be eliminated or replaced by other nodes. A general observation is that an IR node representing a high-level concept will be lowered to a subgraph consisting of numerous IR nodes. Our accurate profiling technique (Chapter 3) presents a solution for tracking node elimination or replacement. To complement this technique, our infrastructure performs the instrumentation at the last optimization phase within the life cycle of an IR node. During instrumentation, we attach the inserted code to an individual IR node as an ICG, and rely on the accurate profiling technique to handle the reconciliation and inlining of the ICG.

Re-entrance of the inserted code. Because the inserted code is subject to compilation before instrumentation, its own IR representation may also contain the targeted operation. This issue can be addressed by not instrumenting the inserted

code itself. However, the inserted code may also invoke other methods. If such a method is compiled and instrumented, it may result in an infinitely recursive invocation of the inserted code. Our infrastructure provides a dedicated mechanism to prevent such infinite recursions.

Efficiency and compactness of the inserted code. If the target operation is common, the emitted machine code may be greatly inflated due to the instrumentation. Thus, efficiency and compactness of the inserted code has to be taken into consideration. While offering freedom for the profiler developers to write arbitrary instrumentations, our infrastructure applies various optimizations, including escape analysis, partial evaluation, and dead code elimination on the inserted code. In many cases, these optimizations significantly reduce the size of the inserted code.

While these general challenges have to be addressed in the implementation, we now focus on desired properties of the programming model for IR profiling. Below, we motivate the main design choices besides using IR-level operations as profiling targets.

High-level programming model. Although targeting IR-level operations, we aim at a high-level programming model for describing IR instrumentations, to accelerate the development of new profilers. That is, the developer can profile the operation represented by an IR node, without the knowledge of the IR node per se. To this end, our infrastructure offers the same level of abstraction as the IR in the form of simply events (having a small, easy-to-use interface) that are decoupled from the corresponding IR nodes. Context information specific to the IR-level operations is included within the events. For subscribing to an event, the profiler developer only needs to designate a callback method in the profiler.

Association of an IR event with its originating bytecode. A key benefit that differentiates IR-level profiling from binary instrumentation is the ability to trace an IR back to its originating bytecode (if any). Such mapping information is available via tracking node transformation during dynamic compilation, but is discarded in the emitted binary code. The originating bytecode can be useful for locating the hotspot of certain IR-level operations in the bytecode. For all IR-level events, our infrastructure provides access to the originating bytecode. In case of IR-level operations inserted by the dynamic compiler, a predefined constant indicates the absence of an associated bytecode.

Composition of bytecode-level and IR-level profiling. By invoking the same profiler both from a bytecode instrumentation and from a callback method for IR profiling, we enable the composition of two instrumentation techniques within one

profiler. For example, bytecode-level profiling can be used to maintain a calling-context representation to store counters for the observed IR-level operations in a calling-context-sensitive manner. Even though the two techniques are decoupled, the bytecode instrumentation may perturb IR-level profiling, if the dynamic compiler cannot distinguish the code inserted by a bytecode instrumentation from the base-program code. Our framework guarantees that if a tool applies the accurate bytecode profiling technique presented in Chapter 3, it will avoid IR-level profiling within all inserted instrumentation code.

4.3 Approach

In this section we present our approach to profiling at the IR level in detail. We start with the programming model of our event-based IR profiling framework (Subsection 4.3.1), followed by an explanation of its implementation (Subsection 4.3.2). We then present the architecture of our approach (Subsection 4.3.3). Finally, we discuss optimizations on the IR instrumentation (Subsection 4.3.4).

4.3.1 Programming Model

The key element of our event-based IR profiling framework is *IR event*, which can be subscribed by a profiler to get notified of the occurrence of certain IR-level operations. The IR event also serves as a data container for passing VM-internal or compiler-internal information to the profiler. In our programming model, IR events are standard Java classes implementing the $\mathrm{IREvent}$ interface. The internal context information is then stored in instance fields. $\mathrm{IREvent}$ also offers methods to access the originating method and bytecode index of the corresponding IR-level operation.

We create an IR event for each IR node type in the Graal compiler. Table 4.1 summarizes a selective list of IR events, along with their context information. As a running example, Figure 4.1b presents the $\mathrm{MemoryBarrierEvent}$ class. It contains an instance field of type int for indicating different kinds of barriers.

To be able to profile an IR event, the profiler needs to designate an *IR callback*. The IR callback is a static method annotated with $\mathrm{@IRCallback}$. Its method body is used as a template that is instantiated and inlined at each corresponding IR node. Any profiling routine that the IR callback invokes should be included in the bootstrap classes, to ensure the routine's visibility in any other package, including those from the Java class library. The target IR event is specified in the event attribute in the $\mathrm{@IRCallback}$ annotation. The IR callback may include the IR

Event	*Description*	*Field*
Control: Split&Merge		
IfEvent	Split into two paths.	
MergeEvent	Merge of multiple paths.	
SwitchEvent	Switch amongst multiple paths.	
Control: Invocation		
InvokeEvent	Invocation of a Java method. The "descriptor" field indicates the identifier of the method.	descriptor: String
NativeCallEvent	Invocation of a native method. The "descriptor" field indicates the identifier of the method.	descriptor: String
Control: Loop		
LoopBeginEvent	Entrance of a loop.	
LoopBackEdgeEvent	Back-edge of a loop.	
LoopExitEvent	Exit of a loop.	
Control: Runtime		
DeoptimizationEvent	Deoptimization. The "action" field encodes the deoptimization action and the "reason" field encodes the deoptimization reason.	action: int reason: int
SafePointEvent	Safe point.	
Data: Allocation		
NewArrayEvent	Array allocation. The "elemType" field indicates the element type of the allocated array.	elemType: Class
NewInstanceEvent	Instance allocation. The "type" field indicates the type of the allocated instance.	type: Class
Data: Consistency		
ReadAndAddEvent	Atomic read&add operation.	
ReadAndWriteEvent	Atomic read&write operation.	
CompareAndSwapEvent	Atomic compare&swap operation.	
MemoryBarrierEvent	Memory barrier. The "barrierKind" field encodes the type of the memory barrier.	barrierKind: int
MonitorEnterEvent	Entrance of a monitor. The "lock" field indicates the object on which an exclusive lock is obtained.	lock: Object
MonitorExitEvent	Exit of a monitor. The "lock" field indicates the object on which an exclusive lock is released.	lock: Object

Table 4.1. A selective list of IR events in our framework. The third column corresponds to the internal context information carried by the IR event.

```
1 public interface IREvent {
2   default String originatingMethod() { ... }
3   default int originatingBytecodeIndex() { ... }
4   ...
5 }
6
7 @Target(ElementType.METHOD)
8 public @interface IRCallback {
9   Class<? extends IREvent> event();
10 }
```

(a) Definition of IREvent and @IRCallback.

```
1 public class MemoryBarrierEvent implements IREvent {
2   private int barrierKind;
3   public int getBarrierKind() { return barrierKind; }
4   ...
5 }
```

(b) Definition of MemoryBarrierEvent.

```
1 public class MemoryBarrierProfiler {
2   @IRCallback(event = MemoryBarrierEvent.class)
3   static void callback(MemoryBarrierEvent e) {
4     profile(e.getBarrierKind());
5   }
6   static void profile(int barrierKind) {
7     // profile number of barriers per kind
8   }
9 }
```

(c) A profiler subscribing to MemoryBarrierEvent.

Figure 4.1. MemoryBarrierEvent and a profiler subscribing to it.

event as formal parameter and access the contents of the IR event within its body. The specified formal parameter will be type-checked by our framework to ensure the validity of the IR callback.

Figure 4.1c demonstrates a simple memory-barrier profiler that declares an IR callback. The callback method is annotated with @IRCallback that is parameterized by MemoryBarrierEvent. This indicates that the profiler is listening to MemoryBarrierEvent. Furthermore, the IR callback receives a formal parameter of type MemoryBarrierEvent. The barrier kind carried by the input event instance is then passed to a profiling method.

Even though the developer may freely compose IR callbacks, there are two general guidelines that help improve accuracy and performance of an IR profiler. Firstly, each IR callback has to be thread-safe. Because the IR event can be triggered from multiple threads, the developer needs to employ thread-local data

structure, concurrent data structure, or proper synchronization to ensure the correctness of the profile. Secondly, the IR callback should be short. Otherwise, if the targeted IR-level operation occurs frequently, the emitted machine code would be significantly inflated by the instrumentation, increasing compilation time and occupying a larger part of the limited code cache. If the profiler contains complex operations, a good practice is to create a separate profiling method instead of squeezing all the code into the IR callback. However, for simple counter-based profilers, it is more beneficial to embed the incrementing of the counter within the IR callback, so as to avoid the overhead of extra method invocations and bypass activations in the callbacks.

With respect to the composition of IR-level and bytecode-level profiling, while these techniques offer different mechanisms for emitting events, they can share the same profiler backend. Hence, one scenario is to use IR-level profiling for updating custom software performance counters, and to use bytecode-level profiling as the consumer of those performance counters.

The composition of both profiling techniques reveals two issues concerning the accuracy of the profile. Firstly, the code inserted by bytecode instrumentation as well as its callees are subject to IR-level profiling, which may result in over-profiling. To tackle this issue, we apply the mechanism for bypassing IR-level profiling on the bytecode instrumentation marked by the delimitation API methods from Table 3.1. Implementation details will be discussed in Subsection 4.3.2. Secondly, one may apply both profiling techniques to emit the same kind of events to achieve full code coverage both in the interpreter and in the compiled code. Consequently, the compiled code may emit redundant events both from the bytecode instrumentation and from the IR-level instrumentation. The use of the $\mathrm{isMethodCompiled}$ intrinsic presented in Chapter 3 allows one to avoid emitting bytecode-level events in the compiled code.

4.3.2 Implementation

While the IR-event definitions are decoupled from the concrete IR, the underlying implementations heavily depend on the dynamic compiler and its IR. We hereby base the discussion on a graph-oriented IR in SSA form, with optimizations implemented as IR graph transformations.

To provide library support for an IR event, our framework demands a specification that defines the associated IR node type and the instantiation of the IR event. The associated IR node type is used both for selecting instrumentation locations and for determining the timing of the instrumentation. As discussed previously, each IR node type has its own life cycle. Therefore, we maintain a

```
1 // Instantiation of the IREvent
2 MemoryBarrierEvent newEvent = new MemoryBarrierEvent();
3 newEvent.barrierKind = ... // The init value is provided by
4                            // the corresponding specification.
5 // Inlined IRCallback
6 MemoryBarrierProfiler.profile(newEvent.getBarrierKind());
```

Figure 4.2. Pseudo code of the instrumentation.

mapping from each IR node type to the last optimization phase within its life cycle. Our framework inserts an instrumentation before the mapped phase.

The instrumentation inserts a subgraph at each IR node that is associated with the target IR event. The inserted code consists of two parts: the instantiation of the IR event and the method body of the IR callback. If applicable, the formal argument of the IR callback (i.e., the event instance) will be type-checked to ensure consistency with the associated event. Upon creation of the instrumentation template, any reference to the formal parameter in the IR callback will be replaced by the allocated IR event. For example, Figure 4.2 illustrates the pseudo code of the instrumentation derived from the IR callback in Figure 4.1c. The reference to the formal argument e in the IR callback is replaced by the allocated $\mathrm{newEvent}$.

Our framework avoids recursive entrance of the callback method using a per-thread "bypass" flag [72]. If we detect a method invocation in the IR callback that cannot be inlined, we activate the bypass by wrapping the inserted code with a test against the value of the bypass flag. This flag is set before executing the inlined IR callback and cleared afterwards. Consequently, code executed in the IR callback will not trigger any callback. The bypass flag is also externally accessible, for bypassing the IR callback in certain situations such as for code inserted by a bytecode instrumentation.

Our framework reuses the ICG-related techniques described in Chapter 3. During IR instrumentation, we attach an ICG (which is transformed from the IR callback preceded by the IR-event instantiation code) to the target IR node. In general, because the instrumentation is to indicate the successful execution of the target IR node, we create an ICG parameterized with PRED. In the cases of jump nodes or termination nodes such as deoptimization, we create an ICG parameterized with SUCC. Once created, the ICG will be reconciled according to the compiler optimizations and eventually spliced back into the base-program IR, as presented in Chapter 3.

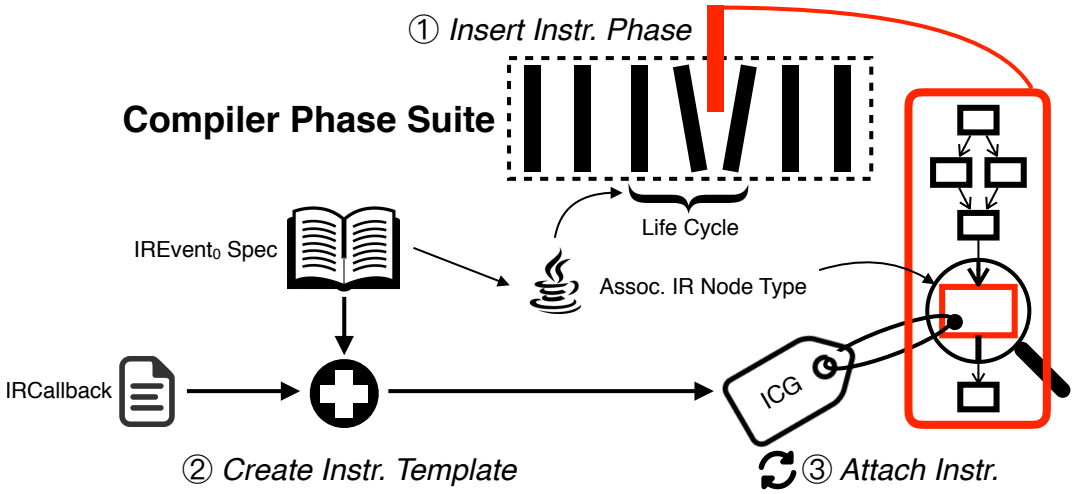


Figure 4.3. Overview of the event-based IR profiling framework.

4.3.3 Architecture

Figure 4.3 gives an overview of the IR profiling framework. During initialization, our framework first looks up the IR node type associated with the specified IR event, and inserts an instrumentation phase preceding the last optimization phase within the life cycle of the type (step 1). Then, it merges both the IR-event instantiation logic and the IR callback into an instrumentation template (step 2). Upon dynamic compilation, the instrumentation phase searches for the IR nodes of the associated types and attaches an ICG to them; the ICG is instantiated from the instrumentation template (step 3). Eventually, the ICGs are spliced back into the base program IR before emitting the machine code.

4.3.4 Optimizations on the Inserted Code

Similar to other instrumentation techniques, an IR instrumentation may result in greatly inflated code. Besides suggesting developers to keep IR callbacks concise by extracting profiling logic into a separate method, we also perform various optimizations on the instrumentation. One very beneficial optimization is escape analysis on the instantiated IR event. If the allocated IR event does not escape the IR callback, our framework is able to eliminate the allocation and the initialization of the IR event. For example, Figure 4.4 illustrates the pseudo code of the instrumentation after performing escape analysis on Figure 4.2. The eventual instrumentation no longer contains the allocation and initialization of MemoryBarrierEvent.

```
1 int barrierKind = ... // The init value is provided by
2                       // the corresponding specification.
3 // Inlined IRCallback
4 MemoryBarrierProfiler.profile(barrierKind);
```

Figure 4.4. Applying escape analysis on the instrumentation from Figure 4.2.

Another beneficial optimization is partial evaluation [51, 89] of the instantiated instrumentation. Because many IR events carry constant information, there is the potential for optimizing the instrumentation using the contents of the concrete target IR node. Especially, if the designated IR callback contains branching statements on such constant information, our framework is able to eliminate dead branches and eventually the corresponding condition tests.

