[1]

Exploring Impact of Profile Data on Code Quality in the HotSpot JVM

APRIL W. WADE, University of Kansas, USA
PRASAD A. KULKARNI, University of Kansas, USA
MICHAEL R. JANTZ, University of Tennessee, USA

Managed language virtual machines (VM) rely on dynamic or just-in-time (JIT) compilation to generate optimized native code at run-time to deliver high execution performance. Many VMs and JIT compilers collect *profile* data at run-time to enable profile-guided optimizations (PGO) that customize the generated native code to different program inputs. PGOs are generally considered integral for VMs to produce high-quality and performant native code.

In this work we study and quantify the performance benefits of PGOs, understand the importance of profiling data quantity and quality/accuracy to effectively guide PGOs, and assess the impact of individual PGOs on VM performance. The insights obtained from this work can be used to understand the current state of PGOs, develop strategies to more efficiently balance the cost and exploit the potential of PGOs, and explore the implications of and challenges for the alternative ahead-of-time (AOT) compilation model used by VMs.

Additional Key Words and Phrases: Program profiling, Profile-guided optimizations

ACM Reference Format:
April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2019. Exploring Impact of Profile Data on Code Quality in the HotSpot JVM. *ACM Trans. Embedd. Comput. Syst.* ?, ?, Article ? (October 2019), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Program *profiling* involves collecting relevant information regarding the dynamic or execution-time behavior of a program. Such information is then used by compilers to both improve the effectiveness of traditional optimizations and enable a new class of dynamic optimizations to improve the quality of generated code. Compiler transformations that employ profile information to benefit program performance are called profile-guided optimizations (PGO).

Managed language platforms, such as Java, provide an accessible, secure, platform-independent and high-performance development and run-time environment. Virtual machines (VM) for managed

[1] **Extension of Conference Paper** This work extends our conference submission titled AOT vs JIT: Impact of Profile Data on Code Quality (LCTES)[39]. We extend this earlier work by: (a) re-implementing all experiments in the JDK 9 release of the HotSpot JVM whereas the previous work was based on a development snapshot, (b) extending the set of the benchmarks used in our experiments with ScalaBench suite which provides insight into behavior of programs written in non-Java JVM languages and thus alters some of our conclusions, (c) expanding the set of inputs used in Section 5.3, (d) adding an investigation in Section 5.4 into the impact of specific optimizations implemented in HotSpot.

Authors' addresses: April W. Wade, University of Kansas, 1450 Jayhawk Blvd, Lawrence, KS, 66046, USA, aprilwade@ku.edu; Prasad A. Kulkarni, University of Kansas, 1450 Jayhawk Blvd, Lawrence, KS, 66046, USA, prasadk@ku.edu; Michael R. Jantz, University of Tennessee, Knoxville, TN, 37996, USA, mrjantz@utk.edu.

1539-9087/2019/10-ART?
https://doi.org/0000001.0000001

languages commonly employ profiling assisted algorithms to achieve high program performance. Knowledge of program behavior derived through profiling can empower the dynamic VM algorithms to maximize their effectiveness while minimizing run-time costs [3, 17]. Incidentally, the advanced execution environment supported by VMs provides the ideal flexibility and power for the collection and application of program profiling during dynamic or just-in-time (JIT) compilation and optimization.

In spite of their widespread adoption, important properties of PGOs used in current managed language VMs remain unexplored. While they are typically believed to benefit program performance, the actual benefit of profile data to PGOs is seldom quantified. Likewise, we also need to better understand the impact of the *amount* and *accuracy* of profile data on the effectiveness of dependent PGOs. The effect of cross-input profile data that may be available to *offline* profiling techniques on PGO efficiency is also not entirely clear. Lastly, which profile-guided compiler optimizations are most dependent on profile data as well as responsible for program performance improvement is unknown and needs further exploration. The goal of this work is to carefully and systematically investigate these properties of PGOs for a state-of-the-art managed language VM through rigorous and novel VM implementation mechanisms and experimental strategies.

Better understanding the properties and benefits of PGOs is especially critical now with the emergence of the ahead-of-time (AOT) compilation model in mainstream VM systems [32]. AOT or load-time compilation is conducted offline and occurs only once when the program is first compiled or installed on the device. This model eliminates the time and energy overhead of JIT compilation during each program execution, along with the need to support the profiling, code cache and related runtime infrastructure. However, the AOT compilation model does not have access to the execution specific profile data that is used by PGOs during JIT compilation to customize the native code for individual inputs and enable the application of additional aggressive and potentially unsafe optimizations *speculatively*. Unlike most AOT systems, VMs can provide support for the speculatively compiled code to be de-compiled if the speculative condition is invalidated later. Appreciating the trade-offs of the AOT and JIT compilation models needs a clearer understanding of the benefits that compiler optimizations derive from the availability of more accurate, fresh and customized profile data.

In this work, we develop a variety of innovative experiments and VM frameworks to answer the following questions:

(1) How much does customized (from the same program run) profile data impact the code quality generated by JIT compilers?
(2) How does the amount of profile data impact the effectiveness of PGOs?
(3) How do inaccuracies in profile data affect the quality of generated code? To quantify the impact of inaccurate profile data, we develop techniques to collect and apply profile data from different inputs for the same program as well as to systematically introduce noise into the profile data.
(4) How does profile data impact the benefit and effectiveness of individual optimizations? To answer this issue, we extend the HotSpot Java VM [30] to better isolate individual JIT optimizations and provide interfaces to control their application at run-time.

We believe that our research provides greater insight in the workings, characteristics, and benefits of existing profiling based VM optimization systems, and demonstrates some of the challenges that AOT compilation systems must overcome to achieve comparable code quality to JIT based VMs. All our experiments in this work use OpenJDK's HotSpot Java VM and programs from the DaCapo, ScalaBench, and SPECjvm2008 benchmark suites. HotSpot is a production quality JVM and is nearly synonymous with the JVM on the x86. We believe that other production JVMs in use

on x86 employ similar optimizations to those used by HotSpot. Therefore, we expect HotSpot's behavior to likely be representative of other JVMs, and the results we present to be portable to other JVMs on x86.

The remainder of the paper is structured as follows. We explore background material related to JIT compilers and PGOs as well as related works in Section 2. We present the tools, benchmarks, and hardware used in our experiments in Section 3. We describe the modifications and extensions we made to the HotSpot JVM to facilitate our experiments in Section 4. We describe the experiments we perform, and present and discuss their results in Section 5. Finally, we suggest possible avenues for future work and present our final conclusions in Sections 6 and 7, respectively.

2 BACKGROUND AND RELATED WORK

In this section we describe some applications of profiling to individual optimization problems. We also present prior work investigating properties of profiling and PGOs, and compare the goals of our current research with related past studies.

Profiling data can be collected using *offline* and *online* schemes. Offline profiling uses additional prior runs of the program to generate profile data. A later compilation can then use this profile to guide code optimization decisions. Offline profiling is used by static compilers like GNU gcc/g++ [11, 21, 27, 31]. Dynamic or online profiling collects profile information during the same program run, and is commonly employed by advanced managed language run-times, like those for Java [4, 13, 30, 38]. Researchers have also developed static analysis techniques to estimate some run-time information for PGOs [41]. While JIT compilers typically use online profiling, AOT compilers may employ offline profiling data or static analysis to guide adaptive optimization decisions. Some of our studies in this work assess the impact of imprecise profile-based guidance on the quality of code generated by PGOs.

Profile data has traditionally been employed to find the *hot* or frequently executed program blocks or functions. Knowledge of hot program regions can then be used to focus compilation and optimization effort. For example, many Java VMs only compile and apply PGOs to the hot program methods to minimize JIT compilation overhead at run-time, in a technique called selective compilation [3, 17, 24, 30]. Profile information is also used to direct many other optimization tasks. For instance, profile data was used to randomize/diversify cold code blocks to reduce overhead [18], during profile-guided meta-programming [10], to improve code cache management in JVMs [33], to improve heap data locality in garbage collected runtimes [20], to guide object placement in partitioned hot/cold heaps to lower memory energy consumption [22], etc. Our goal in this work is not to generate new or improve existing PGOs, but to determine how inaccuracy in profile data or static analysis based estimators can impact the effectiveness of PGOs.

Several prior studies compare the accuracy and impact of sampling-based profilers on adaptive tasks. The accuracy of any given profile data can be compared directly with the known correct profile, if it is available [5, 14, 28]. When the correct profile itself either cannot be generated or is not known, researchers have used causality analysis to assess if their profile is able to correctly guide the dependent adaptive task [29, 34]. Rather than evaluate the accuracy of the profiler, part of this work assesses how profiles derived from different plausible program inputs can represent the program execution for the current run. To our knowledge, this work is the first to conduct a thorough systematic quantification of representative-ness of different profile data and the effect of such dissimilarity on the effectiveness of PGOs in a standard Java VM.

Previous studies have explored static and AOT compilation of Java to benefit short-running programs (startup performance) due to reduced JIT compilation overhead [19, 35, 40]. Instead, in this work we study the effect on generated code quality (i.e., steady-state performance) that is important to longer-running programs.

Several recent works have developed novel strategies for transparent and low-overhead PGO deployment for embedded and warehouse-scale applications [12, 23]. Observations from our work may help construct and improve future PGO deployment systems.

3 TOOLS, BENCHMARKS, AND EXPERIMENTAL SETUP

In this section we provide a brief background on the properties of the HotSpot VM and the benchmarks used that are relevant to this work. We also explain some details of our experimental setup.

HotSpot Internals: All our work for this paper was conducted using Oracle's production-grade Java virtual machine (HotSpot) in JDK-9 [30]. HotSpot provides the reference JVM implementation and includes high-quality and state-of-the-art JIT compilers. HotSpot's emulation engine includes a high-performance threaded bytecode interpreter and two distinct JIT compilers. The *client* or *c1* JIT compiler is designed for fast program *startup*. The *c1* compiler is very fast, but applies fewer and simpler compiler optimizations. The *server* or *c2* JIT compiler is slower and applies a broad range of traditional and profile-guided optimizations to generate higher-quality code for fast *steady-state* program performance. In this research we focus on *code quality* and therefore only use the *c2* compiler for all our experiments.

Program execution in HotSpot begins in the interpreter. The HotSpot interpreter profiles program execution to collect various program behavior statistics, including the *invocation* and loop *back-edge* counts for all program methods. If the sum of the invocation and loop-backedge counts for a method exceeds a fixed threshold, then HotSpot queues that method to be compiled.

Background Compilation: HotSpot employs a technique called *background compilation*, where JIT compilation occurs in separate OS *threads* in parallel with application execution [24]. Background compilation prevents application stalls due to JIT compilation. However, it can also delay method compilation (relative to the application threads) if the compilation queue is backed up; during which time the method running in the interpreter can continue collecting profile data. Therefore, we disable background compilation for most of our experiments to allow more determinism and control over when each method is compiled and the amount of profile data collected prior to compilation.