We rely on partial evaluation to create embedded compiler-time filters. For example, even though the dynamic compiler should insert memory barriers preceding or succeeding volatile filed accesses for implementing the Java Memory Model (JMM) [80], part of these barriers amount to no-ops especially on processors without out-of-order load/store [63]. The dynamic compiler still places the IR nodes that represent memory barriers for all barrier kinds in the architecture-independent front-end. These IR nodes will then be evaluated based on the barrier kind in the architecture-dependent back-end. For example, on the x86_64 architecture, all barriers except those separating a store operation from subsequent load operations need not be inserted. Thus, we create an embedded filter and profile only the materialized memory barriers, as shown in Figure 4.5a. When the instrumentation is instantiated using a memory barrier that separates a store operation from subsequent load operations, the resulting instrumentation is the code within the if block (Figure 4.5b). Otherwise, the resulting instrumentation is empty (Figure 4.5c).

4.4 Evaluation

For our evaluation, we modify the StoreLoadBarrierProfiler from Figure 4.5a to further categorize the executed barriers by their originating bytecode. It reports the number of executed barriers in the compiled code per originating bytecode on an x86_64 architecture, so as to detect the hottest locations with such expensive memory operations. We profile selected benchmarks[15] from the DaCapo 9.12

[15]We excluded the tomcat benchmark due to a well known issue (see http://sf.net/p/dacapobench/bugs/68/). We also excluded the eclipse benchmark due to its incompatibility with Java 8.

```
1 public class StoreLoadBarrierProfiler {
2   @IRCallback(event = MemoryBarrierEvent.class)
3   static void callback(MemoryBarrierEvent e) {
4     if ((e.getBarrierKind() & STORE_LOAD) != 0) {
5       profileStoreLoad();
6     }
7   }
8   static void profileStoreLoad() {
9     // count StoreLoad barriers
10  }
11 }
```

(a) IR callback with compile-time filter that selects only memory barriers inserted between a store operation and subsequent load operations. These memory barriers are denoted as STORE_LOAD (0x04 in the Graal compiler), and are often combined with those between a store operation and subsequent store operations (denoted by STORE_STORE, 0x08), for implementing the Java Memory Model with respect to volatile field writes.

```
1   StoreLoadBarrierProfiler.profileStoreLoad();
```

(b) Inserted instrumentation after a memory barrier for separating a store operation and subsequent load operations.

```
1   // empty instrumentation
```

(c) Instrumentation on other memory barriers removed by partial evaluation.

Figure 4.5. Partial evaluation of the instrumentation.

suite and all benchmarks from the ScalaBench suite on a multi-core platform[16]. We exclude the profile from the compiler threads and report the results for the 1st (startup) and the 15th (steady state) benchmark iterations.

Table 4.2 shows the collected results for DaCapo and Scalabench. We note a varying difference between the startup and the steady-state iterations for individual benchmarks. The reasons are twofold. Firstly, during the startup iteration the benchmark code is in the least optimized form. With more methods being compiled, the chance of triggering our IR profiler increases. Secondly, the startup iteration may initialize the input for the whole benchmarking process. For example, the fop benchmark fetches data from a zip file and stores the data in a ConcurrentHashMap during the startup[17]. In general, the DaCapo benchmarks

[16]Intel Xeon E5-2680, 2.7 GHz, 8 cores, 64 GB RAM, CPU frequency scaling and Turbo mode disabled, hyper-threading enabled, Oracle JDK 1.8.0_101 b13 Hotspot Server VM (64-bit), running on Ubuntu Linux Server 64-bit version 14.04.1

[17]We observe executed barriers in ZipFile$ZipFileInputStream.close() and in ConcurrentHashMap.putVal(Object,Object,boolean).

	Benchmark	*1st iteration*			*15th iteration*		
		$\#_{Total}$	$\#_{APP}$	$\%_{APP}$	$\#_{Total}$	$\#_{APP}$	$\%_{APP}$
Dacapo	avrora	422	0	0.0	0	0	0.0
	batik	50 326	45 396	90.2	98 694	96 893	98.2
	fop	1924	0	0.0	0	0	0.0
	h2	87 971	87 949	100.0	97 725	93 723	95.9
	jython	4 688 919	486 116	10.4	4 360 693	420 539	9.6
	luindex	104	0	0.0	11 243	0	0.0
	lusearch	4	0	0.0	1903	0	0.0
	pmd	14 845	0	0.0	10 391	0	0.0
	sunflow	6	0	0.0	0	0	0.0
	tradebeans	101 206	0	0.0	85 887	0	0.0
	tradesoap	764 066	6764	0.9	920 517	15 660	1.7
	xalan	10 627	0	0.0	81 471	0	0.0
	Average			16.8			17.1
ScalaBench	actors	20 602 229	5 782 267	28.1	22 502 956	7 155 007	31.8
	apparat	972 148	728 747	75.0	1 109 886	801 213	72.2
	factorie	2108	336	15.9	2719	342	12.6
	kiama	281 044	46 518	16.6	282 099	47 112	16.7
	scalac	738 585	689 629	93.4	726 018	716 094	98.6
	scaladoc	552 692	523 979	94.8	568 495	549 287	96.6
	scalap	2258	0	0.0	12 527	4322	34.5
	scalariform	101 945	93 917	92.1	139 590	130 663	93.6
	scalatest	123 883	0	0.0	63 911	12 154	19.0
	scalaxb	24 489	45	0.2	33 849	0	0.0
	specs	101 079	3	0.0	50 294	7045	14.0
	tmt	3953	0	0.0	39 802	1551	3.9
	Average			34.7			41.1

Table 4.2. Number of executed memory barriers for dynamically compiled code in DaCapo and ScalaBench benchmarks. The $_{APP}$ subscript denotes memory barriers originating from the application code instead of the Java class library.

execute significantly less barriers when compared to the ScalaBench benchmarks. Especially, there are three workloads that do not execute a single memory barrier in the steady state. On the contrary, the ScalaBench suite contains more workloads

with considerable numbers of executed barriers.

Table 4.2 also shows the number and the proportion of executed barriers derived from the application code (in contrast to the Java class library) for each benchmark. For the startup iteration of DaCapo, the percentage of executed barriers from application code is 16.8% on average. That is, most memory barriers are executed in code of the Java class library, such as in the java.util.concurrent package. For the steady-state iteration of DaCapo, the percentage of memory barriers executed in application code slightly increases to 17.1%. In both the startup and the steady-state iterations, only 4 out of 12 benchmarks execute memory barriers in the application code. In comparison, the average percentages for ScalaBench are 34.7% and 41.1% for the startup and steady-state iteration, respectively. Moreover, only a single benchmark executes no barrier in application code in the steady-state iteration.

We analyze the hottest bytecodes reported by our profiler for each benchmark. In particular, we find that in jython the memory barrier resulting from the volatile field store at org.python.core.PyList.list_append(PyObject)#11 is executed 315 089 times (85 182/s) in the steady state. Further manual analysis shows that the corresponding volatile field is loaded only during sorting of the PyList, which is rarely invoked. This breaks the assumption of the barrier-placement strategy that the read accesses to a volatile field would outnumber the writes to the field [63]. Our profiler helps identifying such situation where an alternative placement strategy that issues the StoreLoad barriers before each volatile read would be beneficial.

With respect to performance, the StoreLoadBarrierProfiler employs a lightweight technique that maintains thread-local counters and increments them once a MemoryBarrierEvent is received. Thanks to the optimizations we apply to the inserted code, the instantiation of the MemoryBarrierEvent is eliminated and the compile-time filter is evaluated based on the kind of the memory barrier. The eventual inserted code is reduced to just a few extra memory-access instructions. As a result, the profiler introduces only negligible overhead below 0.1%, even for exact profiling.

4.5 Discussion

Below, we discuss the benefits and limitations about our approach.

Applicability. Our approach provides abstractions of the IR-level operations and allows profiling them without knowledge of the underlying implementation of the IR operations in the dynamic compiler. For example, one only needs to know about

the concept of a memory barrier instead of the dedicated (and often complex) IR representation (e.g., MembarNode representing a memory barrier in Graal's IR). Our approach offers profiling in an event-based manner. The developer can thus focus on the analysis code and rely on our framework for the instrumentation.

Our approach also enables the composition of IR-level profiling and bytecode-level profiling. This can be beneficial for binding IR-level metrics to the calling context maintained by bytecode-level profiling. In addition, because our approach targets IR-level operations that do not exist during interpretation, it is essential to combine bytecode-level profiling for covering the full program execution. For example, one may need to emit memory-barrier events both in the interpreter and in the compiled code for data-race detection. In this case, the isMethodCompiled intrinsic presented in Chapter 3 can be useful for excluding the redundant bytecode-level profiling in the compiled code.

Extensibility. The implementation of the IR events is modular in the sense that each IR event corresponds to a specification defining the associated IR node type and the instantiation of the event. To extend the library with a new IR event, we only need to add a new specification. In our framework, we enumerate all IR node types within the Graal compiler and create an IR event for each type. Yet, there is still room for extensibility in the future work. One possibility would be to provide events representing IR nodes with certain traits. For example, for detecting loop unswitching opportunities, the developer would subscribe to an event representing a branch operation within a loop. Another possibility would be to support events representing IR subgraphs of a particular structure (e.g., basic blocks at the IR level) instead of individual IR nodes.

Performance. In general, the performance of the custom IR callback (as well as the profiling routine it typically invokes) does not depend on the event instantiation code, which is generated by our approach and inlined at each instrumentation site. However, the latter may significantly increase the memory usage and lead to additional garbage collections. Thanks to the escape analysis performed on the inserted code, allocations of the IR event can be avoided if the event instance does not escape the inserted code. In practice, the IR event is often needed only conceptually, and its carried internal information can be directly used after scalar replacement. Moreover, whether an instrumentation is needed in a particular code site can be evaluated during compilation by constructing branches based on compile-time constants.

4.6 Summary

In this chapter we present an easy-to-use event-based framework for profiling IR-level operations. Our framework provides an abstraction of each IR node in the form of an event, and the developer only needs to register callbacks to subscribe to these events. We demonstrate the usage as well as the internal implementation and optimization with a profiler subscribing to $\mathrm{MemoryBarrierEvent}$. Our framework also enables the composition of IR-level profiling and bytecode-level profiling.

We use the running example of memory barrier profiler to conduct a simple workload characterization study on the DaCapo and ScalaBench benchmarks. In the next chapter, we will use a profiler subscribing to the $\mathrm{DeoptimizationEvent}$ for conducting an empirical study on deoptimization behavior.

Chapter 5

Empirical Study on Deoptimization

In this chapter, we present an empirical study on deoptimization in a VM with Graal. We start with the motivation of this study in Section 5.1. We discuss the relevant background on speculation and deoptimization in Section 5.2. In Section 5.3, we discuss the general deoptimization profiling result and analyze two special cases with considerable amount of repeated deoptimizations. We then present the alternative deoptimization strategies in Section 5.4 and evaluate them in Section 5.5. An assessment of the strengths and limitations of our study can be found in Section 5.6.

5.1 Motivation

Besides producing machine-code for the underlying hardware platform, the dynamic compiler is also in an ideal position to perform speculative optimizations based on the collected profiling information. While *profile-driven* and *feedback-driven* optimizations are not exclusive to managed platforms with dynamic compilers, a dynamic compiler works with profiles that reflect the actual behavior of the currently executing program. This provides the compiler with a more accurate view of the common-case behavior which the compiler should optimize. If a certain assumption about program behavior turns out to be wrong, the affected code can be recompiled to reflect the new behavior. This allows the dynamic compiler to pay less attention to uncommon execution paths, replacing them with traps that switch from program's machine code back to the virtual machine's (VM) runtime which then decides how to handle the situation. As a result, the compiler needs to do less work and produces higher-density code for the common code paths. Combined with aggressive inlining and code specialization based on receiver type feedback, a dynamic compiler can optimize away a significant

portion of the abstraction overhead commonly found in object-oriented programs that make heavy use of small methods and dynamic binding.

The pioneering work by the authors of the SmallTalk-80 [26] and the Self-93 [48] systems has laid down the foundations of modern dynamic compilers, and sparked an enormous body of research [3] on techniques that make managed language platforms fast, such as selective compilation [48, 59, 19, 60, 61], profiling for feedback-directed optimization and code generation [101, 7, 110, 94], or dynamic deoptimization and on-stack replacement [47, 82, 33, 58, 52]. As a result, adaptive compilation and speculative optimization techniques are now widely used. Ideally, speculative optimizations will always turn out to be right and provide performance gains that outweigh the one-time cost in terms of compilation time before the program terminates. In reality, some speculations in the machine code will be wrong, and trigger *deoptimization*. Besides switching to interpreted (or otherwise less optimized) execution mode, deoptimization may also trigger recompilation of the affected code, thus wasting previous compilation work and adding to the overall cost of compilation.

How often does this happen and for what reason? How much compilation effort is wasted and what is the cost of speculative optimizations? What happens when the compiled code triggers deoptimization? In fact, these aspects of speculative optimizations have not been previously studied in the literature—unlike, e.g., the trade-offs involved in selective compilation. We therefore analyze the deoptimization behavior of code compiled by the Graal dynamic compiler and the behavior of the VM runtime in response to the deoptimizations. Even though Graal has not (yet) replaced the classic C2 server compiler, it is integrated in Oracle's HotSpot Virtual Machine and serves as the basis for the Truffle framework for self-optimizing interpreters [104]. Truffle allows executing programs written in modern dynamic languages on the JVM and generally outperforms the original interpreters. Similarly to the classic C2 compiler, Graal performs feedback-directed optimizations and generates code that speculates on receiver types and uncommon paths, but is more aggressive about it. Unlike the C2 compiler, when Graal reaches a deoptimization site in the compiled code, it switches back to interpreted mode and discards the machine code with the aim to generate it again using better profiling information. The C2 compiler is more conservative and in many cases discards the compiled code only after it triggers multiple deoptimizations. The obvious question is then: which of the two approaches is better, and how often programs actually violate the assumptions put in the code by the dynamic compiler?

To answer this question, we characterize the deoptimization causes in the code produced by Graal for the DaCapo [12], ScalaBench [88], and Octane [37]

benchmark suites (Section 5.3). We show that only a small fraction (∼2%) of deoptimization sites is triggered, most of which (∼98%) cause reprofiling. We investigate the causes of two types of repeatedly triggered deoptimizations that appear in the profile. We provide two alternative deoptimization strategies for the Graal compiler. A *conservative* strategy, which defers invalidation of compiled code until enough deoptimizations are observed (default HotSpot behavior not used by Graal), and an *adaptive* strategy which switches among various deoptimization actions based on a precise deoptimization profile (Section 5.4). We evaluate the performance of both deoptimization strategies and compare them to the default strategy used by Graal (Section 5.5). We show that the *conservative* strategy may cause extra compilation work, while the *adaptive* strategy reduces compilation work and provides statistically significant benefits to startup performance on a single-core system with both static and dynamic languages.

5.2 Background

In this section we first provide background on the use of deoptimization in speculative optimizations, and then complement it with details specific to the Graal compiler.

5.2.1 Speculation and Deoptimization

Speculative optimizations are aimed at optimizing for the common case, which is approximated using profiling data collected during program execution. Common speculative optimizations include implicit null checks, uncommon conditional branch elision, and type specialization. If a speculation turns out to be wrong, deoptimization allows the VM to ensure that the program always executes correctly, albeit more slowly.

Deoptimizations are usually triggered synchronously with program execution, either explicitly by invoking a deoptimization routine of the VM runtime, or implicitly, by performing an operation which causes a signal (e.g., segmentation fault in the case of a null pointer) to be sent to the VM, which handles the signal and switches execution to the interpreter. Deoptimizations can be also triggered asynchronously at the VM level, when the program invalidates assumptions under which it was compiled, e.g., when the second class implementing an interface is loaded.

The ability to trigger deoptimization from compiled code allows the compiler to avoid generating code that will be rarely used, e.g., code that constructs and

throws exceptions, because exceptions should be rare in well-written programs. This applies both to explicitly thrown exceptions as well as exceptions that can be thrown implicitly by operations such as array access or division by zero. Based on the profiling feedback, the dynamic compiler can apply a similar approach to conditional jumps, replacing low-probability branches with a deoptimization trigger. Hence, the compiler saves computing resources by avoiding code generation for the uncommon paths. Moreover, this approach helps speed up global optimizations thanks to the reduced program state, and makes the generated machine code more compact, resulting in better instruction cache performance.

Another common kind of speculative optimization relies on type feedback, which allows the compiler to specialize code to most commonly used types. For instance, the targets of a virtual method invocation may be inlined (or the invocation can be devirtualized) if only a limited number of receiver types has been observed at a particular callsite. The type-specific code will be guarded by type-checking conditions, while a generic code path representing an uncommon branch may trigger deoptimization to handle the invocation in the interpreter.

While deoptimization is handled by the VM runtime, the compiler needs to provide the VM with details on how to handle it. This information is typically provided in form of parameters passed to the invocation of the deoptimization trigger routine in the generated code. For example, if recompilation of the code that triggers a deoptimization is unlikely to make it any better, the VM is instructed to just switch to the interpreter and leave the compiled code as-is. If a deoptimization does not depend on profiling data and could be eliminated by recompiling the code, the code is invalidated and the corresponding compilation unit is immediately scheduled for recompilation. If a deoptimization was caused by insufficient profiling information, besides invalidating the machine code, the VM also attempts to reprofile the method thoroughly and recompile it later based on the updated profile. To avoid an endless cycle of recompilation and deoptimization for pathologic cases, per-method counters are used to stop recompilation of a method if it has been recompiled too many times (yet did not eliminate the deoptimization).

In state-of-the-art dynamic compilers the mapping between a deoptimization reason and the corresponding deoptimization action is hard-coded. This makes perfect sense for certain cases, when there is only a single suitable deoptimization action. However, determining the most suitable action for situations in which the deoptimization is caused by an incomplete profile is difficult. For instance, when the compiler inlines potential callee methods based on the receiver type profile, it inserts a reprofiling deoptimization trigger in the uncommon (generic) path to cope with previously unseen receiver types. When encountering a very rare

Graal Deopt Action	*Description*	*HotSpot Deopt Action*
None	Do not invalidate the compiled code.	none
RecompileIfTooManyDeopts	Do not invalidate the compiled code and schedule a recompilation if enough deoptimizations are seen.	maybe_recompile
InvalidateReprofile	Invalidate the compiled code and reset the invocation counter.	reinterpret
InvalidateRecompile	Invalidate the compiled code and schedule a recompilation immediately.	make_not_entrant
InvalidateStopCompiling	Invalidate the compiled code and stop compiling the outermost method of this compilation.	make_not_compilable

Table 5.1. Deoptimization actions in the Graal compiler. The common prefix of the corresponding HotSpot deoptimization actions ("Action_") are omitted.

receiver type, deoptimization (including reprofiling) is triggered. However, due to the (usually) limited receiver type profile space[18], the newly collected profiling information might not include the rare case at the time of recompilation. The dynamic compiler will then either use the original invocation as the uncommon path (if megamorphic inlining is supported), or not inline the callsite at all. In both cases, the reprofiling and recompilation effort is wasted, and the recompiled code may become even worse.

5.2.2 Deoptimization in the Graal Compiler

The Graal compiler makes heavy use of profile-directed speculative optimizations and is thus more likely to exhibit deoptimizations. Because Graal only provides the last-level compiler, it can only instruct the HotSpot runtime what action to perform during deoptimization. The HotSpot runtime takes care of everything else. The deoptimization actions used internally by Graal can be therefore directly mapped to the deoptimization actions defined in the HotSpot runtime.

The possible deoptimization actions are summarized in Table 5.1. Apart from the None action, which only switches execution to the interpreter, all other options influence the compilation unit which triggered the deoptimization in some way. Most of them invalidate the compilation unit's machine code immediately, with

[18] -XX:TypeProfileWidth in the Oracle JVM, defaults to 2 in standard HotSpot runtime or 8 in the Graal compiler.

Deoptimization Reason	*Description*	*Associated Action*
None	Absence of a relevant deoptimization.	-
NullCheckException	Unexpected null or zero divisor.	None InvalidateRecompile InvalidateReprofile
BoundsCheckException	Unexpected array index.	InvalidateReprofile
ClassCastException	Unexpected object class.	InvalidateReprofile
ArrayStoreException	Unexpected array class.	InvalidateReprofile
UnreachedCode	Unexpected reached code.	InvalidateRecompile InvalidateReprofile
TypeCheckedInliningViolated	Unexpected receiver type.	InvalidateReprofile
OptimizedTypeCheckViolated	Unexpected operand type.	InvalidateRecompile InvalidateReprofile
NotCompiledExceptionHandler	Exception handler is not compiled.	InvalidateRecompile
Unresolved	Encountered an unresolved class.	InvalidateRecompile
JavaSubroutineMismatch	Unexpected JSR return address.	InvalidateReprofile
ArithmeticException	A null_check due to division by zero.	None InvalidateReprofile
RuntimeConstraint	Arbitrary runtime constraint violated.	None InvalidateRecompile InvalidateReprofile
LoopLimitCheck	Compiler generated loop limits check failed.	InvalidateRecompile
TransferToInterpreter	Explicit transfer to interpreter.	-

Table 5.2. Deoptimization reasons in the Graal compiler.

the exception of the RecompileIfTooManyDeopts action, which depends on a profile of preceding deoptimizations, and only invalidates the compiled code if too many deoptimizations are triggered at the same site or within the compilation unit.

Even though the deoptimization action is fixed in the compiled code, the HotSpot runtime rewrites the actual action either to force reprofiling or to avoid endless deoptimization and recompilation cycles. If a recompilation is scheduled for the second time for the same deoptimization site with the same reason, the HotSpot runtime rewrites the action to InvalidateReprofile, which resets

method's hotness counters and causes it to be reprofiled. If the total number of recompilations of any method exceeds a threshold, the HotSpot runtime rewrites the action to InvalidateStopCompiling to prevent further recompilation of the method.

To illustrate how Graal uses these deoptimization actions, Table 5.2 shows the deoptimization reasons along with the associated actions as defined and used throughout the Graal code base. The table reveals that the actions RecompileIfTooManyDeopts and InvalidateStopCompiling are not used as of Graal v0.17[19]. This suggests that the compiler tries to keep full control over invalidation of compiled code, and that it tries not to give up any optimization opportunity until the HotStop runtime enforces certain actions.

Some of the deoptimization reasons are used with multiple actions, depending on the situation in which the deoptimization is invoked. For instance, the OptimizedTypeCheckViolated reason is used when inlining the target of an interface with a single implementation, and when optimizing instanceof checks. In the former case, if a guard on the expected receiver type fails, the compiler invokes the InvalidateRecompile action with the reason OptimizedTypeCheckViolated, because it has produced the compiled code under the assumption that there is only a single implementation of a particular interface. In the latter case, the compiler checks against types derived from the given type that have been observed so far. Because the occurrence of a previously unseen type indicates an incomplete type profile, the compiler invokes the InvalidateReprofile action to get a more accurate type profile. If the compiler knew that the previously unseen type was a very rare case, it could invoke the None action. However, because encountering a new type may also signify a phase change in the application, Graal uses the InvalidateReprofile action.

Nevertheless, the mapping between deoptimization reasons and deoptimization actions in the Graal compiler is hard-coded and represents the trade-offs between startup and steady-state performance made by the compiler developers. Yet the HotSpot demands both information, to profile the occurrence of each deoptimization reason per method, and to adapt the deoptimization action according to the existing deoptimization profile. In the following sections, we provide quantitative and qualitative analyses of how these decisions influence the Graal compiler's actual deoptimization behavior.

[19] https://github.com/graalvm/graal-core/tree/graal-vm-0.17

5.3 Study of Deoptimization Behavior

In this section we analyze the deoptimization behavior of the HotSpot VM with the Graal compiler when executing benchmarks from the DaCapo [12], ScalaBench [88], and Octane [37] benchmark suites. The individual benchmarks are based on real-world programs written in Java and Python (DaCapo), Scala (ScalaBench), and JavaScript (Octane), slightly modified to run under a benchmarking harness suitable for experimental evaluation. The Python workloads are executed by Jython, a Python interpreter written in Java, the Scala workloads are compiled to Java bytecode, and the JavaScript workloads are executed by Graal.js, a JavaScript runtime written in Java on top of the Truffle framework [104].

We first analyze the kind of deoptimization sites emitted by Graal and the frequency with which they are triggered during execution, and then investigate two specific cases in which the same deoptimizations are triggered repeatedly.

5.3.1 Profiling Deoptimizations

To collect information about deoptimizations, we implement a profiler based on our IR-level profiling technique presented in Chapter 4. The profiler subscribes to the DeoptimizationEvent and instruments each deoptimization site and reports the number of deoptimizations triggered at that site during execution.

The identity of each deoptimization site consists of the deoptimization reason, action, the originating method and bytecode index, and (optionally) a context identifying the compilation root if the method was inlined. The information encoded in the site identity along with the number of deoptimizations triggered at the site allow us to perform qualitative and quantitative analysis of the deoptimizations triggered in the compiled code produced by Graal. To this end, we profile selected benchmarks[20] from the DaCapo 9.12 suite, all benchmarks from the ScalaBench suite, and selected benchmarks[21] from the Octane suite on a multi-core platform[22].

We present the resulting profile from different perspectives. First we provide a static break-down of the deoptimization sites and deoptimization actions found

[20] We excluded the tomcat benchmark due to a well known issue (see `http://sf.net/p/dacapobench/bugs/68/`). We also excluded the eclipse benchmark due to its incompatibility with Java 8.

[21] We excluded the pdf.js benchmark due to an internal exception.

[22] Intel Xeon E5-2680, 2.7 GHz, 8 cores, 64 GB RAM, CPU frequency scaling and Turbo mode disabled, hyper-threading enabled, Oracle JDK 1.8.0_101 b13 HotSpot Server VM (64-bit), running on Ubuntu Linux Server 64-bit version 14.04.1

in the code emitted by Graal (Section 5.3.1). This is complemented by a dynamic view of deoptimization sites that are actually triggered during execution (Section 5.3.1). Finally, we look at the most frequent repeatedly-triggered deoptimizations, because these are potential candidates for wasted compilation work (Section 5.3.1).

Deoptimizations Sites Emitted

The profiling results of the emitted deoptimization sites are summarized in Table 5.3 and Table 5.4. The top-level column groups represent the actions used at the deoptimization sites. We only track three of the five possible deoptimization actions, because Graal does not make use of the other two (c.f. Section 5.2). The bottom-level columns correspond to the number of deoptimization sites invoking a particular action and the fraction of the total number of sites.

In general, the number of deoptimization sites emitted during a benchmark's lifetime varies significantly, ranging from 2000 to 23 000. For the DaCapo benchmarks, 94.17% of the total deoptimization sites invoke the InvalidateReprofile action, 3.23% just switch to the interpreter (action None), and 2.60% invoke the InvalidateRecompile action. For the ScalaBench benchmarks, the compiler emits a slightly higher proportion (95.44%) of the InvalidateReprofile deoptimization sites and a lower proportion (1.16%) of the InvalidateRecompile sites. We attribute this to the fact that the Scala language features are compiled into complex call chains in the Java bytecode. During dynamic compilation, these callsites are optimized with type guards that lead to InvalidateReprofile deoptimization sites. To summarize, in standard Java/Scala applications the Graal compiler favors speculative profile-directed optimizations, which invoke the InvalidateReprofile deoptimization action in their guard failure paths.

For the Octane benchmarks, the compiled code of the Graal.js self-optimizing interpreter contains a higher proportion (4.74%) of the InvalidateRecompile deoptimization sites. One of the reasons for this difference is that language runtimes implemented on top of the Truffle framework heavily utilize the Truffle API. Because this API consists of many interfaces with a single implementation, the compiled code for callsites invoking the Truffle API uses guarded devirtualized invocations. Consequently, the (many) corresponding guard failure paths invoke the InvalidateRecompile deoptimization action with OptimizedTypeCheckViolated as the reason (c.f. Section 5.2). The second reason for the higher proportion of InvalidateRecompile deoptimization sites is that the Truffle framework encourages aggressive type specialization in the interpretation of abstract syntax tree (AST) nodes of the hosted language. Internally, Truffle uses Java's exception

Benchmark	None		Reprofile		Recompile	
	# Sites	%	# Sites	%	# Sites	%
avrora	94	3.15	2813	94.21	79	2.65
batik	147	3.48	3991	94.48	86	2.04
fop	186	3.83	4639	95.55	30	0.62
h2	208	2.60	7516	93.96	275	3.44
jython	337	2.94	10 837	94.54	289	2.52
luindex	196	6.40	2839	92.75	26	0.85
lusearch	204	6.68	2785	91.22	64	2.10
pmd	163	2.56	5942	93.21	270	4.24
sunflow	92	4.11	2123	94.73	26	1.16
tradebeans	267	2.68	9307	93.37	394	3.95
tradesoap	608	2.76	20 866	94.56	593	2.69
xalan	225	3.65	5880	95.33	63	1.02
Total	2727	3.23	79 538	94.17	2195	2.60
actors	116	2.50	4418	95.17	108	2.33
apparat	230	3.79	5751	94.71	91	1.50
factorie	133	3.98	3153	94.40	54	1.62
kiama	178	4.90	3423	94.25	31	0.85
scalac	289	1.82	15 525	97.62	90	0.57
scaladoc	288	2.54	10 909	96.10	155	1.37
scalap	133	5.16	2428	94.15	18	0.70
scalariform	189	4.27	4198	94.74	44	0.99
scalatest	215	4.96	4083	94.19	37	0.85
scalaxb	166	4.43	3547	94.74	31	0.83
specs	212	5.40	3672	93.60	39	0.99
tmt	191	3.96	4535	93.99	99	2.05
Total	2340	3.40	65 642	95.44	797	1.16

Table 5.3. The number and percentage of deoptimization sites with a particular action emitted during the first benchmark iteration of the DaCapo and ScalaBench workloads.

Benchmark	None		Reprofile		Recompile	
	# Sites	%	# Sites	%	# Sites	%
box2d	195	2.19	8162	91.76	538	6.05
code-load	699	2.85	23 041	93.79	827	3.37
crypto	142	2.08	6325	92.59	364	5.33
deltablue	136	2.31	5395	91.78	347	5.90
earley-boyer	172	2.36	6740	92.48	376	5.16
gbemu	200	1.91	9743	92.89	546	5.21
mandreel	367	3.33	10 197	92.41	470	4.26
navier-stokes	129	2.43	4930	92.76	256	4.82
raytrace	133	2.16	5696	92.42	334	5.42
regexp	221	2.27	9050	93.11	449	4.62
richards	113	2.15	4834	91.85	316	6.00
splay	123	2.02	5664	93.11	296	4.87
typescript	294	2.04	13 403	93.13	694	4.82
zlib	204	2.93	6458	92.79	298	4.28
Total	3128	2.43	119 638	92.83	6111	4.74

Table 5.4. The number and percentage of deoptimization sites with a particular action emitted during the first benchmark iteration of the Octane workloads on Truffle/JS.

mechanism to undo type specialization, and because at the time the type specialization occurs the exception handler has never been executed (otherwise the type specialization would not happen in the first place), the dynamic compiler considers the exception handler to be uncommon and replaces it with a deoptimization site which invokes the InvalidateRecompile action with NotCompiledExceptionHandler as the reason. This mechanism allows Truffle to attempt aggressive type specialization and recompile with generic types if a type-related exception occurs.