Method Deoptimization: A JVM may need to occasionally invalidate and *deoptimize* a compiled method. Deoptimizations are typically caused if a condition assumed or present during JIT compilation is invalidated by a later execution event. Deoptimized methods are interpreted on future invocations, until they become hot again and recompiled. Thus, frequent method deoptimizations can influence the program's execution time. In this study we verify that our experiments do not cause abnormal or performance-affecting deoptimization activity. Likewise, to achieve a fair comparison, all the experimental configurations in this work allow deoptimized methods to be recompiled later if they regain hotness.

Benchmark Suites: Our experiments use benchmarks from the DaCapo [9], SPECjvm2008 [37], and ScalaBench [36] suites. Five DaCapo benchmarks, `batik`, `eclipse`, `tomcat`, `tradebeans` and `tradesoap` are excluded because they fail to run with the unmodified HotSpot-9.[1] Similarly, we exclude the `actors` benchmark from ScalaBench because it crashes when run on stock JDK9. We also leave out SPEC's compiler benchmarks (*compiler* and *sunflow*) due to incompatibilities with HotSpot-9. Finally, other than *monte_carlo*, the remaining programs in SPEC's numerical *scimark*

[1] `batik` and `eclipse` fail due to incompatibilities that were introduced in OpenJDK 8 and have been observed and reported by others [1, 2]. `tradebeans` and `tradesoap` witness frequent, but inconsistent failures with the default configuration. We have not fully investigated the cause of the failures, but we believe it is related to issues reported in [8].

benchmark (*lu*, *sor*, and *sparse*) fail to derive any benefit from PGOs in HotSpot. Therefore, we exclude these programs from our later discussion to improve graph presentation for the more interesting benchmarks. Unless specified otherwise, the DaCapo and ScalaBench programs are run with their *default* input, and the SPEC benchmarks use their *startup* input configuration.

Our experiments attempt to evaluate the quality of code generated by PGOs during JIT compilation by measuring program execution time after all desired compilations are complete. We exploit a mechanism provided by the DaCapo, ScalaBench and SPEC harness that allows a benchmark to be *iterated* multiple times. To achieve determinism most of our experiments restrict the set of methods compiled to those that are detected to be hot and are compiled in the first program iteration. Each run iterates the benchmark 12 times and measures the program run-time during its final iteration.

Table 1 describes some characteristics of the benchmark used in this work. The first column in Table 1 gives the benchmark name. The next column reports the average steady-state program run-time with the default HotSpot setup. The final three columns provide the number of methods compiled by each benchmark during its first iteration (startup), at the end of 12 iterations (steady-state), and by a compiler that compiles all program methods on their first invocation respectively. To account for inherent timing variations during the benchmark runs, all the run-time results in this paper report the (geometric) average and 95% confidence intervals over 10 runs for each benchmark-configuration pair [15].

Our experiments were conducted on a cluster of identically configured Intel x86-64 2.4GHz machines running the Fedora Linux OS. To further minimize the possibility of hardware effects influencing our observations, for each configuration, we execute the benchmark on the same set of 'N' machines (N equals 10, the number of runs), with 'run_i' performed on machine 'i' ($0<i<$N).

4 CONSTRUCTED EXPERIMENTAL FRAMEWORKS

We implement many new mechanisms in the HotSpot VM to correctly and fairly conduct our experiments for this study[2]. In this section we describe these engineered frameworks.

4.1 Detect User-Defined Program Execution Points

Ordinarily, the VM does not possess the ability to efficiently detect user-defined program points as they are reached during execution. We found that many of our experiments would benefit from such a VM capability, especially to detect the start/end of individual benchmark *iterations*. Inspired by prior work [25], we add an empty *VM-indicator* method to the harness of each benchmark suite that starts the next program iteration and statically annotate the method with a special flag. We extend the VM to mark such annotated methods when the classfile is loaded. The HotSpot interpreter efficiently checks for this flag at every method invocation and directs VM control-flow to custom user-defined code if it is encountered during execution.

4.2 Import/Export Profile Data

One important contribution of this work is a mechanism that we built in the HotSpot JVM for exporting profiling data recorded during one instance of the VM and importing it during a later instance. Static compilers that support PGOs, like GCC (*gprof* [16]) and LLVM (*llvm-profdata*), possess the ability to collect and dump profile data from one program execution, and use it during a later compilation to guide PGOs. However, such frameworks are uncommon for managed language run-times, such as Java VMs, since they typically rely on online profiling.

[2]The source code for the modified version of HotSpot used in our experiments may be found at https://github.com/aprilwade/tecs20-sources

Benchmark	Steady-State run-time (ms)	Methods compiled		
		Startup	Steady	All
DaCapo benchmark suite (*default* input)				
avrora	5916.00	396	487	5125
fop	488.90	656	1277	8474
h2	5541.20	772	842	6435
jython	2699.10	1309	1419	8519
luindex	843.10	295	483	5052
lusearch	2013.20	352	384	4376
pmd	1616.00	1085	1642	7296
sunflow	2045.80	350	373	6018
xalan	936.30	828	1242	6153
SPEC JVM 2008 benchmark suite (*startup* configuration)				
compress	1507.00	98	103	3296
crypto.aes	3803.90	109	116	4042
crypto.rsa	361.90	183	304	4104
crypto.signverify	755.40	161	215	3933
derby	998.50	845	886	8950
mpegaudio	2637.90	148	149	3481
scimark.monte_carlo	1452.20	77	76	3264
serial	4550.20	313	412	4194
sunflow	1352.50	319	348	4946
xml.validation	689.40	518	784	5845
ScalaBench benchmark suite (*default* configuration)				
apparat	13400.00	1399	1979	8756
factorie	33195.10	733	758	5427
kiama	879.00	670	1093	7233
scalac	2453.20	2356	4428	33443
scaladoc	2159.80	1832	3186	15145
scalap	206.00	493	831	6407
scalariform	714.20	898	1454	8096
scalatest	1342.30	1045	1621	31543
scalaxb	693.80	681	1289	7743
specs	1761.50	665	1333	25851
tmt	9593.70	823	1027	6644

Table 1. Relevant benchmarks properties

For many data types, including counter and boolean values, the serialization/deserialization process is relatively straightforward. However, there are exceptions like the pointers to the VM structures that represent JVM classes. Since pointer values are specific to each execution instance, we abstract such data types by recording the corresponding class name (including package path), in the serialized format. Later during deserialization, we perform a lookup to find a loaded class structure with a matching name.

Looking up a class name requires that class to have previously been loaded by the VM. The design of the class-loading infrastructure in HotSpot prevents us from loading classes during the deserialization process. Therefore, we delay the deserialization process until all referenced class names in the imported profile file have already been loaded. In order to achieve a reasonable lookup-hit rate, our framework prevents methods from being compiled during the first iteration of

the benchmark and performs the deserialization of the profiling data in between the first and second benchmark iterations. Even with this mechanism, there are a few lookup misses. We analyzed some of these misses and found that many of them come from what appear to be dynamically generated classes with semi-random names. Since there are only a few such cases, we do not attempt to resolve the class name in such cases.

Another challenge is serializing profile data structures that vary in layout depending on the bytecodes that make up the method. Specifically, for each method, HotSpot maintains an array of structures that hold the profiling information for particular bytecodes in the method. For example, a virtual call bytecode corresponds to a structure that records the receiver types seen at call site. Resolving such challenges required precise and meticulous implementation.

4.3 Control Method Compilation Order

The order in which methods are compiled in the HotSpot VM is known to influence later optimization decisions, especially for method inlining. Configurations that compile an identical set of methods in different orders can generate different compiled native codes and result in different program run-times. Therefore, we built a mechanism in the VM to sort and compile the set of hot methods in an external user-defined order. However, a naïve implementation of such a mechanism may delay the compilation of some hot methods if any other methods that precede it in the sorted order have not yet been compiled. This delay in compilation is problematic for our study since the delayed methods will continue to collect additional profile data, which can affect optimization decisions.

Our mechanism to resolve this issue conducts the experiment in two runs for each benchmark configuration. The first *training* run uses the framework just described to export the profile data for each hot method at the proper point during execution. In the second *evaluation* run, the first benchmark iteration is completely interpreted and conducts no JIT compilations. The VM uses the VM-indicator mechanism to detect the end of the first iteration. At this point, the VM stalls the application threads, loads the profile data exported by the training run, and then sorts and compiles the set of hot methods in the given order. The application threads are resumed after all compilation is done.

4.4 Similarity or Representativeness of Program Inputs

Some of our studies employ a new mechanism that we built to quantify the similarity of any two program profiles with respect to the profiling decisions they induce during PGOs. Intuitively, our similarity metric determines the percentage overlap in the program path induced during method compiles by the two profiles being compared. Our metric is analogous in intent to the *overlap* metric used in past works to evaluate profiling accuracy [6].

The representative-ness or similarity of two collected profile data instances is a factor of the dependent PGO. We identified 64 *profile-site* locations in HotSpot's c2 compiler where profiling data is used to inform optimization decisions. We insert hooks at all these locations. When a method is compiled, we record the locations visited and their order. At each hook, we note the name of the current method being compiled (which disambiguates whether this is an inlined method), the current bytecode-index (BCI), and the unique number of the hook location. The record of these profile-site decisions creates a trace of the path the compiler takes as it makes profiling-informed decisions.

Our technique for measuring the similarity of two traces for a given method is inspired by the Unix `diff` utility. Our mechanism calculates the longest-common-subsequence (LCS) of the two traces and divides two times the length of the LCS by the sum of the lengths of the two individual traces. The resulting ratio gives us a percentage measure of similarity. When calculating

the LCS, we treat the tuple of the three recorded data values at each profile-site as an atomic unit, analogous to how the `diff` utility treats individual lines as atomic units when calculating a LCS. To create a measure of similarity for the entire program, we compute an (unweighted) average of the representative-ness measure of every method that was compiled during both VM instances.

5 EXPERIMENTS, RESULTS AND ANALYSIS

In this section we describe the results of our experiments that investigate the characteristics of current profiling-based JIT optimization systems in VMs.

5.1 Impact of Profiling on Generated Code Quality

Our first set of experiments are designed to evaluate and quantify how PGOs in current JIT compilers are able to utilize profiling data to improve generated code quality and performance. We prepare five distinct HotSpot configurations to compare the behavior and performance of JIT and AOT compilation systems. Configurations designed to simulate AOT compilation systems are denied access to profile data.

AOT-all: This configuration compiles all program methods on their first invocation. We disable profile data collection. Compilation occurs at the end of the first benchmark iteration. A method compilation order cannot be enforced as we do not have any other baseline configuration. The last column in Table 1 gives the number of methods compiled by each benchmark in this configuration.

JIT-steady: HotSpot employs selective compilation to only compile methods when they are detected to be hot (invocation+loop-backedge counts exceed 10,000 in HotSpot). This configuration represents the *steady-state* setting. Profiling is enabled. A method compilation order is not enforced and the methods are compiled as they achieve hotness in their first twelve iterations. The number of methods compiled by this configuration for each benchmark is given by the fourth column in Table 1.