Deoptimization Sites Triggered

The profiling results of the triggered deoptimization sites are summarized in Table 5.5 and Table 5.6. We only track the InvalidateReprofile and InvalidateRecompile actions, because deoptimizations with the None action is not triggered at any of the sites emitted. The bottom level columns correspond to the number of deoptimization actions of particular type triggered, the fraction of the total

Benchmark	Reprofile			Recompile		
	# Deopts	%	% Sites	# Deopts	%	% Sites
avrora	43	100.00	1.04	0	0.00	0.00
batik	119	99.17	2.70	1	0.83	0.02
fop	80	100.00	1.52	0	0.00	0.00
h2	242	98.37	2.49	4	1.63	0.05
jython	256	98.84	1.89	3	1.16	0.03
luindex	42	100.00	1.37	0	0.00	0.00
lusearch	33	100.00	0.75	0	0.00	0.00
pmd	78	90.70	1.11	8	9.30	0.09
sunflow	26	100.00	1.12	0	0.00	0.00
tradebeans	302	99.34	2.25	2	0.66	0.02
tradesoap	471	99.16	1.71	4	0.84	0.02
xalan	24	100.00	0.34	0	0.00	0.00
Total	1716	98.73	1.68	22	1.27	0.02
actors	229	96.22	1.87	9	3.78	0.09
apparat	166	88.77	2.45	21	11.23	0.12
factorie	82	100.00	1.71	0	0.00	0.00
kiama	117	100.00	2.26	0	0.00	0.00
scalac	655	99.85	3.15	1	0.15	0.01
scaladoc	450	100.00	2.93	0	0.00	0.00
scalap	44	100.00	1.59	0	0.00	0.00
scalariform	64	100.00	1.11	0	0.00	0.00
scalatest	57	96.61	1.25	2	3.39	0.05
scalaxb	65	98.48	1.68	1	1.52	0.03
specs	51	96.23	1.27	2	3.77	0.05
tmt	112	100.00	1.97	0	0.00	0.00
Total	2092	98.31	2.27	36	1.69	0.03

Table 5.5. The number and percentage of deoptimization sites with a particular action triggered during the first benchmark iteration of the DaCapo and ScalaBench workloads.

number of deoptimizations, and the deoptimization site coverage, that is, the fraction of the total number of deoptimization sites emitted at which at least one deoptimization was triggered.

Of all the sites emitted for the DaCapo benchmarks, only 1.68% were actually triggered and invoked the InvalidateReprofile deoptimization action during execution. The proportion increases to 2.27% in the ScalaBench benchmarks for the same reason that affects the total number of emitted sites. Similarly, only 0.02% of the sites in the DaCapo benchmarks and 0.03% of the sites in the ScalaBench benchmarks were triggered and invoked the InvalidateRecompile action. This indicates that in ordinary Java/Scala applications, deoptimization sites that do not rely on profiling feedback represent only a small fraction of the total number of deoptimization sites and are rarely triggered. In addition, these sites tend to be eliminated by the recompilation they force, therefore they rarely cause repeated deoptimizations. In total, over 98% of all triggered deoptimizations invoke the InvalidateReprofile action, while only less than 2% invoke the InvalidateRecompile action. This suggests that in the code produced by the Graal compiler, deoptimizations are dominated by those that force reprofiling of the affected code.

Compared to DaCapo and ScalaBench, the number and the proportion of the InvalidateRecompile deoptimizations triggered during execution of the Octane benchmarks on top of Graal.js is significantly higher. As discussed earlier, this is because the Truffle code that undoes type specialization in the hosted language is implicitly replaced by deoptimization. Nevertheless, similarly to DaCapo and ScalaBench, the most frequently triggered deoptimization action in the Octane benchmarks is InvalidateReprofile (88.83%).

Deoptimizations Triggered Repeatedly

In Table 5.7 and Table 5.8 we show the number of sites which trigger a particular deoptimization more than once during benchmark execution. In the DaCapo benchmarks, these sites account for 11.67% of deoptimization sites triggered at least once, and for 26.64% of all triggered deoptimizations; the results for ScalaBench are similar. For the Octane benchmarks on Graal.js, the proportion of repeated deoptimization sites drops to 5.96%, which is caused by Truffle invalidating the type specialization code that triggered a deoptimization.

While it is possible for multiple threads to trigger the same deoptimization site in the same version of the compiled code, the majority of the repeated deoptimizations originate from recompiled code. This means that if recompilation does not eliminate these deoptimization sites, reprofiling either does not produce

Benchmark	Reprofile			Recompile		
	# Deopts	%	% Sites	# Deopts	%	% Sites
box2d	140	73.30	1.43	51	26.70	0.46
code-load	403	80.60	1.43	97	19.40	0.37
crypto	105	90.52	1.46	11	9.48	0.16
deltablue	74	87.06	1.19	11	12.94	0.15
earley-boyer	521	94.90	1.65	28	5.10	0.36
gbemu	308	91.12	2.74	30	8.88	0.28
mandreel	206	81.42	1.72	47	18.58	0.37
navier-stokes	106	96.36	1.88	4	3.64	0.06
raytrace	81	80.20	1.28	20	19.80	0.32
regexp	248	95.75	2.27	11	4.25	0.11
richards	79	88.76	1.48	10	11.24	0.19
splay	117	94.35	1.87	7	5.65	0.12
typescript	702	91.29	1.58	67	8.71	0.38
zlib	148	91.93	1.98	13	8.07	0.17
Total	3238	88.83	1.71	407	11.17	0.28

Table 5.6. The number and percentage of deoptimization sites with a particular action triggered during the first benchmark iteration of the Octane workloads on Truffle/JS.

a profile that would change the optimization decisions, or that the profile is not provided in time for the recompilation.

To aid in investigating the reasons behind the worst-case repeated deoptimizations, Table 5.7 and Table 5.8 also list the deoptimization sites that repeatedly trigger the most deoptimizations during the execution of a particular benchmark. We observe that the most frequently triggered deoptimization sites cause reprofiling for three main reasons: TypeCheckedInliningViolated, OptimizedTypeCheckViolated, and UnreachedCode. Deoptimizations specifying UnreachedCode as the reason result from conditional branches that were eliminated based on (assumed) zero execution probability according to the branch profile for the corresponding bytecode. The actors benchmark contains the most frequent deoptimization site of this type in method java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(), which contains a blocking thread synchronization operation. Deoptimizations that specify type-checking violations as the reason result from optimizations that rely on a type profile. Here, the compiler

Benchmark	*Repeated Deoptimizations*			*Most Frequent Site*	
	# Sites	% Sites	% Deopts	#	Reason
avrora	6	19.35	41.86	4	①
batik	4	3.48	7.50	3	②
fop	2	2.70	10.00	6	①
h2	27	13.30	28.46	6	③
jython	22	10.00	23.55	9	①
luindex	0	0.00	0.00	-	-
lusearch	2	8.70	36.36	10	②
pmd	6	7.79	17.44	5	①
sunflow	1	4.00	7.69	2	②
tradebeans	34	15.04	36.84	10	②
tradesoap	58	15.22	32.00	5	②
xalan	1	4.76	16.67	4	②
Total	163	11.67	26.64		
actors	14	15.38	67.65	64	①
apparat	15	9.62	24.60	3	②
factorie	8	14.04	40.24	9	③
kiama	11	13.41	39.32	10	③
scalac	70	13.94	34.15	23	②
scaladoc	41	12.31	35.11	23	②
scalap	2	4.88	11.36	3	②
scalariform	7	14.29	34.38	7	③
scalatest	3	5.36	10.17	2	②
scalaxb	1	1.56	4.55	3	②
specs	1	1.92	3.77	2	②
tmt	7	7.37	21.43	9	③
Total	180	11.40	34.31		

Table 5.7. The number of deoptimization sites triggered repeatedly and the most frequently triggered InvalidateReprofile deoptimization site during the first benchmark iteration of the DaCapo and ScalaBench workloads. The reasons for most frequently triggered deoptimization sites are ① UnreachedCode, ② TypeCheckedInliningViolated, and ③ OptimizedTypeCheckViolated.

Benchmark	*Repeated Deoptimizations*			*Most Frequent Site*	
	# Sites	% Sites	% Deopts	#	Reason
box2d	16	9.52	20.42	6	②
code-load	35	7.92	18.60	6	①
crypto	2	1.80	6.03	5	①
deltablue	5	6.33	12.94	3	②
earley-boyer	5	3.42	74.32	398	②
gbemu	17	5.38	11.54	4	①
mandreel	12	5.19	13.44	5	①
navier-stokes	4	3.88	10.00	5	②
raytrace	2	2.02	3.96	2	②
regexp	16	6.90	16.60	9	①
richards	1	1.14	2.25	2	②
splay	3	2.48	4.84	2	①
typescript	24	8.48	66.32	237	②
zlib	11	7.33	13.66	2	③
Total	153	5.96	33.72		

Table 5.8. The number of deoptimization sites triggered repeatedly and the most frequently triggered InvalidateReprofile deoptimization site during the first benchmark iteration of the Octane workloads on Truffle/JS. The reasons for most frequently triggered deoptimization sites are ① UnreachedCode, ② TypeCheckedInliningViolated, and ③ OptimizedTypeCheckViolated.

typically uses deoptimization in the failure path of a guard that ensures that type-specific code is only reached with proper types. Among the benchmarks that suffer from deoptimizations for these reasons, the scalac and scaladoc benchmarks share the same worst-case deoptimization site which triggers deoptimization 23 times.

In the case of the Octane benchmarks on Graal.js, a high number of repeated deoptimizations are triggered in the earley-boyer and typescript benchmarks. The underlying reason for repeated deoptimizations is the same as in the case of the DaCapo and ScalaBench suites—inaccurate profiling information caused by associating a profiling record with a deoptimization target (instead of origin), and subsequent sharing of this record by multiple deoptimization sites. Unfortunately, code obfuscation in the Graal.js binary release prevents us from presenting the situation in more detail at source code level.

Iteration	1	2	3	4	5	6	...	15	16	17	18	19	20
avrora	43	17	4	1	2	0		0	0	0	0	0	0
batik	120	20	18	14	6	2		1	1	0	0	1	0
fop	80	23	5	4	2	0		0	1	1	0	0	0
h2	246	17	4	1	2	0		0	1	1	0	0	0
jython	259	27	2	1	0	0		1	1	2	1	0	1
luindex	42	14	1	0	0	0		0	0	0	0	0	0
lusearch	33	3	0	0	0	0	...	0	0	0	0	0	0
pmd	86	25	6	11	5	4		1	1	2	3	0	0
sunflow	26	3	1	0	0	0		0	0	0	0	0	0
tradebeans	304	7	4	0	0	0		0	0	0	0	0	0
tradesoap	475	34	3	0	2	2		0	1	0	0	0	1
xalan	24	1	1	0	0	0		0	0	0	0	0	0
actors	238	28	5	6	4	5		0	1	2	1	1	2
apparat	187	22	9	3	5	2		3	2	2	2	2	3
factorie	82	10	0	0	0	0		0	1	2	0	0	0
kiama	117	5	2	3	3	3		0	0	1	0	0	2
scalac	656	123	36	25	22	21		8	14	5	13	8	12
scaladoc	450	106	18	13	1	6		1	0	0	2	2	8
scalap	44	10	0	0	0	0	...	0	0	0	0	0	0
scalariform	64	23	9	5	4	3		1	0	0	0	0	0
scalatest	59	29	8	9	3	1		1	0	0	0	0	0
scalaxb	66	26	3	0	0	1		0	0	0	0	1	0
specs	53	10	9	5	5	7		6	6	4	2	3	4
tmt	112	6	4	3	2	1		2	1	2	2	2	1

Table 5.9. Number of deoptimizations per iteration when executing the DaCapo and ScalaBench benchmarks.

Deoptimizations per Iteration

Finally, Table 5.9 shows the number of deoptimizations triggered in subsequent benchmark iterations for the DaCapo and ScalaBench benchmarks. Most benchmarks encounter no more than 3 deoptimizations per iteration after the 4th iteration, because the compiled code for most of the hot methods stabilizes. However, there are a few cases of repeated deoptimizations, especially in the scalac benchmark, where on average 10 deoptimizations per iteration are triggered even past the 15th iteration. In most cases, TypeCheckedInliningViolated is given

as the reason, and half of the deoptimizations originate at the same bytecode (scala.collection.immutable.HashSet.elemHashCode(Object)#9) inlined in different methods. This suggests that the receiver type profile may not be updated properly (or soon enough) after deoptimization and reprofiling.

5.3.2 Investigating Repeated Deoptimizations

Our findings in Section 5.3.1 indicate that certain deoptimizations are triggered repeatedly at the same site. If a particular deoptimization is triggered by multiple threads in one version of the compiled code, the subsequent recompilation should eliminate the deoptimization site. However, repeated deoptimizations triggered at the same site in multiple subsequent versions of the compiled code indicate a problem, because that site should have been eliminated by recompilations.

By analyzing the cases of repeatedly triggered deoptimization, we have discovered that this situation occurs because an outdated method profile is used during the recompilation. In the Graal compiler, this can happen because Graal inlines methods aggressively, but at the same time, deoptimization site in the inlined code can deoptimize to the caller containing the callsite of the inlined method (if no program state modification precedes the deoptimization site in the inlined code). After deoptimization, when the interpreter wants to invoke the (previously inlined) method at the callsize, the callee can be compiled either at a different level (without speculation and thus deoptimization), or with a different optimization outcome that did not emit a deoptimization site. In both cases, the profile for the callee is not updated, and subsequent recompilations of its inlined code will use an inaccurate profile, resulting in repeated deoptimizations.

We now illustrate the situations leading to repeated deoptimization for two specific cases: the UnreachedCode deoptimization in the actors benchmark, and the type-check related deoptimizations in the scalac benchmark.

Analyzing Deoptimizations in the actors Benchmark

The results in Table 5.7 show that the actors benchmark contains a site which triggers the UnreachedCode deoptimization 64 times during the first iteration of the benchmark execution. Figure 5.1 shows a snippet of code containing this deoptimization site. The await() method invokes the checkInterruptWhileWaiting(Node) method (line 26), which returns a value depending on the result of the Thread.interrupted() method.

When compiling the await() method, Graal inlines the invocation of the (small and private) checkInterruptWhileWaiting(Node) method at the callsite (line 26).

```
1 public abstract class AbstractQueuedSynchronizer
2     extends AbstractOwnableSynchronizer
3     implements java.io.Serializable {
4   final boolean isOnSyncQueue(Node node) {
5     if (node.waitStatus == Node.CONDITION
6         || node.prev == null)
7       return false;
8     ...
9   }
10  public class ConditionObject
11      implements Condition, java.io.Serializable {
12    private int checkInterruptWhileWaiting(Node node) {
13      return Thread.interrupted() ?
14             (transferAfterCancelledWait(node) ?
15             THROW_IE : REINTERRUPT) : 0;
16    }
17
18    public final void await() throws InterruptedException {
19      ...
20      int savedState = fullyRelease(node);
21      int interruptMode = 0;
22      while (!isOnSyncQueue(node)) {
23        LockSupport.park(this);
24        if ((interruptMode
25             = checkInterruptWhileWaiting(node))
26            != 0)
27          break;
28      }
29      ...
30    }
31  }
32 }
```

Figure 5.1. Excerpt from the source code of java.util.concurrent.locks.AbstractQueuedSynchronizer.

The ternary operator used in the return statement of that method is essentially a conditional branch compiled using the ifeq[23] bytecode, for which the VM collects a branch profile. Because thread interruption happens rarely, it is very likely that all invocations of Thread.interrupted() will return false, and the branch profile for the ifeq bytecode will tell the compiler that the branch was taken in 100% of the cases. By default[24], Graal removes the code in the (apparently) unreachable branch, and inserts a guard for the expected result of the Thread.interrupted() method with a failure path which invokes the InvalidateReprofile deoptimization with UnreachedCode as the reason.

In the await() method, threads may block in the park() method at line 23,

[23]Branch if the value on top of the operand stack is zero, i.e., false.

[24]This can be disabled via -Dgraal.RemoveNeverExecutedCode=false.

which returns when a thread is unparked, or when a thread is interrupted. Any thread returning from the park() method will execute the condition at line 25, including the inlined optimized version of checkInterruptWhileWaiting(Node). If a thread was interrupted, the Thread.interrupted() method returns true contrary to the expectation, and causes the thread to trigger a deoptimization. The first thread to trigger the deoptimization will invalidate the compiled code of the await() method by making it *not entrant* (execution entering the compiled code will immediately switch to interpreter), and resume execution in the interpreter.

However, there may be more threads in the same situation, executing the (now invalidated) compiled code—the 64 repeated deoptimizations in the actors benchmark were caused by 64 different threads triggering the same deoptimization in the same version of the compiled code. While this kind of repeated deoptimization causes threads to execute in the interpreter, it only leads to a single recompilation and is relatively harmless. The branch profile for the ifeq bytecode will be updated during interpreted execution, and taken into account during recompilation of the await() method.

But the await() method contains another UnreachedCode deoptimization site that is problematic. In this case, Graal inlines the invocation of the (final) isOnSyncQueue(Node) method at the callsite (line 22). The null-check in the inlined code uses the ifnonnull bytecode (line 6), which is a conditional branch. Based on the associated branch profile indicating 100% *branch-taken* probability, Graal replaces the unreachable branch with a deoptimization which is triggered if node.prev is null.