AOT-steady: This configuration restricts the set of methods compiled to those that are compiled by the *JIT-steady* configuration for each benchmark. We do not enable profiling for this AOT compilation. All methods are compiled after the first program iteration, and a method compilation ordering is not enforced.

JIT-startup: This configuration is similar to earlier *JIT* setup, but restricts the number of methods compiled to those that get *hot* during the first iteration with HotSpot's default setting. The number of methods compiled by each benchmark is given by the third column in Table 1.

AOT-startup: The configuration is similar to *AOT-steady*, but restricts the set of methods compiled to that compiled during *JIT-startup*. Profiling is disabled, and methods are compiled in the order they reach compilation in the *JIT-startup* configuration as described in Section 4.3.

In all cases the run-time of the 12^{th} benchmark iteration is reported to ignore compilation overhead and allow the execution to stabilize.

Figure 1 compares program performance with the AOT and JIT compilation models. The first bar for each benchmark in Figure 1 plots the ratio of the *AOT-all* and *JIT-steady* configurations, the second bar compares the *AOT-steady* and *JIT-steady* configurations, while the last bar compares the *AOT-startup* and *JIT-startup* configurations. The first comparison gives an estimate of the profiling benefit derived by HotSpot-like VMs that employ selective compilation and may only compile a fraction of the program methods. The final two plots for each benchmark can be used to estimate the performance gain due to profiling for VMs and benchmarks that have sufficient time and resources to compile all program methods. By enforcing a common method compilation order,

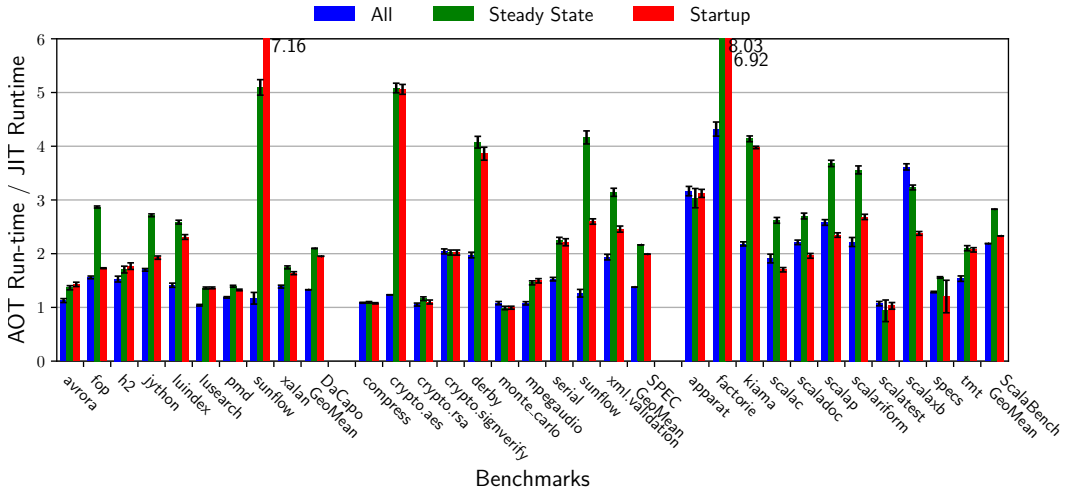

Fig. 1. Profile data and PGOs have a significant impact on program performance on the HotSpot JVM

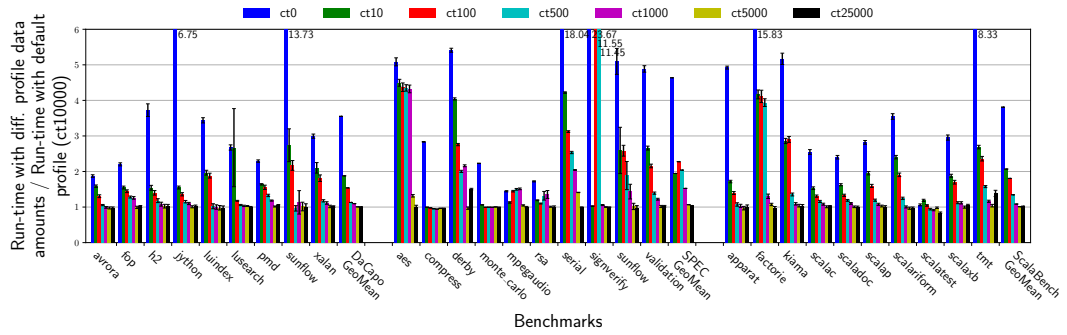

Fig. 2. A small amount of profile data from the current program run is sufficient to effectively guide PGOs on the HotSpot JVM

the *startup* configurations eliminate one additional source of performance unpredictability, and therefore provide a better baseline for comparison. We use these *startup* configurations in our later experiments.

All comparisons uniformly show that PGOs in current VMs for languages like Java are able to employ the program profile behavior to significantly improve the quality of generated code. For instance, the *JIT-startup* configuration is able to improve performance over AOT-startup by 1.95X, 1.99X, 2.33X, on average, for the DaCapo, SPECjvm, and ScalaBench suites respectively.

While they cannot collect or exploit profiling data from the ongoing execution, AOT compilers may have access to mechanisms like offline profiling or static analysis to address this potential loss in performance. Our experiments in later sections help understand the challenges that AOT compilers may need to overcome when using these alternative mechanisms to drive PGOs.

5.2 Impact of Profile Data *Amount* on Code Quality

JIT compilation systems employ online profiling. These systems need to balance the amount of profile data collection with the delay in making optimized code available to the emulation engine.

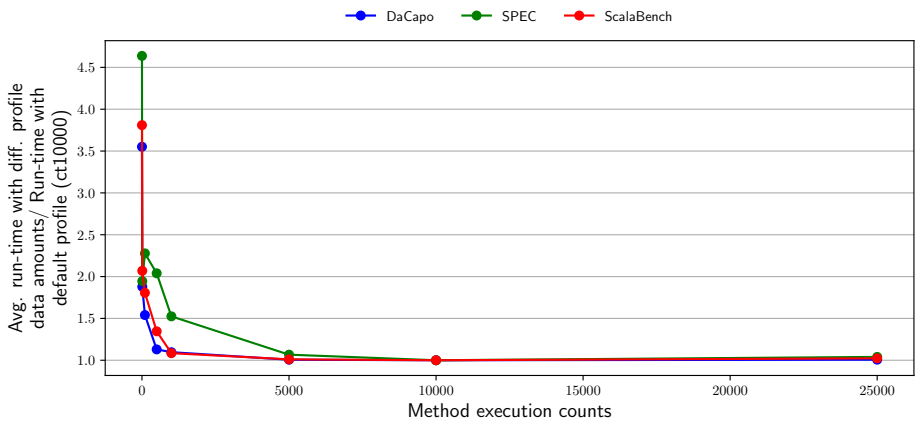

Fig. 3. Average program performance quickly improves and reaches saturation with small increases to the amount of collected program profile information.

Spending too little time profiling the program behavior may have performance implications by incorrectly biasing adaptive optimization decisions. Likewise, staying too long in the profile stage will delay JIT compilation, causing the program execution to remain in the inefficient interpreter for a longer duration. In this section we investigate the issue of how much profile data is needed by current PGOs to make correct profile-based decisions and generate the best quality code.

We design a simple experiment that precisely controls the amount of profile data collected during the multiple different training runs. This experiment employs our frameworks described in Sections 4.2 and 4.3. Thus, the training runs export the collected profile data that is then loaded and used by the evaluation run. We also control the number of methods compiled and their compilation order so that these factors remain uniform across all experimental configurations. We configure the training runs to collect per-method profile information that corresponds to each method executing for 0, 10, 25, 50, 75, 100, 250, 500, 1000, 2500, 5000, 10000, 25000, and 50000 execution (invocation + loop backedge) counts. By default, HotSpot uses the compile threshold of 10000 for its c2 compiler.

Figures 2 and 3 show the results of this experiment. We observe slightly different trends for DaCapo and ScalaBench benchmarks compared to the SPECjvm benchmarks. Overall, one surprising finding is that just a little profile knowledge (like that provided by *ct10*) can substantially benefit performance over no-profiling. Less surprising is the result that performance obtained from increasing profile knowledge quickly reaches saturation. We see only small performance gains with profile data from execution counts beyond 1000 with DaCapo and ScalaBench, and 5000 with SPEC. These results also suggest that offline profiling conducted over long time intervals (that is an option for AOT-based systems) may not have much of an advantage over traditional online profiling based JIT compilation systems.

We also find that, unlike the DaCapo and ScalaBench programs, performance for many SPEC benchmarks does not always improve with increasing profile data, especially at low compile thresholds (see ct10 vs. ct100 for *signverify*). We find that this issue is caused because SPEC benchmarks generally compile fewer methods and have fewer critical hotspots. Therefore, small variance in profile data and resulting optimization decisions cause an outsized impact on final program run-time.

5.3 Impact of Profile Data *Accuracy* on Code Quality

Offline profiling uses profile data collected from previous *training* runs to bias profile-guided optimizations and tasks during the current *production* run. Offline profiling mechanisms have been studied in JIT compilation based VMs to improve program startup performance [7, 26]. AOT compilation systems that cannot customize the single statically generated binary to all program inputs also have the option to employ offline profiling to guide PGOs. In this section we report our observations from experiments conducted to understand two important issues. First, how similar does the guidance provided to the PGOs by the training and evaluation inputs need to be to generate comparable quality code; and second, how much do non-representative program inputs affect quality of guidance provided to PGOs and what is the resulting performance impact.

5.3.1 Offline Profiling with Other Inputs. The DaCapo suite provides two to four distinct input settings for each benchmark program. A *small* and *default* input are provided for every benchmark and most have *large* and *huge* inputs as well. To this set, we have added inputs for each benchmark to ensure that each benchmark has at least 7 distinct inputs available. The `jython` benchmark is an exception. We had difficulty locating programs that were compatible with the version of `jython` that is distributed with DaCapo. As such `jython` only has the 3 inputs supplied by DaCapo available.

The benchmarks can be divided into two categories based on the nature of their inputs. The `h2`, `sunflow`, and `xalan` benchmarks have inputs that are only varied quantitatively. That is, the inputs only differ in terms of one or two numbers that determine the size of the workload. Thus, when creating inputs for these benchmarks, we merely generated inputs with different values from the provided benchmarks. In contrast, the inputs to the `avrora`, `fop`, `jython`, `luindex`, `lusearch`, and `pmd` benchmarks vary in a more qualitative way. For example, the inputs to `luindex` are corpuses of text, some of which overlap with one another. For these benchmarks, we sought out new inputs. For `luindex`, we used the text of public domain novels and the text of randomly selected Wikipedia articles as our sources for new inputs.