If the deoptimization in the loop header is triggered, the code of the await() method will be invalidated and the interpreter will resume execution at beginning of the loop (line 22). The interpreter will then likely invoke the compiled version of the isOnSyncQueue(Node) method, which contains the same guard and deoptimization derived from the same ifnonnull branch profile. In the meantime, because the actors benchmark is highly multi-threaded, another thread may set node.prev to a non-null value. The compiled version of the isOnSyncQueue(Node) method will then execute normally, without retriggering the deoptimization. Without that the isOnSyncQueue(Node) method will not be reinterpreted, and the branch profile for the innonnull bytecode will not be updated. When recompiling the await() method, the compiler will use an inaccurate branch profile and produce the same code that was previously invalidated. In our experiment, we observed 9 deoptimizations originating at the same site, but triggered in different versions of the compiled code. This kind of repeated deoptimizations is more serious, because it causes reprofiling of the await() method (requiring it to be executed in the interpreter more times) and subsequent recompilation, but does

```
1 class HashSet[A] extends Set[A]
2         with GenericSetTemplate[A, HashSet]
3         with SetLike[A, HashSet[A]] {
4   protected def elemHashCode(key: A) =
5       if (key == null) 0 else key.##
6   protected def computeHash(key: A) =
7       improve(elemHashCode(key))
8 }
9 // Java pseudo-code for the ## operation
10 int ##() {
11   if (this instanceof Number) {
12     return BoxesRunTime.hashFromNumber(this);
13   else {
14     return hashCode();
15   }
16 }
```

Figure 5.2. Excerpt from scala.collection.immutable.HashSet.

```
1 if (key.type == String) {
2   // inlined code of String.hashCode
3 } else {
4   deoptimize(InvalidateReprofile,
5     TypeCheckedInliningViolated,
6     HashSet.computeHash /* target method */,
7     0 /* target bytecode index */
8   ); // never returns
9 }
```

Figure 5.3. Pseudo-code of the Graal-compiled code for ##.

not improve the situation.

Analyzing Deoptimizations in the scalac Benchmark

Another deoptimization anomaly that can be observed in the profiling results concerns several benchmarks that exhibit the same pattern of repeated deoptimizations, with either TypeCheckedInliningViolated or OptimizedTypeCheckViolated specified as the reason. This is also true for the steady-state execution of the scalac benchmark shown in Table 5.9, which we now investigate in more detail.

The code containing the deoptimization site is shown in Figure 5.2. At line 7 the computeHash(Object) method invokes the elemHashCode(Object) method, which in turn invokes the ## operation on key. The ## operation is a Scala intrinsic which can be expressed as Java pseudo-code shown in lines 10–16. For every use of the ## operation, the Scala compiler directly inlines the corresponding bytecode sequence into the bytecode it produces.

Line 11 produces an instanceof bytecode which checks for the Number class,

and is subject to type-profile-based optimizations in Graal. When compiling the instanceof bytecode into machine code, the compiler queries the recorded type profile associated with the particular bytecode, and generates tests against the profiled types instead of the operand type, and a failure path which will trigger deoptimization if all the type checks fail.

In our experiment, when compiling the computeHash(Object) method for the first time, the compiler receives a type profile containing only the String class, and generates machine code corresponding to the pseudo-code shown in Figure 5.3. The deoptimization in the else branch actually transfers execution to the beginning of the computeHash(Object) method, because the program state is not mutated between the invocation of the elemHashCode(Object) method and the deoptimization due to the inlined ## operation. When the interpreter reaches the invocation of the elemHashCode(Object) method again, it will likely find the method compiled, so the invocation will switch to machine code. However, with the default tiered compilation strategy, the elemHashCode(Object) method is very likely to be compiled by the level 1 compiler, which is intended for simple methods. As such, level 1 compilation does not use profile-directed optimizations for instanceof and the generated code does not update the profiling information. The compiled version of the elemHashCode(Object) method will therefore correctly handle the ## operation for all types, but the type profile for the inlined code of the ## will not be updated. When Graal compiles the computeHash(Object) method again, it will inline the elemHashCode(Object) method again, but the type profile for the instanceof bytecode will still contain only the String class. The recompiled elemHashCode(Object) method will therefore repeatedly trigger deoptimizations and recompilations.

Consequently, the anomaly occurs when a deoptimization due to an inlined method resumes in the caller and invokes a compiled version of the (previously inlined) callee. If the callee is compiled at level 1, it neither contains profile-directed optimizations nor updates profiling information. When the caller is recompiled (as it is a hot method) and the callee is inlined again, the compiler uses the inaccurate type profile for the code in the callee and generates code that triggers the same deoptimization.

We have also identified a similar problem when Graal devirtualizes method invocations. A devirtualized callsite uses a number of type checks against types from a callsite's receiver profile to invoke concrete methods on specific receiver types, and may trigger deoptimization if it encounters an unexpected receiver type (unless the callsite is megamorphic, which instead performs a virtual method dispatch). The problem occurs if a callsite is devirtualized in the ancestor of the direct caller of a method, which may happen when the direct caller is inlined. If

such a devirtualized (non-megamorphic) callsite triggers a deoptimization and does not transfer execution to the direct caller, the receiver type profile used for devirtualization of the callsite may not be updated if the direct caller also has a standalone compiled version that neither devirtualizes the callsite (and thus trigger the same deoptimization) nor collects profiling information. In general, this situation is caused by the weighted inlining mechanism in the Graal compiler, and the problem would be remedied by either disallowing deoptimization to cross the direct caller's method boundary, or by invalidating its compiled code.

5.4 Alternative Deoptimization Strategies

The deoptimization code produced by Graal mostly invokes the InvalidateReprofile action, hoping to trade extra work in the short term for a potentially better peak performance in the long term. Another reason for using this kind of deoptimization is to cope with application phase changes. These can manifest in the form of completely different execution and type profiles, rendering the compiled code based on profiles from the previous phase obsolete. Obviously, the compiler cannot tell ahead of time whether the actual benefits will outweigh the costs. However, as long as the costs are not excessive, they will be amortized in the long term even without huge performance gains.

With this strategy, the worst-case scenario for long-term performance is the occurrence of rare cases that trigger deoptimization. In this case, the ensuing reprofiling and recompilation will not provide a long-term benefit, but instead cause short-term performance degradation. Worse, during recompilation, the rare case may cause the compiler to abandon speculative optimizations that have worked well before the rare case occurred.

The solution is to introduce some tolerance for rare cases, delaying deoptimizations until the supposedly rare cases become more frequent. This notion is supported by the HotSpot runtime, as the presence of the Action_maybe_recompile deoptimization action suggests. However, Graal does not use its own corresponding action (RecompileIfTooManyDeopts) in the deoptimization code it emits. Presumably, this is because Graal speculates aggressively and the Graal developers do not want to delay recompilation if the program violates optimization assumptions. In addition, because Graal focuses on achieving high peak performance, the cost associated with eager deoptimizations should be amortized in the longer run.

Because the effect of this approach on performance has not been previously studied, we modify Graal to support two additional strategies for handling deop-

timizations and compare the performance achieved with both strategies to the default strategy used by Graal. Unlike the default strategy, which always invokes the InvalidateReprofile action, the alternative strategies differ in the degree of tolerance for rare cases.

5.4.1 Conservative Deoptimization Strategy

The first strategy, referred to as *conservative*, replaces the use of the InvalidateReprofile action with the RecompileIfTooManyDeopts. This strategy relies on the existing mechanisms in the HotSpot runtime to determine when to invalidate the compiled code and when to reinterpret (and possibly reprofile) it. The runtime keeps an execution profile for each method, including information about deoptimizations. The deoptimization profile consists of a counter for each deoptimization reason as well as a recompilation counter. It also stores limited information associated with deoptimization targets (referred to as *traps*), i.e., the bytecode instructions at which the interpreter resumes execution after deoptimization. The per-trap information is keyed to the bytecode index of the target instruction in the target method, and stores the reasons[25] for which the trap was targeted, and whether the method code was invalidated and recompiled due to this trap. The deoptimization reasons are split into two categories considered separately. The first category, referred to as *per-method*, represents reasons that are only considered at the method level, while the second category, referred to as *per-bytecode*, represents reasons that are only considered at the bytecode level.

When a deoptimization is triggered, the HotSpot runtime uses the method profile to make the following decisions: (1a) if the deoptimization reason belongs to the *per-bytecode* category, was previously observed at this trap, and the deoptimization count (taken from the method-level profile) for that reason exceeds a *per-bytecode* threshold[26], the compiled code is invalidated; (1b) if the deoptimization reason belongs to the *per-method* category and the deoptimization count for that reason exceeds a *per-method* threshold[27], the compiled code is invalidated; (2) for compiled code that is to be invalidated, if the per-trap information shows that the code has been previously recompiled for the same *per-bytecode* reason, or if the recompilation counter is greater than 0 for other reasons, the runtime resets the method execution and back-edge counters to facilitate reprofiling; (3) if the

[25] To limit memory consumption, only one precise reason can be stored, otherwise the profile just indicates that there is more than one reason.

[26] -XX:PerBytecodeTrapLimit, defaults to 4.

[27] -XX:PerMethodTrapLimit, defaults to 100.

recompilation counter for a *per-bytecode* reason exceeds a *per-bytecode* threshold[28], or a *per-method* threshold[29] for *per-method* reasons, the deoptimizing method is made *not compilable*.

The per-trap information is inherently approximate. For example, it does not distinguish between two deoptimization sites sharing the same deoptimization target. But when Graal is enabled, it makes it even more approximate. While the trap bytecode index always refers to the instruction in the bytecode of the target method, updates to the per-trap information are stored in the profile of the method in which a deoptimization occurred (not the deoptimization target, as in the case of HotSpot without Graal). A deoptimization triggered by an inlined method will therefore update the per-trap information of the compilation root using an index associated with the bytecode in the target method. This is presumably to avoid spurious invalidation of the compiled code of methods that were inlined with speculative optimizations. However, if several methods inlined in the same compilation root contain a trap instruction, they may share the same slot in the per-trap profile of the compilation root. In addition, due to Graal's aggressive inlining, the deoptimization target may cross method boundaries—a deoptimization from an inlined method may target the returning bytecode of the previous callsite in the caller.

5.4.2 Adaptive Deoptimization Strategy

The second strategy, referred to as *adaptive*, uses a custom deoptimization profile to choose a deoptimization action both during dynamic compilation and during program execution. Unlike the HotSpot runtime or Graal (c.f. Subsection 5.4.1), we simply associate a deoptimization counter with each deoptimization site ID (c.f. Section 5.3), but disregard the stack trace for inlined methods. This means that methods inlined in different compilation roots will update the same deoptimization counters.

During compilation, whenever Graal intends to emit the InvalidateReprofile deoptimization at a particular site, we check the value of the counter corresponding to that site, and emit the default deoptimization code (invoking InvalidateReprofile) if the value is between two thresholds, deoptsTolerated (exclusive, defaults to 1) and deoptsAllowed (inclusive, defaults to 100). If the counter exceeds the deoptsAllowed threshold, too many deoptimizations have been triggered at that particular site, and we instead emit code to invoke the InvalidateStop-

[28] -XX:PerBytecodeRecompilationCutoff, defaults to 200.
[29] -XX:PerMethodRecompilationCutoff, defaults to 400.

Compiling deoptimization. If the method containing the deoptimization site is being inlined, we mark the method as non-inlineable and emit the InvalidateRecompile deoptimization in the inlined code. Consequently, the method is inlined one last time in the compilation root being compiled, but will not be inlined in future recompilations of any method. If the counter does not exceed the deoptsTolerated threshold, the number of deoptimizations triggered at the site is considered tolerable, and we emit code that chooses between the None and InvalidateReprofile deoptimization actions at runtime. When such a deoptimization site is reached and the corresponding deoptimization counter still does not exceed the deoptsTolerated threshold, the deoptimization just switches to the interpreter and keeps the compiled code as-is (the None action). Otherwise the deoptimization invalidates the code and resets the hotness counters of the corresponding method to force reprofiling (the InvalidateReprofile action).

To avoid using a stale deoptimization profile during application phase changes, the counters for deoptimization sites involved in a particular compilation are aged in each compilation. Alternatively, we provide an option to age the deoptimization profile periodically, which allows tolerating deoptimizations based on rates, instead of absolute numbers.

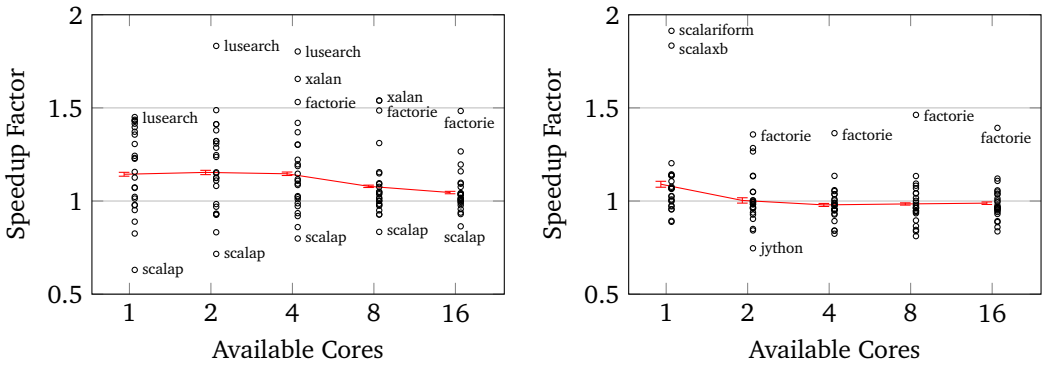
5.5 Performance Evaluation

We now evaluate performance of the two alternative strategies and compare them to the default strategy used by Graal. Using the same set of benchmarks and the same hardware platform as presented in Section 5.3, we evaluate the deoptimization strategies with a varying number of CPU cores available to the JVM. To minimize interference due to compilation of Graal classes, we enable bootstrapping of Graal[30] at JVM startup.

Because the DaCapo and ScalaBench benchmark suites are similar (ScalaBench uses the DaCapo benchmarking harness), we present the results for these two benchmark suites separately from the results for the Octane benchmarks on Graal.js, which are not directly comparable to the results from the other two suites. We also subject the results from the DaCapo and ScalaBench benchmark suites to more extensive evaluation, whereas the results for the Octane benchmarks are meant to illustrate the indirect impact of deoptimization strategies on the performance of the hosted language (JavaScript).

To illustrate the variability in the data observed during multiple benchmark runs, we use statistical bootstrap with 50000 replicas to calculate 99% confidence

[30] Enabled by the -XX:+BootstrapJVMCI option.



(a) Startup (b) Steady-state

Figure 5.4. Speed-up factors for startup and steady-state performance of Graal compared to C2 when running the DaCapo and ScalaBench benchmarks with increasing number of CPU cores available to the JVM.

interval of the geometric mean and (where applicable) the difference of means. In the plots below, we indicate the confidence interval using "whiskers" around the respective values.

5.5.1 DaCapo and ScalaBench Evaluation

To evaluate the impact of the deoptimization strategies on the performance of the benchmarks from the DaCapo and ScalaBench benchmark suites, we collect the following performance metrics: (1) startup time, i.e., the wall-clock time for the execution of the first benchmark iteration, (2) steady-state execution time, i.e., the wall-clock time for the execution of the last[31] benchmark iteration, and (3) compilation time in each iteration, i.e., the CPU time spent in compiler threads during the benchmark iteration.

For each benchmark from the DaCapo and ScalaBench benchmark suites, we show both the speed-up factors for the individual benchmarks as well as the overall speed-up factor calculated as a geometric mean of the individual speed-up factors.

Choosing the Baseline

The choice of the baseline for evaluating the performance of the alternative deoptimization strategies in Graal deserves a justification. Because changes were

[31] All the benchmarks were run for at least 10 iterations and 10 seconds.

made to the original Graal implementation, using HotSpot with Graal in place of the server compiler is our default choice. However, the production configuration of the HotSpot JVM still uses the C2 server compiler in the last compilation tier, which makes C2 a candidate for a performance baseline. Moreover, reporting changes against a well-known HotSpot configuration can help assessing the relevance of the presented changes.