In this section, we evaluate the effectiveness of recording the program behavior with one of the aforementioned inputs, and then using that *offline* profile data to guide PGOs during an evaluation run with the *default* input set. For each benchmark-input pair, one training run and one evaluation run is performed. The training run uses one benchmark-input pair and collects profiling data and method compilation order. The evaluation run executes the benchmark using the *default* input and the profile data collected during the training run. We use the setup described in Sections 4.2 and 4.3 for these experiments. At the end of the evaluation run's first iteration, the VM stalls all application threads, imports the stored profile data and compiles all the hot methods in the compilation order provided by the training run. The program run-time is recorded at the end of 12 benchmark iterations.

Figure 4 displays the run-time reported by the offline-profiling configuration as compared to the run-time from the first configuration for each benchmark. We find that, with a few exceptions (most notably `input1` of `pmd` and `input9` of `xalan`), the use of profiling data collected from different inputs provides performance within 6% of using profiling data from the *default* input. Surprisingly, in several cases, for example `input5` of `h2`, using the training data from another input provides a significant performance improvement compared to using the data collected from the same input as the one being executed. Our results suggest that while offline profiles from different inputs result in equivalent program performance in most cases, pathological cases that see a significant performance impact can exist.

In Section 4.4 we described our technique to compare and quantify the similarity or overlap in the paths taken through the HotSpot JIT optimizer for a given method/program by two different

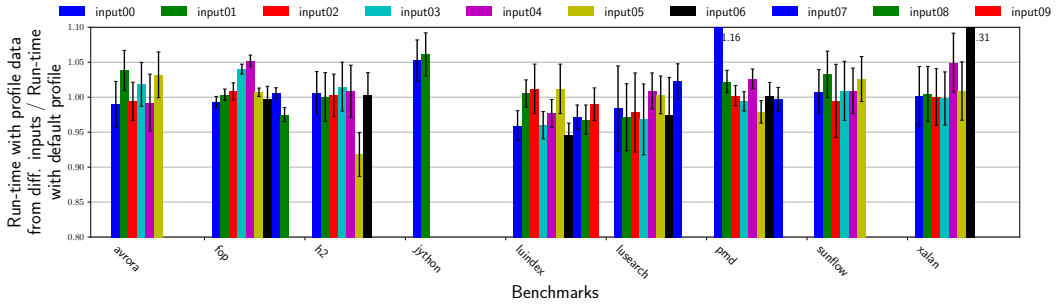

Fig. 4. In most cases, offline profiling using another input produces a binary that achieves good performance with a later evaluation run with the *default* input.

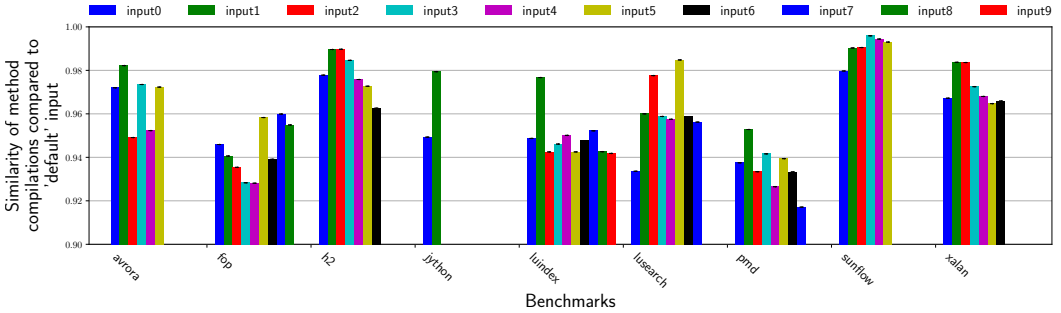

Fig. 5. The compiler behavior of DaCapo's *default* can be closely represented by loading profiling information collected using other input sets.

program inputs. Figure 5 uses this mechanism to quantify the representative-ness of each input compared to the *default* input for each benchmark. We find that with an average similarity score of more than 90%, program behavior with inputs other than the *default* input are still quite representative of its behavior with the *default* input itself, with regards to guiding the PGOs in the HotSpot JVM.

At the same time, while our profile data similarity metric is useful (as will be reconfirmed in the next section and Figure 6), correlation between the similarity metric and program performance during cross-input runs (Figures 4 and 5) is poor. Our similarity metric compares decisions at hundreds of program sites and is thus far unable to weigh individual profile data based decisions by their resulting impact on execution performance. We plan to explore more accurate similarity metrics in future work.

5.3.2 Offline Profiling with Randomized *Program Input.* Although the input sets provided by DaCapo and the ones we have collected generate representative profiles for the *default* input set for most benchmarks, it is unclear (a) if other program inputs may provide varying representative-ness, and (b) what is the effect of such plausible variance on the effectiveness of PGOs and delivered code quality. Unfortunately, we do not know of any Java benchmark suite that includes a deliberately and systematically designed diverse set of program inputs. It was also not obvious to us how to generate such diverse input sets for our set of benchmarks. Instead we develop a novel approach to systematically vary the representative-ness of the known program profile for any program-input pair, and study its effect on performance.

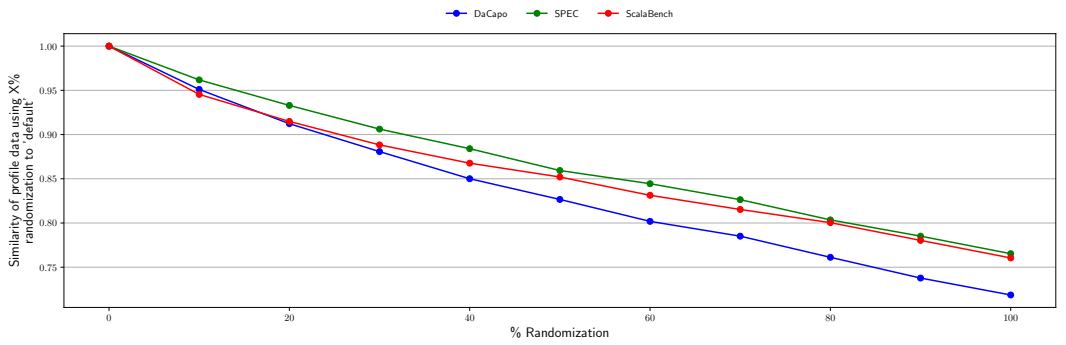

Fig. 6. Average representative-ness of the profile trace for various randomization configurations as compared to HotSpot's default *reactive* configuration

Our approach first conducts a training run to collect and export the complete per-method profile data for each benchmark with its standard *default* input set. This profile contains multiple fields, such as branch-taken counts, trap information, etc. Then, we methodically introduce random noise into this profile data with controlled probabilities as each profile data field is loaded during later evaluation runs. We call this process *randomization* of the profile data. Thus, a profile data randomization with a probability of X% will alter a profile data field with a probability of X% and leave it unchanged with a probability of (100-X)%.

Randomization of the profile data field depends on the type of the field. For boolean fields, randomization flips the boolean value. For integer counter fields, randomization will set the field to a `low` or `high` value with the same probability. A `low` counter value is guaranteed to be less than the fixed VM threshold for that counter, and a `high` counter value exceeds the threshold. For class pointer fields, a non-`null` field will be set to `null` with the same probability. If the randomization does not nullify the entire field, then each referenced class in that field may again be set to `null` with the same probability. We do not yet attempt to alter a class pointer to instead reference another random class. Likewise, we also do not attempt to update a `null` class pointer to reference some other random program class. This randomized profile data will be used later during the run by the VM to guide PGOs during JIT compilation of the hot program methods.

Our experiment employs randomization values in increments of 10, from 0% to 100%. We employ our mechanism described in Section 4.4 to calculate the similarity metric of each randomized profile data. Figure 6 shows the average representative-ness metric over all benchmarks for all the randomization ratios attempted. We see that profile data similarity decreases with increasing randomization, and validates that our randomization technique is working as intended to alter the representative-ness of profile data.

This curve shows that even small profile data imperfections noticeably affect the similarity metric. Yet, even a completely random (100% randomization) program input still achieves a reasonably high similarity metric (72% for DaCapo, 77% for SPEC, and 76% for ScalaBench), indicating that even vastly different profiles result in the compiler following a similar optimization path in a majority of the cases.

Figures 7 and 8 show the performance implications of using varying levels of imperfect profile data to guide PGOs in HotSpot's c2 compiler. For each benchmark, each bar in Figure 7 plots the ratio of program run-time when the VM is using the indicated randomization of profile data to program run-time in the default scenario when using online profile data from the same run with

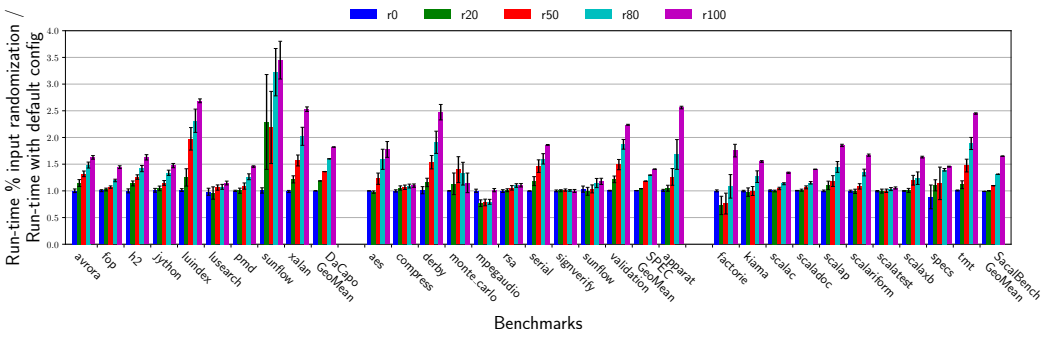

Fig. 7. Impact of varying profile data inaccuracy on *individual* program performance on the HotSpot JVM

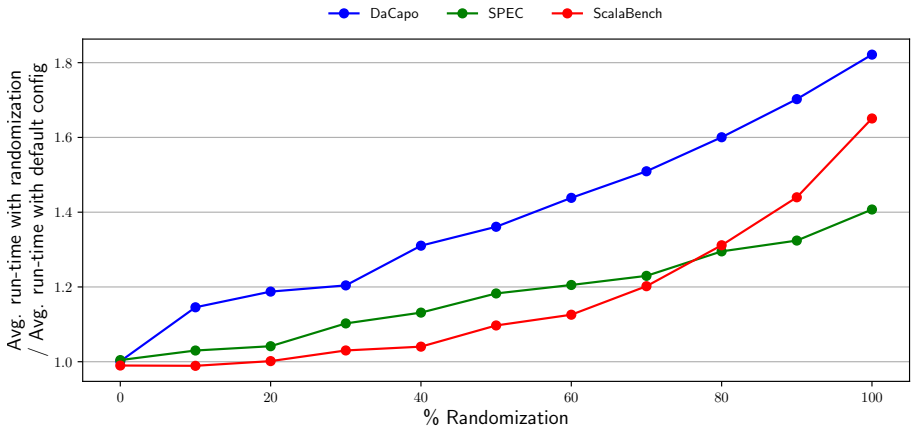

Fig. 8. Impact of varying profile data inaccuracy on *average* program performance on the HotSpot JVM

no randomization. Again, we employ the frameworks described earlier in Sections 4.2 and 4.3 to produce a fair comparison.