A potential problem may arise if the Graal baseline was significantly slower than C2. Any performance improvements would be reported against a slow baseline, but the peak performance might not reach or exceed that of C2. To resolve this tension, we evaluate the relative performance of the two potential baselines, C2 and Graal, using the same set of DaCapo and ScalaBench benchmarks used in the evaluation of the alternative deoptimization strategies.

The results of this comparison for different number of cores available to the JVM are shown in Figure 5.4. In the case of startup performance (Figure 5.4a), we can observe that on average, the Graal baseline outperforms the C2 baseline. We attribute this to the fact that we enable bootstrapping of the Graal compiler, which may not only precompile methods from the Graal compiler itself, but also precompile frequently executed methods from the Java class library.

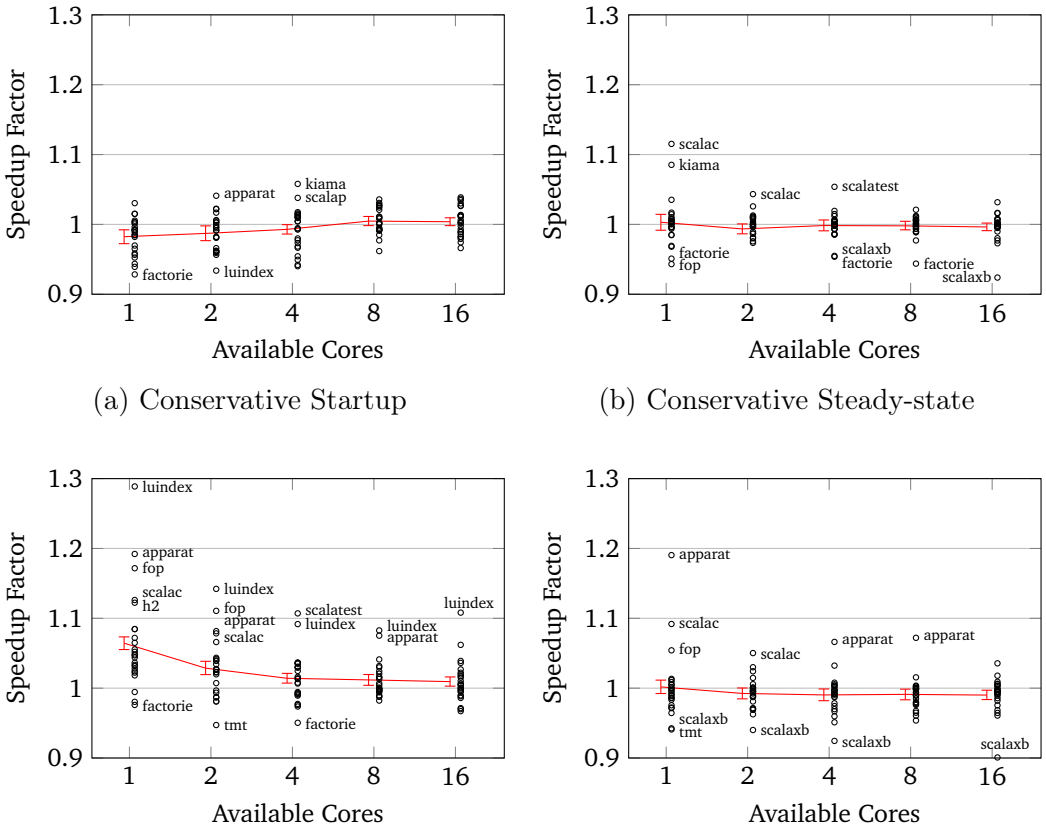
On the other hand, the plot depicting steady state performance (Figure 5.4b) shows that on average, the Graal baseline becomes slightly slower (2.1% in the worst case for 4 cores, with speed-up factor of 0.979 and 99% confidence interval of [0.971, 0.987]) than C2 as more CPU cores are made available to the JVM. The single-core case is an exception in which Graal outperforms C2 by 9% (speed-up factor of 1.090 with a 99% confidence interval of [1.074, 1.106]).

This experiment validates our choice of baseline — Graal is a competitive compiler for our workload.

Start-up Performance

In Figure 5.5, we show the results of the startup and steady-state performance evaluation for the benchmarks from the DaCapo and ScalaBench suites. The default deoptimization strategy used by Graal represents the baseline, and the plots show the speed-up factor of the *conservative* and the *adaptive* strategies with respect to the default strategy. The line connecting the overall speed-up factors illustrates the trend of the overall speed-up as the number of CPU cores available to the JVM increases.

The plot in Figure 5.5a shows that in the single-core case, the *conservative* strategy is approximately 1.8% slower than the default strategy (speed-up factor of 0.982 with 99% confidence interval of [0.972, 0.992]).



(a) Conservative Startup

(b) Conservative Steady-state

(c) Adaptive Startup

(d) Adaptive Steady-state

Figure 5.5. Speed-up factors for the startup and steady-state performance when running DaCapo and ScalaBench benchmarks with increasing number of CPU cores.

As the number of available CPU cores increases, the single-core slowdown becomes a very slight speed-up for 16 cores. The *conservative* strategy apparently causes more compilation work and benefits from the ability to hide compilation latency as much as the default strategy. Apparently, while tolerating some deoptimizations may provide a slight performance benefit, in this case it is completely outweighed by the extra compilation work.

In contrast, Figure 5.5c shows that the *adaptive* strategy is faster by approximately 6.4% in the single-core case (speed-up factor of 1.064 with 99% confidence interval of [1.055, 1.073]), and is on average slightly faster than the default strategy. The *adaptive* strategy causes less compilation work, improving startup performance on average, but the benefit diminishes with the increasing number of available CPU cores, because the (baseline) default strategy can hide

some its compilation latency.

The two alternative strategies differ mainly in the level of tolerance for deoptimizations, the accuracy of the deoptimization profile used to make decisions, and the deoptimization actions taken. The *conservative* strategy actually makes the compiler less sensitive to changes in profiling information during startup. On the one hand, the RecompileIfTooManyDeopts deoptimization used by the *conservative* strategy delays recompilation, but on the other hand it causes methods to be recompiled without being thoroughly reprofiled. Recall also that unlike the *adaptive* strategy, the *conservative* strategy associates deoptimization profile with the target of a deoptimization, not its origin. This impairs the ability to tolerate rare deoptimizations but deal with deoptimizations that are repeatedly triggered at the same deoptimization site. Due to inlining, the code triggering the deoptimizations may be duplicated in different methods and target different deoptimization traps, spreading the information about a single deoptimization site among different profiles.

The effect of tolerating deoptimizations is clearly workload dependent, and the results show a few interesting cases. The luindex benchmark clearly benefits from the *adaptive* strategy, as it exhibits a speed-up factor of 1.289 in the single-core case, and a speed-up factor of at least 1.082 throughout the whole experiment. Interestingly, it does not benefit from the *conservative* strategy, exhibiting a slow-down (speed-up factor of 0.965) in the single-core case, even though the compilation times for both strategies are similar.

In contrast to luindex, the factorie benchmark does not benefit from either of the strategies, exhibiting slowdowns (speed-up factors from 0.928 to 0.996 throughout the experiment. Further investigation shows that the slowdown results from an increased number of deoptimizations which may result in more time spent in the interpreter.

Steady-state Performance

The plots in Figure 5.5b and Figure 5.5d show the steady-state performance for both strategies. Even though the results for individual benchmarks may differ slightly, on average the steady-state performance of the *conservative* strategy does not really differ from the default strategy.

In the case of the *adaptive* strategy, the overall speed-up factor remains slightly below 1 as the number of CPU cores increases. We attribute this to the fact that unlike the *conservative* strategy, which is supported by the HotSpot runtime and attempts to store all profiling data efficiently, the implementation of the *adaptive* strategy is far from optimized. It uses more memory to store profiling data, and

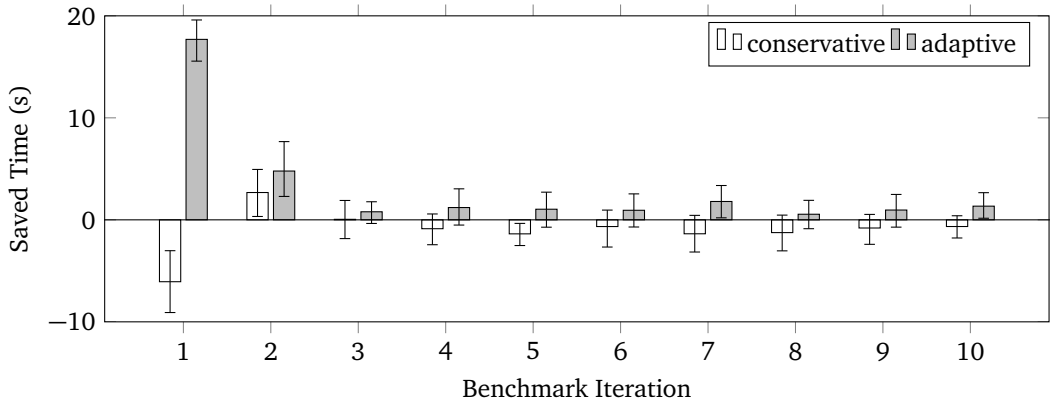


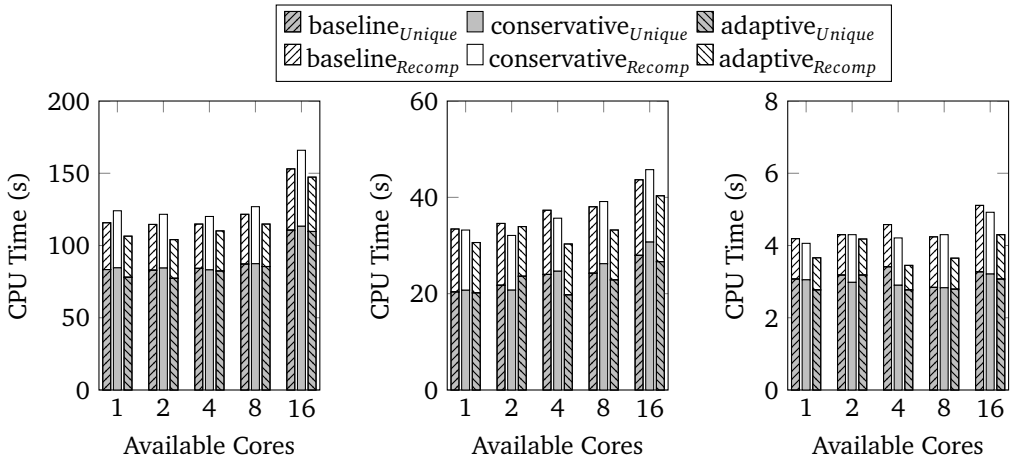
Figure 5.6. Total saved execution time for the selected 24 DaCapo and ScalaBench benchmarks in single-core setup. Negative values represent a slowdown.

emits conditional code and a volatile memory access at deoptimization sites that select deoptimization action at runtime. We expect this to impact performance, especially given the memory barriers associated with the volatile memory access and the increasing number of CPU cores.

For some benchmarks, the increased tolerance to deoptimizations is beneficial even during steady-state execution. The scalac benchmark benefits from both strategies in the single-core and dual-core configurations, exhibiting a performance improvement of 9.4% (single-core) or 5% (dual-core) over the baseline with the *adaptive* strategy, and 11.6% (single-core) or 4.4% (dual-core) with the *conservative* strategy in the single-core case. The apparat benchmark benefits from the *adaptive* strategy even in 4 or 8-core configurations, which we attribute to the aging of the deoptimization profile. On the other hand, benchmarks such as tmt exhibit an average 4% slow-down in steady-state performance for all core configurations. Short-running benchmarks (less than 300ms) such as fop and scalaxb have a tendency to amplify speed-ups and slow-downs, so they appear to be outliers in the plots.

Overall Execution and Compilation Time

Figure 5.6 shows the amount of execution time saved for the 24 benchmarks from the DaCapo and ScalaBench suites together in a single-core configuration. When considering the total execution time, the execution time of each benchmark provides a weight to its respective speed-up or slow-down. With the *adaptive* strategy, the first iteration of all benchmarks finishes 17.7 seconds earlier than with the default strategy (which required 337 seconds in total), resulting in a



(a) 1^{st} Iteration (b) 2^{nd} Iteration (c) 10^{th} Iteration

Figure 5.7. The total CPU time spent compiling (□+■) and recompiling (□) when executing the 24 selected DaCapo and ScalaBench benchmarks.

speed-up factor of 1.053. With the *conservative* strategy, the first iteration takes 6 seconds longer than with the default strategy, resulting in a speed-up factor of 0.982. Note that these speed-up factors implicitly weigh the speed-up achieved for individual benchmarks by the execution time of each benchmark, giving a more conservative estimate than the geometric mean of speed-up factors, which treats all benchmarks with equal weight. The improvement observable with the *adaptive* strategy diminishes with the increasing number of available CPU cores, but the *adaptive* strategy still manages to save some time on each iteration, which would accumulate in the long run. Considering the overall execution time shows that the *adaptive* strategy does not necessarily hurt steady-state performance, as the results discussed in Section 5.5.1 may suggest.

Even though there are benefits in avoiding repeated deoptimizations, the influence of the increasing number of CPU cores on the performance results suggests that the differences in performance can be mostly attributed to compilation. To support this observation, Figure 5.7 provides a summary of the compilation log for all strategies. The data shows that indeed the *adaptive* strategy saves approximately 8% in the total compilation time compared to the default strategy in the first iteration of the single-core scenario, which benefits the most. The *alternative* strategy mostly saves time in all scenarios, but the impact on total execution time diminishes with increased number of cores available, and in steady-state execution. In contrast, the *conservative* strategy is apparently not a good fit for the first iteration, because it creates more compilation work. It saves some compilation

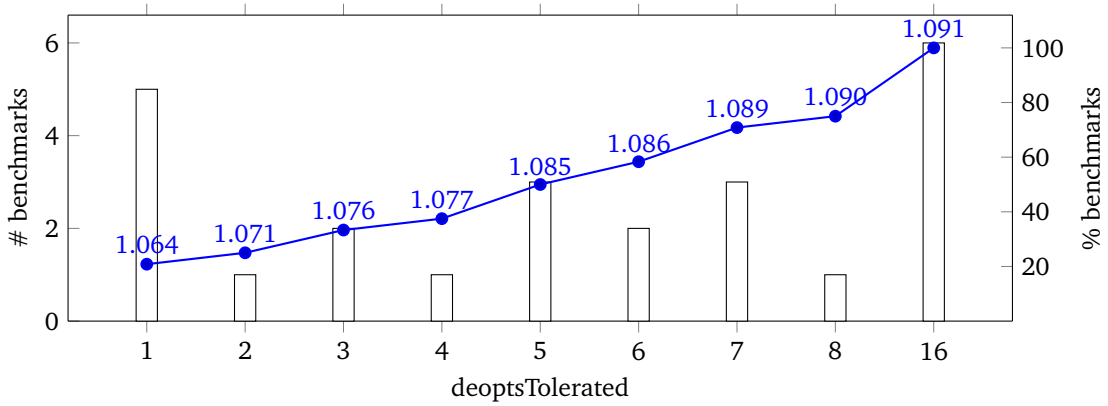


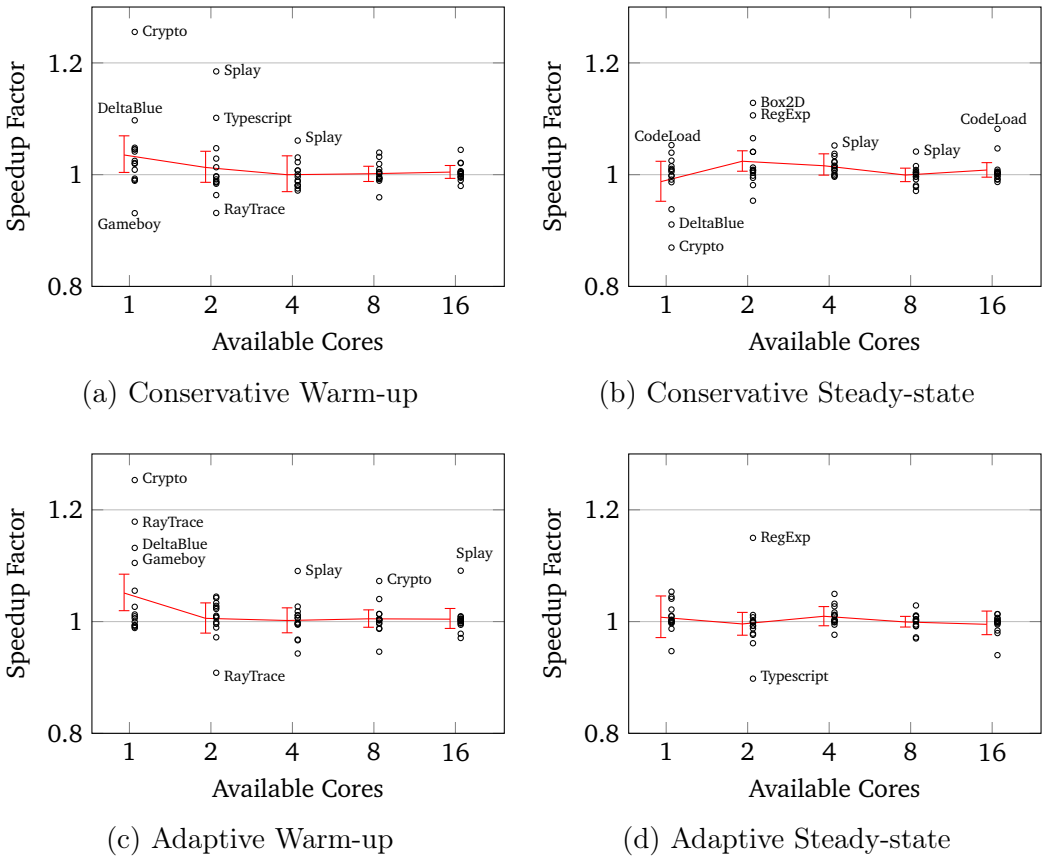
Figure 5.8. Theoretical speed-up with optimal values of deoptsTolerated for each of the 24 selected DaCapo and ScalaBench benchmarks. The bars represent the number of benchmarks for which the value was optimal. The line connecting the dots shows cumulative percentage of benchmarks for which the optimal threshold does not exceed the given value. Associated with each dot is the overall speed-up factor the would be achieved if we managed to choose an optimal threshold for each benchmark not exceeding the given value.

time in later iterations, but too little too late.

Tolerance for Deoptimizations in the Adaptive Strategy

The tolerance of the *adaptive* strategy to deoptimizations can be adjusted by changing the deoptsTolerated and deoptsAllowed thresholds (c.f. Subsection 5.4.2). The results presented so far were obtained with the default values, but we are interested in how different levels of tolerance to deoptimizations impact performance of the strategy. Because the strategy had the most effect on the 1st benchmark iteration in the single-core configuration, we evaluated the performance of the *adaptive* strategy with the deoptsTolerated threshold set to 1–8, and 16. We analyzed the speed-up factors of individual benchmarks for all tested values of the deoptsTolerated threshold, and selected the threshold value resulting in maximal speed-up factor as optimal for each benchmark.

The tolerance to deoptimizations, and thus the value of the deoptsTolerated threshold, is clearly a property of a particular workload and represents a tuning parameter. If we were able to (quickly) determine the appropriate threshold based on the character of the workload being executed, the parameter could be adjusted in response to program behavior. To gauge the potential for improvement, Figure 5.8 shows the theoretical speed-up factor that could be achieved, if we



(a) Conservative Warm-up

(b) Conservative Steady-state

(c) Adaptive Warm-up

(d) Adaptive Steady-state

Figure 5.9. Speed-up factors for the warm-up and steady-state performance when running Octane benchmarks on Graal.js with increasing number of CPU cores.

managed to find the optimal deoptsTolerated threshold (within a given limit) for each benchmark. The plot shows that searching for an optimal threshold in the range of 1–5 would provide an optimal value for approximately 50% of benchmarks (given the upper bound of 16), and yield a speed-up factor of 1.085.

5.5.2 Octane on Graal.js Evaluation

To evaluate the performance of the Octane benchmarks running on Graal.js, we use the benchmarking harness for the Octane suite provided in the GraalVM binary release[32]. The harness uses benchmark-specific warm-up times ranging from 15 to 120 seconds, and a common steady-state period of 10 seconds. When

[32]http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html

finished executing a benchmark, the harness reports per-iteration execution times achieved during the warm-up and steady-state periods. We collect the per-iteration execution times for both phases and report the speed-up factors w.r.t. the default deoptimization strategy. Because the warm-up and steady-state phases are defined differently for the Octane suite than for the DaCapo and ScalaBench suites, we report the speed-up factors separately.

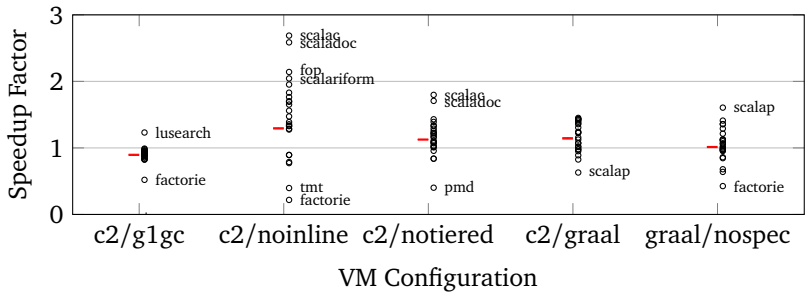
The plots in Figure 5.9a and Figure 5.9c show the warm-up performance of Octane benchmarks running on Graal.js with the *conservative* strategy and the *adaptive* strategy, respectively. We again show the speed-up factors for the individual benchmarks and the overall speed-up factor calculated as a geometric mean of the individual speed-up factors. In the single core case, both alternative deoptimization strategies achieve better warm-up performance than the default strategy. On average, the *conservative* strategy is approximately 3.5% faster (speed-up factor of 1.035 with 99% confidence interval of [1.004, 1.069]) and the *adaptive* strategy is approximately 5.1% faster (speed-up factor of 1.051 with 99% confidence interval of [1.020, 1.085]) than the default strategy.

The results indicate that the JavaScript runtime implemented using the Truffle framework generally benefits from tolerating deoptimizations during startup due to the reduction of the compilation work. This is potentially beneficial for JavaScript workloads that mostly execute code once, instead of repeatedly. However, similarly to the DaCapo and ScalaBench benchmarks, the benefit diminishes as the number of available CPU cores increases. Even though JavaScript is a single-threaded language, the runtime may use additional CPU cores to hide compilation latency.

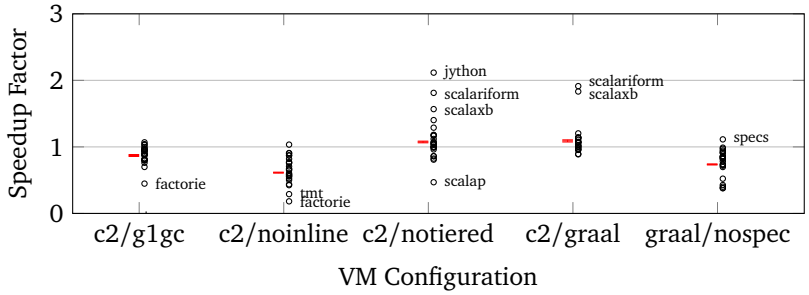
Finally the plots in Figure 5.9b and Figure 5.9d show the steady-state performance of Octane benchmarks with the *conservative* and *adaptive* strategies, respectively. Neither of them deviates from the performance of the default strategy in a significant way.

5.5.3 On the Scale of Performance Changes

The results of performance evaluation indicate that on average the *adaptive* deoptimization strategy provides moderate improvements to startup performance in a single-core scenario. As the number of cores and benchmark runtime increases, the effect wears off, until it disappears. Because the improvement is moderate, it is difficult to assess how it fits the overall picture. In his 1974 paper, Knuth notes that in established engineering disciplines, 12% improvement, easily obtained, is never considered marginal [55]. The improvements obtained here are roughly half of that, but still rather easily obtained, given the complexity of the other



(a) Startup (1-core)



(b) Steady-state (1-core)

Figure 5.10. Speed-up factors for the startup and steady-state performance of different VM configurations compared to their respective baseline.

parts of the VM.

Arguably, the execution time aspect of the improvement diminishes with more CPU cores available to the JVM, but the computation saved remains. To provide a frame of reference, we evaluate the single-core performance of five different configurations of the HotSpot VM and compare it with their respective baselines. Two of the configuration changes swap entire VM subsystems, while three other changes alter the behavior of the dynamic compiler.
The first baseline is the default configuration of HotSpot equipped with C2 to which we compare the performance obtained with the following configurations:

g1gc Replaces the default garbage collector in HotSpot with the Garbage First (G1) garbage collector.

noinline Disables inlining in C2.

notiered Disables tiered compilation in HotSpot, i.e., disables C1 compiler.

graal Replaces the C2 server compiler with Graal.

The second baseline is the default configuration of HotSpot equipped with Graal to which we compare the following configuration:

nospec Disables the majority of speculative optimizations relying on deoptimization in Graal, providing a rough estimate the the performance gains enabled by deoptimization.

The results of the evaluation are shown in Figure 5.10. The subfigures correspond to startup (Figure 5.10a) and steady state (Figure 5.10b) performance, each showing average performance of the first four configurations compared to the C2 baseline, followed by the fifth configuration compared to the Graal baseline.

In a single-core setting, the change of the GC algorithm caused a 10.4% drop in startup performance, and a 12.9% drop in steady-state performance. We are aware that this results from different mode of operation of the G1 collector, which is typically recommended for heaps exceeding 6 GB. However, it illustrates the kind of performance impact a careless swap of a GC may have in a particular scenario.

The *noinline* and *nospec* configurations reduce the amount of compilation work, either due to avoiding redundant compilation of methods that could have been inlined, or due to compiling immediately using the highest-tier compiler. Consequently, we observe a significantly better startup performance in the single-core scenario—29.4% improvement due to disabled inlining, and 12.6% due to disabled tiered compilation. In steady state, disabled tiered compilation retains a 7.4% performance improvement, but disabled inlining changes the situation dramatically. Because inlining is a critical optimization that increases optimization scope and effectively enables inter-procedural optimization, disabling inlining causes a 40% drop in steady-state performance.

The *graal* configuration illustrates the effect of replacing a C2 server compiler with Graal. We have explored this situation more closely in Section 5.5.1, here we just note a 14.4% improvement in startup performance, and 9% improvement in steady-state performance.

Finally, the *nospec* configuration illustrates the effect of disabling various speculative optimizations in the Graal compiler. These include elimination of unreached branches, heuristic inlining, speculative instanceof test, elimination of unreached exception handlers, and elimination of safepoints within a loop. On average, this change appears to have neutral impact on startup performance, but has a significant impact later, resulting in a 26.1% performance drop in the steady state.

This suggests that the above speculative optimizations pay off in the long term, but do not provide much benefit at startup. The *adaptive* strategy complements this

by providing a moderate improvement in startup performance without adversely affecting performance in the long term.

5.6 Discussion

Below, we discuss the benefits and limitations about our study.

Representativeness. Our study is conducted only on a Graal-enabled HotSpot VM. We choose the Graal compiler because it aggressively employs speculative optimizations, which potentially lead to more deoptimizations. Indeed, our profiler reports a considerable amount of deoptimizations across the application code and the Java class library, and reveals some buggy behavior of repeated deoptimizations and recompilations in the Graal compiler's emitted code. Still, some of our findings can serve as a starting point for experiments on other VMs. If they heavily rely on speculation, the accuracy of the deoptimization profile matters for an adaptive recompilation mechanism that tolerates infrequent deoptimizations. Such mechanism is often essential in a device with limited resource such as a mobile phone.

Performance. Unlike the default deoptimization strategy and the conservative strategy, the adaptive strategy maintains its profile separately in addition to HotSpot VM's built-in profile. For a consistent access of the customized profile, we insert memory barriers at the places where the conditional code selects different deoptimization actions based on the profile. Even though deoptimization should occur infrequently, the inserted memory barriers may negatively affect performance, especially with an increasing number of CPU cores. Moreover, the increased memory usage on the Java heap due to the customized profile may lead to increase the garbage-collection costs. Hence, while the adaptive strategy outperforms the other two strategies during the startup phase, there is still room for improvement by optimizing the data structure keeping the accurate deoptimization profile.

5.7 Summary

In this chapter we build a deoptimization profiler based on the IR profiling technique presented in Chapter 4, and conduct a study of deoptimization behavior in benchmarks executing on a Graal-enabled HotSpot VM. We profile deoptimization sites in the code produced by the Graal compiler, and provide a qualitative and quantitative analysis of deoptimization causes in commonly used benchmark

suites such as DaCapo, ScalaBench, and Octane, which provide workloads derived from real applications and libraries written in Java, Python, Scala, and JavaScript. We show that only a small fraction of deoptimization sites actually trigger deoptimizations at runtime, and that most of the deoptimizations actually triggered in Graal-compiled code unconditionally invalidate and reprofile the method which caused a deoptimization.

To gain insight on the trade-offs made by Graal in its default deoptimization strategy, we modify Graal to add support for two alternative deoptimization strategies and evaluate benchmark performance using the three strategies. We show that by avoiding the *conservative* strategy provided by the HotSpot VM runtime, Graal gains better startup performance. However, we also show that certain tolerance to deoptimizations can provide performance benefits, if used with a precise deoptimization profile. The *adaptive* strategy, which switches among various deoptimization actions based on a precise deoptimization profile, manages to reduce the amount of method recompilations and eliminate certain repetitive deoptimizations. As a result, on a single-core system, it improves the average start-up performance by 6.4% in the DaCapo and ScalaBench benchmarks, and by 5.1% in the Octane benchmarks.

Finally, we show that tolerance to deoptimizations is a workload-specific parameter, and that finding correlation between some workload characteristics and the appropriate level of tolerance to deoptimizations can potentially provide additional performance benefits.

Chapter 6

Conclusion

Profiling is a generally adopted technique to reason about program behavior during execution. In a managed runtime system, prevailing profilers either yield wrong results or are unable to intercept low-level operations in the presence of dynamic compilation. The reasons are twofold.

Firstly, profilers based on bytecode instrumentation are not aware of the optimizations performed by the dynamic compiler, which in turn is not able to distinguish the base-program code from the inserted instrumentation code. Hence, the inserted code either over-profiles the optimized code or perturbs the optimizations. In both situations, the resulting profile will be inaccurate.

Secondly, prevailing instrumentation techniques suffer from limitations on intercepting IR-level operations. In the case of bytecode instrumentation, it is not possible to intercept IR-level operations that do not have an associated originating bytecode. Binary instrumentation lacks a mapping from the collected profile of program behavior at the machine-code level to higher-level operations, and thus may result in profiles that are not actionable.

In this dissertation, we tackle the problems on profiling in the presence of dynamic compilation in two ways. To address over-profiling and perturbation of optimizations in bytecode instrumentation, we introduce a 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. To address the inability of intercepting IR-level operations, we introduce a technique to perform instrumentation at the IR level.

6.1 Summary of Contributions

Below we summarize the contributions of this dissertation.

Accurate Bytecode Profiling. We present a new technique to make profilers aware of dynamic compiler optimizations and to avoid perturbation of the optimizations. We implement our technique in Oracle's Graal compiler and integrate it into the Graal project in OpenJDK. We provide a set of query intrinsics for retrieving the optimization decisions within inserted profiling code.

We present profilers to explore the impact of (partial) escape analysis and stack allocation on heap usage and object lifetime, and to explore the impact of method inlining on callsite profiling, demonstrating that our approach helps improve the accuracy of existing bytecode-instrumentation-based tools. We present tools to identify inlining opportunities, and to study the impact of method inlining considering varying levels of calling context, demonstrating that our approach enables new tools that can help further improve the optimizations performed by dynamic compilers. 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. Our testing framework has already helped the developers of Graal to locate and fix performance bugs in their compiler.