All benchmarks show an identical trend with performance degrading with increasing profile data imperfection in most cases. The scale of performance change varies significantly between the programs, and is likely a factor of several concerns, including the benefit derived from profiling and the significance of the sites randomized. We observe that while performance for the DaCapo benchmarks uniformly degrades with increasing randomization, the SPEC and ScalaBench programs notice some jitter. For the SPEC benchmarks, we believe this effect is again a result of the programs themselves having fewer and more prominent hotspots. We hypothesize that the ScalaBench performance behavior occurs due to Scala programs tendency to be written in a functional style that creates highly-nested function calls that often rely on calls made on interface methods, which in the degenerate case can be expensive to execute. Consequentially, the impact of incorrect profiling data increases geometrically, rather than linearly as it does for the DaCapo programs. One important finding is that even small imperfections in profile data can significantly lower the effectiveness of PGOs, which bears serious implications for offline profiling based optimization strategies. Note that this is a limit study; whether actual program inputs can generate such a diverse range of profiles is an open issue.

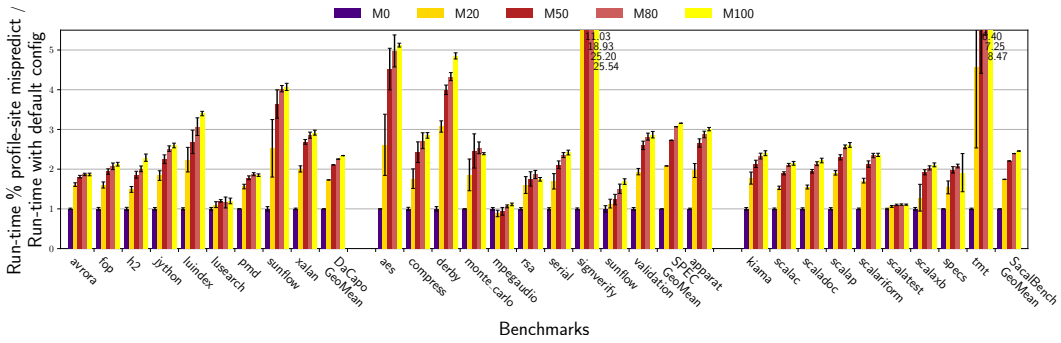

Fig. 9. Increasing the probability of mispredicting at a profile-site branch increases the negative impact on the quality of generated code on the HotSpot JVM (individual benchmark view)

5.3.3 Randomizing Profile-Site Decisions at Compilation. Profile data is used at various *profile-sites* during compilation to affect optimization decisions. As mentioned earlier, we identify 64 static profile-sites in the source code of HotSpot's c2 compiler where some profile data determines the path taken by the compiler. In this section we study and quantify the sensitivity of the compiler to incorrect decisions taken at profile-sites. Again, we define accurate profile decisions as those induced by *online* profiling, where the profile data for the current run is dynamically collected by the VM during the same measured program run. This result is important to static analysis based prediction techniques that may be employed by AOT compilers to guide PGOs.

Our experiment to quantify the performance impact of compiler sensitivity systematically varies the probability of the compiler taking a wrong decision (relative to that taken by the online profiling based *reactive* HotSpot configuration) at a profile-site. Our experiment reverses the path taken at each profile-site with a given user-specified probability (referred to later as the *mispredict* probability). Thus, a mispredict probability of 0% forces the c2 compiler to take the same path as that taken by the *reactive* HotSpot configuration every time and at every profile-site reached during compilation. In contrast, a mispredict probability of 100% forces the c2 compiler to take the *wrong* path at every profile-site, whenever feasible.[3] We found that mispredicting the trap-related profile-sites[4] produces high instability in HotSpot and causes a very high number of deoptimizations. Therefore, we currently always predict correctly for this set of profile-sites.

Figures 9 and 10 show the impact of different mispredict probabilities on program performance as compared to the program run-time achieved by the default reactive HotSpot configuration with 0% mispredict probability. We find that even a small mispredict probability causes a noticeable degradation in generated code quality. A 4% mispredict probability increases program run-time by 15.1% for DaCapo (28.7% for SPEC), while 100% misprediction causes a 2X slowdown for DaCapo (over 3X for SPEC). Thus, our experiments show that the HotSpot c2 compiler relies on correct prediction at most profile-sites to maximize effectiveness. This result sets a high bar for any technique that attempts to correctly predict the direction of individual profile-sites to improve code quality.

[3]It is not always feasible to take the wrong path. For instance, if the profile-site references a profile data type that is a class pointer, and the profile data recorded by the *reactive* configuration is `null`, then taking the reverse path may require us to now provide an actual plausible class pointer value. Our setup does not yet have the capability to construct such values.
[4]Sites that determine whether a trap event, such as an array-out-of-bounds exception, occurs at a particular BCI or in a given method.

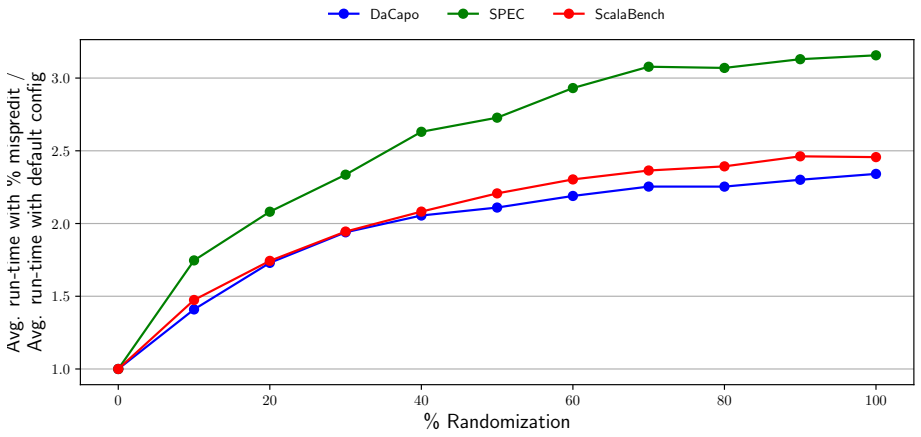

Fig. 10. Impact of varying the probability of mispredicting at a profile-site branch on *average* program performance

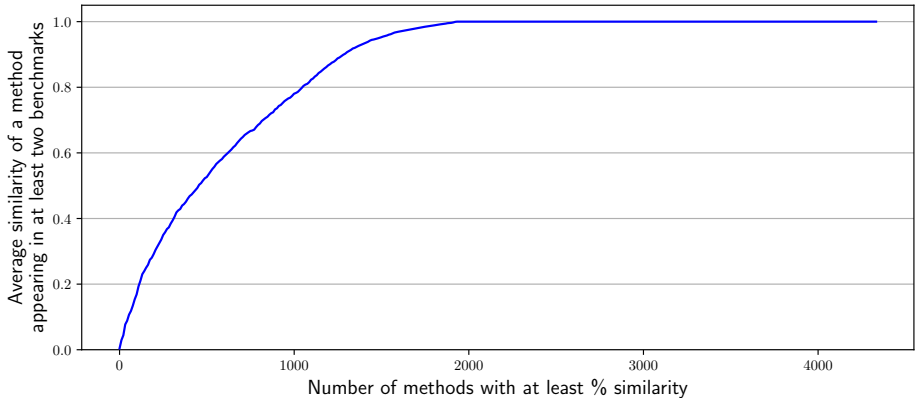

Fig. 11. The majority of all methods are compiled identically when invoked in different benchmarks.

5.3.4 Comparing similarity of method compilations across benchmarks. Several (library) methods are reached and compiled during different benchmark executions. In this section we apply and study the compilation similarity metric (described in Section 4.4) for methods that are compiled across different benchmarks and potentially vastly different contexts. To do so, we collect the traces of compilation decisions for each method compiled in (a single iteration of) the *default* run of every benchmark in all three suites. Then, for every method that was compiled in at least two of the benchmarks, we compute the similarity of how the method was compiled in each pair of benchmarks that compiled it. For methods that were compiled in more than two benchmarks, we average the similarity from each benchmark-pair to produce a single value for each method.

The results from this study are shown in a cumulative frequency graph in Figure 11. We see from this figure that cross-benchmark context seems to matter less to decisions made during PGOs, even when they are invoked from different programs. In fact, 58% of the methods are compiled identically even when being compiled in different applications and there is a 91% average similarity for all compilations of the same method across benchmarks. This result shows that even in what should be highly different workloads, the majority of methods are compiled identically and that

only a small minority of methods have their compilations significantly influenced by contextual differences in profile data. While interesting, we note that the results of this study are limited by the observation that important program methods are less likely to be shared between benchmarks in many cases.

5.4 Impact of Profiling on Individual Optimizations

Most modern compilers, including HotSpot's c2 compiler, employ many different optimization phases to improve the quality of generated code and achieve higher execution performance. Distinct optimizations are likely to contribute differently to the overall performance improvement. Likewise, while many individual optimizations may use profile data to guide their heuristics and operation, the impact or importance of profile data for each optimization is also likely to differ. In this section we describe results from our study that attempts to understand and quantify both the performance impact of each optimization and the impact of profiling data on the effectiveness of *individual* optimizations in a JIT compiler.

For this study we identified 52 flags in JDK9 HotSpot that control or influence optimizations in its c2 compiler. In addition to these 52 flags, we reintroduced 8 other optimization control flags that existed in earlier JDK versions but were removed in JDK9. Disabling all 60 optimization flags lowers average program performance by 2.64X, 4.18X, and 3.03X for benchmarks in the DaCapo, SPEC and ScalaBench suites respectively.

We realize and note the following challenges for this study. First, in spite of our best efforts there may be optimizations that we do not yet control. Such optimizations will be active in all our experiments. Second, disabling an optimization may block some program analysis if the analysis if intricately woven together with the transformation. Preventing the analysis may hurt later optimizations that depend on that information. We have attempted to identify and prevent such cases.

Additionally, *register allocation* is not one of our 60 flags and is active in all our experimental configurations. Since HotSpot did not provide a way to disable register allocation in JDK9, we updated this pass to allow curtailing the number of available registers. We found that with fewer than four available registers, the method stack size grew enough for the c2 compiler to abort compilation for many methods. This issue in addition to the expected loss in efficiency from more memory accesses due to fewer registers resulted in a drastic performance loss for most benchmarks. With register allocation limited to a reduced set of 8, 4, and 2 registers, average benchmark performance reduced by (133%, 147%, 1825%), (125%, 200%, 2877%), and (108%, 120%, 1982%) for DaCapo, SPEC and ScalaBench respectively. We keep register allocation ON in all our experiments.