IR Profiling. We present an event-based framework to support profiling of the IR-level operations used during dynamic compilation. We implement our framework in Oracle's Graal compiler, together with the accurate bytecode-level profiling technique.

We present a characterization study on the memory-barrier usage in the code produced by Graal for the DaCapo and ScalaBench benchmark suites. We also present an empirical study on the deoptimization causes in the code produced by Graal for the DaCapo, ScalaBench, and Octane benchmark suites. We provide two additional feedback-directed deoptimization strategies. We evaluate the performance of both deoptimization strategies and compare them to the default strategy used by Graal. We observe an improvement of the start-up performance with our adaptive deoptimization strategy. We also find that the choice of a deoptimization strategy has negligible impact on steady-state performance. This indicates that the cost of speculation matters mainly during start-up, where it can disturb the delicate balance between executing the program and compilation, but is quickly amortized in steady state.

6.2 Future Work

The work presented in this dissertation opens several future research directions. Below, we give an overview of our research plans:

Flexible and Extensible IR Profiling. We have presented an event-based IR profiling framework, which predefines all the IR events that are one-to-one mapped to the IR node types. We plan to make the framework easily extensibile. Firstly, we would allow the definition of events with certain traits, e.g., an event indicating a conditional within a loop. Secondly, we would like to support custom IR-level events representing IR subgraphs with specific properties instead of only single IR nodes.

Inspection of Compiler Optimizations. One of the key reasons for introducing IR profiling is that it allows one to inspect compiler optimizations. For example, the compiler developer may be interested in finding out the efficiency of the loop unswitching optimization, which moves a conditional outside of a loop. This can be achieved by profiling the executed number of conditionals that are unswitched. To this end, we need a mechanism that allows the profiler developer to perform static analysis on the IR graph before and after specific compiler phases.

Enhancing Existing Tools. Prevailing tools based on bytecode instrumentation may emit a misleading profile due to perturbations or the inability of precisely intercepting low-level operations. Our accurate bytecode profiling technique and the IR profiling technique address these problems. A possible future direction would be to enhance existing tools to use our techniques and evaluate the improved accuracy of the new profilers. One possible target is the modeling of garbage-collector behavior based on program traces, as prevailing techniques observe significant inaccuracies due to perturbations caused by inserted instrumentation code [66].

Bibliography

[1] Ammons, G., Ball, T., Larus, J.R.: Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation. pp. 85–96. PLDI '97, ACM, New York, NY, USA (1997)

[2] Anderson, J.M., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A., Weihl, W.E.: Continuous Profiling: Where Have All the Cycles Gone? ACM Trans. Comput. Syst. 15(4), 357–390 (Nov 1997)

[3] 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 (Feb 2005)

[4] 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, New York, NY, USA (2000)

[5] 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, New York, NY, USA (2000)

[6] Arnold, M., Grove, D.: Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 51–62. CGO '05, IEEE Computer Society, Washington, DC, USA (2005)

[7] 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, New York, NY, USA (2002)

[8] 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, New York, NY, USA (2001)

[9] Arnold, M., Sweeney, P.F.: Approximating the calling context tree via sampling. Tech. rep., IBM Research (2000)

[10] Ball, T., Larus, J.R.: Efficient Path Profiling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture. pp. 46–57. MICRO 29, IEEE Computer Society, Washington, DC, USA (1996)

[11] Binder, W.: Portable and Accurate Sampling Profiling for Java. Softw. Pract. Exper. 36(6), 615–650 (May 2006)

[12] 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, New York, NY, USA (2006)

[13] 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, New York, NY, USA (1999)

[14] Blanchet, B.: Escape Analysis for Java: Theory and Practice. ACM Trans. Program. Lang. Syst. 25(6), 713–775 (Nov 2003)

[15] Bond, M.D., McKinley, K.S.: Continuous Path and Edge Profiling. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 130–140. MICRO 38, IEEE Computer Society, Washington, DC, USA (2005)

[16] Bruening, D., Garnett, T., Amarasinghe, S.: An Infrastructure for Adaptive Dynamic Optimization. In: Proceedings of the International Symposium

on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 265–275. CGO '03, IEEE Computer Society, Washington, DC, USA (2003)

[17] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and extensible component systems (2002)

[18] 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, New York, NY, USA (1999)

[19] Buytaert, D., Georges, A., Hind, M., Arnold, M., Eeckhout, L., De Bosschere, K.: Using Hpm-sampling to Drive Dynamic Compilation. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 553–568. OOPSLA '07, ACM, New York, NY, USA (2007)

[20] Chambers, C., Ungar, D.: Making pure object-oriented languages practical. In: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications. pp. 1–15. OOPSLA '91, ACM, New York, NY, USA (1991)

[21] Chiba, S.: Load-Time Structural Reflection in Java. In: Proceedings of the 14th European Conference on Object-Oriented Programming. pp. 313–336. ECOOP '00, Springer-Verlag, London, UK, UK (2000)

[22] 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, New York, NY, USA (1999)

[23] 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, New York, NY, USA (1995)

[24] Dahm, M.: Byte Code Engineering, pp. 267–277. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

[25] 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, London, UK, UK (1999)

[26] Deutsch, L.P., Schiffman, A.M.: Efficient Implementation of the Smalltalk-80 System. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 297–302. POPL '84, ACM, New York, NY, USA (1984)

[27] Dmitriev, M.: Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation. Tech. rep., Mountain View, CA, USA (2003)

[28] 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, New York, NY, USA (2013)

[29] Duesterwald, E., Bala, V.: Software Profiling for Hot Path Prediction: Less is More. SIGPLAN Not. 35(11), 202–211 (Nov 2000)

[30] Fang, L., Dou, L., Xu, G.: PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming (ECOOP 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 37, pp. 296–320. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)

[31] Feeley, M.: Polling Efficiently on Stock Hardware. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. pp. 179–187. FPCA '93, ACM, New York, NY, USA (1993)

[32] 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 (Jul 1987)

[33] Fink, S.J., Qian, F.: Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In: International Symposium on Code Generation and Optimization, 2003. CGO 2003. pp. 241–252. CGO '03, IEEE Computer Society, Washington, DC, USA (March 2003)

[34] Flanagan, C., Freund, S.N.: FastTrack: Efficient and Precise Dynamic Race Detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–133. PLDI '09, ACM, New York, NY, USA (2009)

[35] Flanagan, C., Freund, S.N.: The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 1–8. PASTE '10, ACM, New York, NY, USA (2010)

[36] Froyd, N., Mellor-Crummey, J., Fowler, R.: Low-overhead Call Path Profiling of Unmodified, Optimized Code. In: Proceedings of the 19th Annual International Conference on Supercomputing. pp. 81–90. ICS '05, ACM, New York, NY, USA (2005)

[37] Google: Octane 2.0 JavaScript Benchmark. `https://developers.google.com/octane/`

[38] Google: The V8 JavaScript engine. `https://developers.google.com/v8/`

[39] Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A Call Graph Execution Profiler. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction. pp. 120–126. SIGPLAN '82, ACM, New York, NY, USA (1982)

[40] Gu, D., Verbrugge, C.: Phase-based Adaptive Recompilation in a JVM. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 24–34. CGO '08, ACM, New York, NY, USA (2008)

[41] 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 (Feb 2014)

[42] Hauswirth, M., Sweeney, P.F., Diwan, A., Hind, M.: Vertical Profiling: Understanding the Behavior of Object-priented Applications. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 251–269. OOPSLA '04, ACM, New York, NY, USA (2004)

[43] 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, Washington, DC, USA (2003)

[44] 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 (May 2006)

[45] Hirzel, M., Chilimbi, T.: Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In: 4th ACM Workshop on Feedback-Directed and Dynamic Optimization. pp. 117–126. FDDO '01 (2001)

[46] Hofer, P., Gnedt, D., Mössenböck, H.: Lightweight Java Profiling with Partial Safepoints and Incremental Stack Tracing. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. pp. 75–86. ICPE '15, ACM, New York, NY, USA (2015)

[47] 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, New York, NY, USA (1992)

[48] Hölzle, U., Ungar, D.: Reconciling Responsiveness with Performance in Pure Object-oriented Languages. ACM Trans. Program. Lang. Syst. 18(4), 355–400 (Jul 1996)

[49] IBM: J9 Virtual Machine. `https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html`

[50] IBM: T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php/Main_Page`

[51] Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)

[52] Kedlaya, M.N., Robatmili, B., Caşcaval, C., Hardekopf, B.: Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 103–114. VEE '14, ACM, New York, NY, USA (2014)

[53] 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, New York, NY, USA (2012)

[54] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming. pp. 327–353. ECOOP '01, Springer-Verlag, London, UK, UK (2001)

[55] Knuth, D.E.: Structured programming with go to statements. ACM Comput. Surv. 6(4), 261–301 (Dec 1974)

[56] 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, New York, NY, USA (2005)

[57] 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, Washington, DC, USA (2007)

[58] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java HotSpot™ Client Compiler for Java 6. ACM Trans. Archit. Code Optim. 5(1), 7:1–7:32 (May 2008)

[59] Krintz, C.J., Grove, D., Sarkar, V., Calder, B.: Reducing the overhead of dynamic compilation. Software: Practice and Experience 31(8), 717–738 (2001)

[60] Kulkarni, P., Arnold, M., Hind, M.: Dynamic Compilation: The Benefits of Early Investing. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. pp. 94–104. VEE '07, ACM, New York, NY, USA (2007)

[61] Kulkarni, P.A.: JIT Compilation Policy for Modern Machines. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 773–788. OOPSLA '11, ACM, New York, NY, USA (2011)

[62] Langdale, G., Gross, T.: Evaluating the Relationship Between the Usefulness and Accuracy of Profiles. In: Proc. Workshop on Duplicating, Deconstructing, and Debunking (2003)

[63] Lea, D.: The JSR-133 Cookbook for Compiler Writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`

[64] Levin, R., Newman, I., Haber, G.: Complementing Missing and Inaccurate Profiling Using a Minimum Cost Circulation Algorithm. In: Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers. pp. 291–304. HiPEAC'08, Springer-Verlag, Berlin, Heidelberg (2008)

[65] 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)

[66] 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, New York, NY, USA (2014)

[67] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005)

[68] Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., Qi, Z.: Introduction to Dynamic Program Analysis with DiSL. In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. pp. 429–430. ICPE '13, ACM, New York, NY, USA (2013)

[69] Marek, L., Kell, S., Zheng, Y., Bulej, L., Binder, W., Tůma, P., Ansaloni, D., Sarimbekov, A., Sewe, A.: ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform. In: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences. pp. 105–114. GPCE '13, ACM, New York, NY, USA (2013)

[70] 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, New York, NY, USA (2012)

[71] Molnar, P., Krall, A., Brandner, F.: Stack Allocation of Objects in the CACAO Virtual Machine. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 153–161. PPPJ '09, ACM, New York, NY, USA (2009)

[72] Moret, P., Binder, W., Tanter, E.: Polymorphic Bytecode Instrumentation. In: Proceedings of the Tenth International Conference on Aspect-oriented Software Development. pp. 129–140. AOSD '11, ACM, New York, NY, USA (2011)

[73] Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R.: Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 198–208. CGO '07, IEEE Computer Society, Washington, DC, USA (2007)

[74] 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, New York, NY, USA (2010)

[75] Naik, M.: Chord: A Program Analysis Platform for Java. `http://www.seas.upenn.edu/~mhnaik/chord.html`

[76] Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89–100. PLDI '07, ACM, New York, NY, USA (2007)

[77] Oracle: Graal project. `http://openjdk.java.net/projects/graal/`

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

[79] Oracle: Java virtual machine tool interface. `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/`

[80] Oracle: JSR 133: Java Memory Model and Thread Specification Revision. `https://jcp.org/en/jsr/detail?id=133`

[81] Oracle: Netbeans profiler. `https://profiler.netbeans.org`

[82] Paleczny, M., Vick, C., Click, C.: The Java HotSpot™ Server Compiler. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1. pp. 1–1. JVM'01, USENIX Association, Berkeley, CA, USA (2001)

[83] Pettis, K., Hansen, R.C.: Profile Guided Code Positioning. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. pp. 16–27. PLDI '90, ACM, New York, NY, USA (1990)

[84] 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, New York, NY, USA (2013)

[85] Sarimbekov, A., Stadler, L., Bulej, L., Sewe, A., Podzimek, A., Zheng, Y., Binder, W.: Workload Characterization of JVM Languages. Softw. Pract. Exper. 46(8), 1053–1089 (Aug 2016)

[86] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. pp. 27–37. SOSP '97, ACM, New York, NY, USA (1997)

[87] 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)

[88] 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, New York, NY, USA (2011)

[89] Shali, A., Cook, W.R.: Hybrid Partial Evaluation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 375–390. OOPSLA '11, ACM, New York, NY, USA (2011)

[90] 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 (Jun 2015)

[91] 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, New York, NY, USA (2000)

[92] SPEC: SPEC95. `http://www.specbench.org/osg/cpu95/`

[93] 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, New York, NY, USA (2014)

[94] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., Nakatani, T.: Design and Evaluation of Dynamic Optimizations for a Java Just-in-time Compiler. ACM Trans. Program. Lang. Syst. 27(4), 732–785 (Jul 2005)

[95] Sweeney, P.F., Hauswirth, M., Cahoon, B., Cheng, P., Diwan, A., Grove, D., Hind, M.: Using Hardware Performance Monitors to Understand the Behavior of Java Applications. In: Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3. pp. 5–5. VM'04, USENIX Association, Berkeley, CA, USA (2004)

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

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

[98] Tian, K., Zhang, E., Shen, X.: A step towards transparent integration of input-consciousness into dynamic program optimizations. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 445–462. OOPSLA '11, ACM, New York, NY, USA (2011)

[99] Vallée-Rai, R., Gagnon, E., Hendren, L.J., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Proceedings of the 9th International Conference on Compiler Construction. pp. 18–34. CC '00, Springer-Verlag, London, UK, UK (2000)

[100] Whaley, J.: A Portable Sampling-based Profiler for Java Virtual Machines. In: Proceedings of the ACM 2000 Conference on Java Grande. pp. 78–87. JAVA '00, ACM, New York, NY, USA (2000)

[101] Whaley, J.: Partial Method Compilation Using Dynamic Profile Information. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 166–179. OOPSLA '01, ACM, New York, NY, USA (2001)

[102] Wimmer, C., Jovanovic, V., Eckstein, E., Würthinger, T.: One Compiler: Deoptimization to Optimized Code. In: Proceedings of the 26th International Conference on Compiler Construction. pp. 55–64. CC 2017, ACM, New York, NY, USA (2017)

[103] Wu, B., Zhou, M., Shen, X., Gao, Y., Silvera, R., Yiu, G.: Simple Profile Rectifications Go a Long Way. In: Proceedings of the 27th European Conference on Object-Oriented Programming. pp. 654–678. ECOOP'13, Springer-Verlag, Berlin, Heidelberg (2013)

[104] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to Rule Them All. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 187–204. Onward! 2013, ACM, New York, NY, USA (2013)

[105] Xu, G.: Finding Reusable Data Structures. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 1017–1034. OOPSLA '12, ACM, New York, NY, USA (2012)

[106] 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, New York, NY, USA (2013)

[107] Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the Flow: Profiling Copies to Find Runtime Bloat. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 419–430. PLDI '09, ACM, New York, NY, USA (2009)

[108] Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., Sevitsky, G.: Finding Low-utility Data Structures. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 174–186. PLDI '10, ACM, New York, NY, USA (2010)

[109] Yan, D., Xu, G., Rountev, A.: Uncovering Performance Problems in Java Applications with Reference Propagation Profiling. In: Proceedings of the 34th International Conference on Software Engineering. pp. 134–144. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012)

[110] Yasue, T., Suganuma, T., Komatsu, H., Nakatani, T.: An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques. pp. 148–. PACT '03, IEEE Computer Society, Washington, DC, USA (2003)

[111] Zhuang, X., Serrano, M.J., Cain, H.W., Choi, J.D.: Accurate, Efficient, and Adaptive Calling Context Profiling. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 263–271. PLDI '06, ACM, New York, NY, USA (2006)