Experimental Design. Experiments to study the effect of (profiling on) each individual optimization flag (say, 'opt_A') could be designed in one of (at least) two ways:

(1) **EXP-A**: Change the configuration of opt_A (opt_A=ON Vs. opt_A=OFF, or opt_A with profile data Vs. opt_A without profile data), while all other optimization flags are ON and applied uniformly. Several compiler optimizations are known to *enable* one another; this configuration will include the enabling effect other optimizations have on the performance of opt_A. At the same time, several compiler optimizations may target the same program inefficiency (as opt_A). Therefore, this configuration may undervalue the impact of opt_A as other optimizations compensate for the effect of opt_A in certain cases.
(2) **EXP-B**: Change the configuration of opt_A, while all other optimization flags are OFF. This experiment discards the enabling effect that other optimizations have on opt_A, but may better quantify the full potential of opt_A.

The primary goal of our experiments in this section is to understand the effect of profile data on individual optimizations in HotSpot. This study will compare two experiments that both apply each

Opt. Flag Name	Description
UseCHA	Class hierarchy analysis; can devirtualize calls when, for example, no subclasses exist at runtime.
UseTypeProfile	Allows type speculation based on observed type information.
OptoCoalesce	Pessimistic copy coalescing during register allocation.
UseLoopPredicate	Converts array range and loop invariant checks into uncommon traps and moves them outside of the body of their containing loop.
LoopPeeling	Simplifies a loop by breaking it into multiple copies of the original.
ParseGVN	Global value numbering; removes redundant computations
PeepholeRemoveCopies	Removes redundant loads of stack values and register-to-register copies.
ImplicitNullChecks	Allows null pointer dereferences to be caught in a signal handler rather than checked explicitly.

Table 2. Description of important optimization flags

Benchmarks	Optimizations									
	Inlining	others	UseCHA	UseType Profile	Opto Coalesce	UseLoop Predicate	Loop Peeling	ParseGVN	Peephole Remove Copies	Implicit NullChecks
dacapo										
avrora	**2897.30**	**148.30**	51.90	787.40	155.50	-88.40	-85.10	-14.90	74.40	**140.60**
fop	**1119.10**	-7.70	26.80	142.50	11.70	-16.10	-3.10	13.30	15.50	-8.80
h2	**5025.70**	89.90	-135.40	1033.50	-45.00	-16.00	-85.30	118.10	73.00	55.50
jython	**8305.30**	39.50	77.70	838.60	27.40	4.10	-1.40	9.50	141.80	52.30
luindex	**1489.70**	**1.90**	**28.30**	1045.40	**21.40**	0.70	**9.30**	**35.20**	51.70	**11.30**
lusearch	**935.90**	**18.40**	20.70	228.30	**21.30**	**16.40**	**14.10**	**26.40**	**19.40**	**24.80**
pmd	**863.80**	-1.40	52.90	174.80	7.80	1.60	**-7.00**	-2.10	17.20	5.20
sunflow	**25353.30**	**281.00**	**76.80**	2924.10	118.70	**49.70**	**104.90**	**147.00**	**168.30**	**188.00**
xalan	**957.60**	10.90	8.10	128.00	-0.60	-1.00	-3.90	7.80	10.10	**9.20**
MEAN	**5216.41**	**64.53**	23.09	811.40	35.36	-5.44	**-6.39**	37.81	63.49	**53.12**
specjvm										
compress	1523.80	**28.80**	**1.50**	1.90	54.50	40.00	5.10	**140.40**	37.50	**30.30**
crypto.aes	**15380.10**	-26.20	-20.80	1321.10	149.40	-19.00	-28.90	1052.40	50.10	-27.90
crypto.rsa	969.90	-2.00	-0.40	**11.60**	1.80	2.90	-2.10	0.90	1.60	0.90
crypto.signverify	1645.60	-1.00	-2.30	3.00	13.60	30.70	-1.30	1.20	17.60	-0.90
derby	**3291.40**	**10.90**	-18.50	259.60	-13.00	-2.00	-12.40	**55.10**	**43.00**	**28.80**
monte_carlo	**2614.20**	**3.60**	**2.30**	**-1.50**	0.60	**46.70**	**0.10**	2.90	-4.70	**8.90**
mpegaudio	1330.40	**78.10**	11.80	14.20	25.40	156.40	14.40	**543.90**	146.20	**75.10**
serial	8270.60	**123.20**	-13.40	39.50	107.70	-3.70	-9.90	**159.40**	148.60	**80.20**
sunflow	6315.60	**56.70**	-35.60	**1719.50**	-8.50	-18.20	-17.20	56.70	112.30	**63.30**
xml.validation	**1622.20**	**38.20**	6.40	437.50	16.40	2.80	-3.40	54.50	46.40	**40.90**
MEAN	**4296.38**	**31.03**	-6.90	380.64	34.79	23.66	-5.56	206.74	59.86	**29.96**
scalabench										
apparat	**36144.50**	667.70	121.90	16446.60	346.50	172.10	**-177.50**	**640.50**	561.70	449.70
factorie	**423699.50**	1115.40	2997.20	70581.00	85.10	**2759.40**	-15.40	-1012.50	376.60	358.90
kiama	**3536.90**	**2.80**	30.30	1110.90	-3.70	-13.50	-2.00	0.50	3.60	**10.90**
scalac	**5486.50**	61.70	562.90	975.00	20.00	**31.20**	**9.00**	**28.30**	99.10	45.30
scaladoc	**4751.70**	**88.60**	260.70	1086.80	53.80	**21.80**	**8.30**	**66.10**	72.70	**117.70**
scalap	**766.00**	3.90	3.00	209.70	**3.40**	-0.70	**2.30**	**5.50**	4.20	2.10
scalariform	**2761.40**	**20.10**	21.60	816.10	11.00	**1.10**	-0.30	11.60	15.50	**16.80**
scalatest	184.30	**369.70**	**-17.10**	**205.30**	**-25.90**	**-0.10**	**21.50**	**428.70**	**534.90**	**4.30**
scalaxb	**2245.70**	**11.20**	0.80	688.50	-3.50	**47.20**	-3.30	4.30	5.80	**8.40**
specs	**1600.00**	**249.60**	**284.10**	**433.90**	**232.20**	**240.50**	**245.10**	**254.60**	-2.00	-8.40
tmt	**65270.80**	**89.20**	116.80	7575.10	43.40	70.80	56.50	197.50	196.70	**95.20**
MEAN	**49677.03**	243.63	398.38	9102.63	69.30	**302.71**	13.11	**56.83**	169.89	100.08

Table 3. Performance difference (in msec) when individual optimizations flags are turned OFF (FLAG_OFF configuration). Numbers in **bold** indicate performance effect is greater than corresponding effect in the FLAG_ON configuration in Table 4.

focus optimization – one with profile data ON and the other with profile data OFF, while applying all remaining optimizations consistently in both settings. The intricate engineering in HotSpot makes it exceedingly difficult to reliably turn profile data OFF for only one optimization at a time (while leaving it ON for all the other optimizations). Therefore, EXP_A will make it impossible to

Benchmarks	Optimizations									
	Inlining	others	UseCHA	UseType Profile	Opto Coalesce	UseLoop Predicate	Loop Peeling	ParseGVN	Peephole Remove Copies	Implicit NullChecks
dacapo										
avrora	1604.60	78.10	**423.50**	**1660.00**	**248.60**	**-61.10**	**-39.00**	**8.30**	**140.70**	26.70
fop	846.00	**15.30**	**125.00**	**242.90**	**43.00**	**8.20**	**14.10**	**17.90**	**57.40**	**13.50**
h2	4012.50	**510.40**	**322.90**	**1941.90**	**892.40**	**230.20**	**-6.30**	**279.50**	**1055.20**	**468.40**
jython	7239.10	**95.10**	**552.50**	**1641.40**	**220.10**	**18.30**	**1.80**	**86.10**	**331.10**	**84.00**
luindex	544.80	-39.90	25.60	**2805.60**	1.30	**59.70**	-79.00	-21.20	**81.40**	-40.20
lusearch	841.20	-5.00	**29.50**	**266.30**	13.60	-13.40	-7.20	10.50	5.80	-7.80
pmd	602.40	**19.00**	**91.70**	**198.00**	**33.50**	**34.80**	-25.50	**19.70**	**53.60**	**13.80**
sunflow	22187.70	-150.40	-138.40	**3087.60**	**263.90**	-162.20	-86.30	-49.10	-45.80	-161.70
xalan	753.30	**18.60**	**79.50**	**201.10**	**37.40**	**16.50**	**-1.20**	**30.20**	**18.00**	3.10
MEAN	4292.40	60.13	**167.98**	**1338.31**	**194.87**	**14.56**	-25.40	**42.43**	**188.60**	44.42
specjvm										
compress	**1690.70**	15.80	-13.70	**21.00**	**281.10**	**229.40**	**220.80**	91.10	**259.20**	11.50
crypto.aes	13607.30	**27.70**	**10.30**	**2222.10**	**1640.90**	**584.90**	**1415.60**	**1679.80**	**973.60**	**34.30**
crypto.rsa	**1994.10**	**1.50**	**0.90**	8.30	**9.60**	**14.70**	**2.00**	**7.80**	**9.60**	**7.50**
crypto.signverify	**2149.60**	**2.80**	**0.60**	**3.10**	**141.50**	**307.80**	**200.60**	**199.50**	**82.90**	**0.40**
derby	3095.70	-10.50	**197.00**	**470.90**	**93.50**	**41.30**	**26.80**	23.50	40.60	-51.20
monte_carlo	0.00	-30.90	-317.20	-3.10	**1707.10**	-3.40	-249.70	**2750.00**	**1640.40**	-0.50
mpegaudio	**1471.40**	11.60	**12.00**	**56.30**	**340.10**	**480.60**	**156.90**	541.90	**245.70**	13.20
serial	**8287.30**	-17.60	**86.60**	**197.30**	**305.60**	**62.00**	**83.00**	38.20	**416.90**	-3.20
sunflow	**7962.10**	33.80	**12.80**	1502.50	**311.00**	**28.50**	**16.70**	**192.20**	**212.80**	-18.60
xml.validation	1086.50	4.10	**63.70**	**518.20**	**88.00**	**21.00**	**13.20**	**94.20**	**121.50**	14.50
MEAN	4134.47	3.83	**5.30**	**499.66**	**491.84**	**176.68**	**188.59**	**561.82**	**400.32**	0.79
scalabench										
apparat	12996.10	**4841.20**	**522.00**	**24476.60**	**6669.60**	**1286.00**	-517.00	313.80	**4890.40**	**4765.40**
factorie	188045.50	**5039.90**	**64665.00**	**133633.60**	**8234.70**	1449.50	**4916.40**	**-359.60**	**10122.20**	**2914.10**
kiama	2035.40	-8.80	**227.30**	**1336.60**	**50.10**	**20.90**	**19.30**	**5.60**	**70.40**	3.90
scalac	3725.60	**118.90**	**1434.70**	**1697.60**	**215.20**	-41.70	-41.50	-6.50	**357.40**	**65.60**
scaladoc	3043.10	87.00	**965.50**	**1606.30**	**204.00**	8.10	-26.80	1.30	**276.30**	60.10
scalap	463.40	**10.90**	**75.90**	**248.20**	0.40	**-0.50**	-0.50	0.80	**5.90**	**13.80**
scalariform	1565.00	15.60	**249.70**	**1071.10**	**68.80**	-1.50	**6.70**	**15.80**	**87.10**	12.80
scalatest	**557.00**	-549.80	-24.50	-190.60	-306.20	-384.60	-271.30	-306.60	-598.20	-270.90
scalaxb	1586.40	-1.10	**76.40**	**782.50**	**165.70**	13.00	**1.40**	**11.30**	**177.60**	7.50
specs	1132.90	14.60	24.10	173.80	17.90	17.80	13.90	23.50	**34.20**	**19.50**
tmt	14765.20	51.50	**905.50**	**10363.60**	**1083.70**	**317.00**	**433.50**	**416.30**	**895.30**	82.60
MEAN	20901.42	**874.54**	**6283.78**	**15927.21**	**1491.26**	244.00	**412.19**	10.52	**1483.51**	**697.67**

Table 4. Performance difference (in msec) when individual optimizations flags are turned ON (FLAG_ON configuration, inlining is ON). Numbers in **bold** indicate performance effect is greater than corresponding effect in the FLAG_OFF configuration in Table 3.

separate the effect of profile data on opt_A from its effect on all other optimizations. Therefore, we employ EXP-B to conduct our evaluations in the remainder of this section.

While we employ the EXP-B design, we also perform experiments to explore the enabling and compensating effect of optimizations in HotSpot in their default setting (with profile data always available). We conduct two experiments for each optimization flag (opt_A): (a) **FLAG_OFF** uses the EXP-A design to compare program performance of experiment with opt_A=OFF with the default configuration that sets all flags ON, and (b) **FLAG_ON** uses the EXP-B design to compare program performance of experiment with opt_A=ON with the baseline configuration that sets all flags OFF.

Interestingly, we found that only 8 of 60 optimization flags displayed any significant performance impact in our experiments. Therefore, to simplify presentation, we group the remaining 52 flags into a single set that is operated and presented as one unit. Table 2 lists and describes the 8 significant optimization flags. We also observed that several of our optimization flag sets perform transformations or analysis that *enables* method *inlining*. These flag-sets only show a noticeable impact when combined with *inlining*.

Tables 3 and 4 present the results of the FLAG_OFF and FLAG_ON (with inlining always ON) experiments, respectively, for each benchmarks and optimization flag-set. The first column in each of these tables gives the benchmark name. The remaining columns show the performance benefit (in msec) of each individual optimization flag-set in the respective (FLAG_OFF or FLAG_ON) configuration. The FLAG_ON experiment captures the combined effect of opt_A and its enabling

Benchmarks	Optimizations																	
	Inlining	others		UseCHA		UseType Profile		Opto Coalesce		UseLoop Predicate		Loop Peeling		ParseGVN		Peephole Remove Copies		Implicit NullChecks	
		I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI
DaCapo benchmark suite																			
avrora	0.83			0.95		0.79		0.97											
fop	0.65			0.92		0.84										0.96	0.97		
h2	0.70	0.95				0.79		0.90	0.93							0.89	0.92	0.95	
jython	0.50			0.92		0.78		0.97								0.95	0.95		
luindex	0.88					0.31			0.97		0.98		0.99		0.99	0.98	0.96		0.99
lusearch	0.73					0.89													
pmd	0.78			0.96		0.91		0.98	0.96	0.98						0.97	0.96		
sunflow	0.22					0.51													
xalan	0.66			0.95		0.86		0.97	0.96								0.96		
GEOMEAN	0.62	0.95		0.94		0.71		0.96	0.96	0.98	0.98		0.99		0.99	0.95	0.95	0.95	0.99
SPEC JVM 2008 benchmark suite																			
compress	0.58							0.88	0.82	0.90		0.90				0.89	0.83		
crypto.aes	0.38					0.74		0.80	0.92	0.93	0.96	0.83	0.93	0.80	0.92	0.88	0.95		
crypto.rsa	0.20								0.83		0.78		0.89				0.90		
crypto.signverify	0.38							0.89	0.87	0.77	0.89	0.85	0.94	0.85	0.91	0.94	0.87		
derby	0.37			0.89		0.74											0.96		
monte_carlo								0.75	0.96					0.60	0.83	0.76	0.90		
mpegaudio	0.76							0.93	0.90	0.90	0.90		0.95	0.88	0.87	0.95	0.92		
serial	0.41					0.97		0.95	0.95							0.93	0.95		
sunflow	0.32					0.59		0.92						0.95		0.94			
xml.validation	0.64					0.73		0.95	0.96					0.95		0.94	0.95		
GEOMEAN	0.42			0.89		0.74		0.88	0.90	0.87	0.88	0.86	0.93	0.83	0.88	0.90	0.91		
ScalaBench benchmark suite																			
apparat	0.76	0.88				0.39		0.83	0.97	0.97	1.05					0.88	0.96	0.88	
factorie	0.51			0.66		0.30	0.95		0.95								0.96		
kiama	0.59			0.92		0.53													
scalac	0.68			0.82	0.96	0.79										0.96			
scaladoc	0.68			0.85	0.95	0.75		0.97								0.96	0.97		
scalap	0.61			0.89		0.65													
scalariform	0.61			0.90		0.57		0.97								0.97			
scalatest	0.71															1.44			
scalaxb	0.58			0.97		0.64		0.92	0.97							0.92	0.95		
specs	0.72			0.99		0.94		0.99	0.98	0.99				0.99		0.99	0.97		
tmt	0.61			0.96		0.56		0.95	0.97							0.96	0.97		
GEOMEAN	0.64	0.88		0.88	0.96	0.59	0.95	0.94	0.97	0.98	1.05			0.99		1.00	0.96	0.88	

Table 5. Impact of individual flags w/ profiling data enabled

interaction with all other optimizations (including inlining), partially offset by the compensating effect of other optimizations. In contrast, the FLAG_OFF setting captures the effect of opt_A and its interaction *just* with inlining. Thus, if FLAG_OFF > FLAG_ON for some opt_A, then it indicates that the compensating effect dominates any enabling interaction. If FLAG_ON > FLAG_OFF, then the overall enabling interaction between optimizations dominates.

As mentioned earlier, we find that several optimizations have a significant enabling interaction with method *inlining*. This observation becomes manifest from the *Inlining* column in the two tables, which show that for most benchmarks only turning inlining ON (with FLAG_ON) is much less beneficial than the harm of turning inlining OFF (with FLAG_OFF) as many other optimizations are now unable to achieve their full potential without inlining. We also observe that for most other optimization flag-sets that show significant performance benefits, that benefit in the FLAG_ON configuration is often greater than its effect in the FLAG_OFF configuration. These observations suggest that, (a) other than inlining, the effect of optimizations enabling opportunities for one another may not be as pronounced, and (b) there is a significant compensating effect from several optimizations targeting and addressing the same program inefficiency.

Benefit of Individual Optimizations in HotSpot. We next report results from experiments that evaluate the benefit of individual optimizations in HotSpot (with profile data always available). Our baseline configuration turns all optimizations OFF (EXP-B). We have two other configurations for

Benchmarks	Optimizations																			
	Baseline		others		UseCHA		UseType Profile		Opto Coalesce		UseLoop Predicate		Loop Peeling		ParseGVN		Peephole Remove Copies		Implicit NullChecks	
	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI	I	NI
DaCapo benchmark suite																				
avrora	0.94	1.06			0.88		0.74	1.08	0.91											
fop	0.70				0.66		0.59													
h2	0.62	0.85					0.49													
jython	0.55	0.78					0.43				0.39	0.60								
luindex	0.88	0.97					0.27		0.90	0.99	0.90		0.91	0.98	0.91		0.90			
lusearch	0.83	1.03		1.05			0.73											1.05		1.05
pmd	0.70	0.84			0.67		0.63			0.83	0.69							0.83		
sunflow	0.49	1.09					0.26							1.13						
xalan	0.60	0.85			0.57		0.52													
GEOMEAN	0.69	0.93		1.05	0.69		0.49	1.08	0.91	0.90	0.62	0.60	0.91	1.05	0.91		0.90	0.93		1.05
SPEC JVM 2008 benchmark suite																				
compress	0.52								0.61		0.47		0.47				0.60			
crypto.aes	0.39						0.29		0.33				0.35		0.32	0.96	0.36			
crypto.rsa	0.63																	1.07		
crypto.signverify	0.59										0.49		0.54		0.56					
derby	0.38	0.92			0.34		0.28													
monte_carlo									0.79						0.75		0.87			
mpegaudio	0.75															0.96	0.78			
serial	0.47										0.41	0.84								
sunflow	0.68	1.17					0.41		0.64						0.65					
xml.validation	0.43	0.65					0.31										0.37	0.60		
GEOMEAN	0.52	0.89			0.34		0.32		0.57		0.45	0.84	0.44		0.54	0.96	0.56	0.80		
ScalaBench benchmark suite																				
apparat	0.89		0.79	1.03			0.35		0.75		0.81						0.81		0.78	1.02
factorie	0.46	0.72			0.32		0.14													
kiama	0.64	0.94			0.59		0.34													
scalac	0.77	0.95			0.68		0.60													
scaladoc	0.77				0.70		0.57													
scalap	0.66	0.95			0.59		0.43													
scalariform	0.76	1.03			0.69		0.43													
scalatest	0.91																			
scalaxb	0.70						0.46		0.67											
specs	0.82	1.01					0.77													
tmt	0.69	0.97					0.38													
GEOMEAN	0.72	0.93	0.79	1.03	0.58		0.41		0.71		0.81						0.81		0.78	1.02

Table 6. Impact of profiling data on individual optimization flags

each optimization flag that turn ON each one individually (one configuration with method inlining OFF and another with inlining ON).

Table 5 shows the results of this experiment, but only displays numbers that are statistically significant. The first column gives the benchmark name. The next column shows performance improvement when only the optimization of *function inlining* is enabled. All later columns with label 'NI' compare the program performance with one flag-set ON to the baseline performance that disables all optimization flags. The columns with label 'I' compare the program performance with both the one flag-set and inlining ON to program performance with only inlining ON.

Study Impact of Profile Data on Optimizations. We perform another experiment to determine the amount of benefit optimizations in HotSpot's c2 compiler derive from profile data. Even with the EXP-B configuration, showing the degree to which an optimization is affected by profile data is slightly tricky. Simply using the performance of a profiling enabled configuration normalized against the performance of the corresponding profiling disabled configuration is not sufficient because, (a) there may be other compiler optimizations that we are not yet disabling, and (b) profile data may affect other performance-relevant VM tasks. Manifestation of this issue can be seen in the *Baseline-NI* column in Table 6. This column displays the ratio of a configuration that disables all optimizations but has profiling ON to another configuration that also disables all optimizations but has profiling OFF. Ideally, these two configurations should report identical performance numbers. However, several benchmarks, such as *jython* in DaCapo, *xml.validation* in SPEC and *factorie* in

ScalaBench, report significant performance benefit from profile data even when all optimization flags that we control are turned OFF in both configurations. [5]

Table 6 presents the performance impact of profile data for each optimization flag-set. Each number in this table reports the ratio of some flag-set configuration with profiling ON to an equivalent configuration with profiling OFF. Columns labeled with an 'I' turn inlining ON, and those labeled 'NI' turn inlining OFF. The *Baseline* configuration turns OFF all flags. The other columns report results for experiments that turn one flag-set ON as indicated. To calculate the impact of profiling on any flag-set, the numbers reported in the column for that flag-set must be compared with the corresponding numbers in the *Baseline* column. For each flag-set, we only report values that are statistically different from the *Baseline*. That is, we report the values when the either the baseline or experimental error is less than the absolute difference between the baseline and experimental geometric means. We find that other than *inlining*, *UseTypeProfile* appears to be the only other flag-set that derives significant performance benefit from profile data in our experiments.

Overall Observations. We make the following observations from our experiments in this section. We see (from Table 5) that 52 of the 60 flags (that we grouped together as *others*) don't contribute meaningful performance benefits for our benchmarks, even when all are employed as a unit. With inlining off, no benchmark derives significant benefit from this set of 52 optimization flags. Combining function inlining with the *others* set of flags, improves the performance for two benchmarks over our three benchmark suites.

Function *inlining* is by far the most beneficial optimization. The application of *inlining* alone improves average program performance by 38%, 58% and 36% for our three benchmark suites respectively. *Inling* is also an enabling optimization that improves the effectiveness of several optimization flags, especially, *UseCHA* and *UseTypeProfile*. *Inlining* is also the one that is most affected by profile data. We can see from Table 6 that availability of profile data improves the effectiveness of *inlining* alone by 31%, 48%, and 28% for DaCapo, SPEC and ScalaBench, respectively and on average.

Other than *Inlining*, the *UseTypeProfile* flag has the largest improvement over all and is the most affected by profiling data. This flag significantly impacts the effectivness of inlining. Additionally, as the name suggests, it controls access to the use of type profiling information, so it is heavily affected by the presence or absence of profiling information. *UseCHA*, by comparison, provides a significant improvement by itself, mostly for the same reasons as *UseTypeProfile*, but it is less affected by profiling information because it controls a static analysis that doesn't rely on profiling data directly (though it can affect how profiling data is used).

OptoCoalesce, *UseLoopPredicate*, *LoopPeeling*, *ParseGVN*, and *PeepholeRemoveCopies* all show significant performance improvement on SPEC benchmarks but little effect on benchmarks from the other two suites. We believe this happens because the SPEC benchmarks are more numeric in nature, and have smaller, tighter inner loops that benefit from these optimizations. In contrast, idiomatic Scala, as used in the ScalaBench benchmarks, makes frequent use of higher level abstractions – in particular functional combinators – which obscure loops behind deeply nested function calls. Consequentially, these optimizations provide minimal benefit to those programs without inlining enabled.

ImplicitNullChecks provides the smallest benefit of our "important" flags. In particular, it only provides a significant improvement on a couple of benchmarks. The fact that it is unaffected by

[5] Some programs, especially *sunflow* in both DaCapo and SPEC report performance benefit from turning profiling OFF. Such behavior is plausible when profile data gathered during the early program run is not representative of the remaining execution and improperly biases some dependent optimizations. We have not investigated the exact cause in this particular case when all our optimizations are turned OFF.

profiling information is unsurprising as it controls a speculative optimization that is not applied in successive compilations of methods where it proves to be a source of traps.

6 FUTURE WORK

There are multiple avenues for future work. First, one limitation of this work is that it is only conducted for one VM and compiler, the HotSpot VM's c2 compiler. It is important to investigate if the observations we make in this study can be generalized to other dynamic compilers for Java or other languages. Second, this work investigates the different characteristics of PGOs in JIT optimization systems *individually*. In future work, we will study how these aspects affect one another. For example, we can study the interaction of profiling amount and accuracy, or the impact of profile accuracy on individual JIT compiler optimizations, etc. Third, we intend to explore and compare different algorithms to assess profile similarity in future work. Fourth, this work demonstrated the benefit that program performance can derive from profile information. At the same time, we also find that the collected profile knowledge needs to be sufficiently accurate for the current program input for PGOs to realize their maximum potential. Our next research focus will be on how to extract such profile data in systems where online profiling is not feasible, like AOT systems, and how to customize the generated binaries for different input behaviors. This is a broad research issue, with many questions to explore. For instance, similar to categorizing profile data types, can program behaviors also be categorized into a small finite number of behavioral types? Can improvements be made to profile data collection during offline profiling over multiple program inputs so that the dependent optimizations can be specialized to generate variations of binary programs for different behavioral types? Can we build advanced static analysis techniques to improve coverage of offline profiling inputs to encompass all possible program behaviors? Eventually, in the future, we plan to build runtime systems that can combine the advantages of AOT and JIT compilation systems with none, or at least fewer, of the associated drawbacks.

7 CONCLUSIONS

The standard reactive JIT compilation model used in desktop and server VMs can acquire and exploit program profile information from the current run to guide advanced PGOs to generate high-quality native code. In this work we quantify the impact of profile data on the quality of code produced by the JIT compiler, study how the amount and accuracy of profile data affect program performance, and investigate the effect of profile data on individual optimizations in the JIT compiler for dynamic languages like Java. Additionally, we make a number of interesting, and hitherto unknown, discoveries about the properties of profile data that are critical to maximize its ability to correctly guide dependent PGOs. Within the context of HotSpot's c2 JIT compiler and our set of benchmark programs, we find that, (a) PGOs can achieve significant performance benefits on current JVM systems. (b) Even a small amount of profile data collected at program startup can significantly benefit generated code quality as compared to no-profiling. (c) Profile data collected from different program input sets is often able to correctly guide PGOs for most programs. However, a random or wrong profile data may induce program performance that is worse than not using any profile data. (d) A small fraction of profile-site mispredictions can significantly affect the performance of PGOs to generate high-quality code. (e) Most optimizations do not contribute significantly to performance improvement. Profile data appears to be most profitably used by *inlining* and other optimizations that assist with inlining decisions, like *UseTypeProfile*. We design and construct several innovative VM frameworks and experiments to accomplish this work. We believe that our frameworks, experiments, and observations can prove useful to VM developers and researchers to build compilation systems that can combine the benefits of both AOT and JIT based models.

REFERENCES

[1] [n.d.]. DaCapo Batik benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/1.

[2] [n.d.]. DaCapo Eclipse benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/2.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 92, 2 (February 2005), 449–466.

[4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2011. Adaptive Optimization in the Jalapeno JVM. *SIGPLAN Notices* 46, 4 (May 2011), 65–83.

[5] Matthew Arnold and David Grove. 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the Symposium on Code Generation and Optimization*. 51–62.

[6] Matthew Arnold and David Grove. 2005. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *Proceedings of the Symposium on Code Generation and Optimization (CGO '05)*. 51–62.

[7] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-run Profile Repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 297–311. https://doi.org/10.1145/1094811.1094835

[8] Steve Blackburn, Daniel Frampton, Robin Garner, and John Zigman. 2009. dacapo-9.12-bach. http://dacapobench.org/RELEASE_NOTES.txt.

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

[10] William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. 2015. Profile-guided Meta-programming. In *Proceedings of the Conference on Programming Language Design and Implementation*. 403–412.

[11] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. 1991. Using profile information to assist classic code optimizations. *Software Prac. Experience* 21 (1991), 1301–1321.

[12] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-directed Optimization for Warehouse-scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[13] MichałCierniak, Guei-Yuan Lueh, and James M. Stichnoth. 2000. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the Conference on Programming Language Design and Implementation*. 13–26.

[14] Evelyn Duesterwald and Vasanth Bala. 2000. Software profiling for hot path prediction: Less is more. *SIGPLAN Notices* 35, 11 (Nov. 2000), 202–211.

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*. 57–76.

[16] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. *SIGPLAN Notices* 17, 6 (1982), 120–126.

[17] Urs Hölzle and David Ungar. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.* 18, 4 (1996), 355–400.

[18] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. 1–11.

[19] SungHyun Hong, Jin-Chul Kim, Jin Woo Shin, Soo-Mook Moon, Hyeong-Seok Oh, Jaemok Lee, and Hyung-Kyu Choi. 2007. Java Client Ahead-of-time Compiler for Embedded Systems. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*. 63–72.

[20] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: Improving program locality. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. 69–80.

[21] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1-2 (1993), 229–248.

[22] Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer Memory Management for Managed Language Applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 488–504.

[23] Calin Juravle. 2019. Improving app performance with ART optimizing profiles in the cloud. https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html.

[24] Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. 2000. Reducing the overhead of dynamic compilation. *Software: Practice and Experience* 31, 8 (December 2000), 717–738.

[25] Prasad A. Kulkarni. 2011. JIT Compilation policy for modern machines. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*. 773–788.

[26] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/3132190.3132210

[27] Markus Mock, Craig Chambers, and Susan J. Eggers. 2000. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the Symposium on Microarchitecture*. 291–302.

[28] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the Symposium on Code Generation and Optimization (CGO '07)*. 198–208.

[29] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '10)*. 187–197.

[30] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium*. 1–12.

[31] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. 16–27.

[32] Android Open Source Project. [n.d.]. Introducing ART. https://source.android.com/devices/tech/dalvik/.

[33] Forrest J. Robinson, Michael R. Jantz, and Prasad A. Kulkarni. 2016. Code Cache Management in Managed Language VMs to Reduce Memory Consumption for Embedded Systems. In *Proceedings of the Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*. 11–20.

[34] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 140–153.

[35] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. 2000. Quicksilver: A Quasi-static Compiler for Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*. 66–82.

[36] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 657–676.

[37] SPEC2008. 2008. SPECjvm2008 Benchmarks. http://www.spec.org/jvm2008/.

[38] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2005. Design and Evaluation of Dynamic Optimizations for a Java Just-in-time Compiler. *ACM Transactions on Programming Languages and Systems* 27, 4 (July 2005), 732–785.

[39] April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2017)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

[40] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. 2011. A Method-based Ahead-of-time Compiler for Android Applications. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 15–24.

[41] Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *Proceedings of the Symposium on Microarchitecture*. 1–11.