AOT vs. JIT: Impact of Profile Data on Code Quality

April W. Wade
University of Kansas, USA
t982w485@ku.edu

Prasad A. Kulkarni
University of Kansas, USA
prasadk@ku.edu

Michael R. Jantz
University of Tennessee, USA
mrjantz@utk.edu

Abstract

Just-in-time (JIT) compilation during program execution and ahead-of-time (AOT) compilation during software installation are alternate techniques used by managed language virtual machines (VM) to generate optimized native code while simultaneously achieving binary code portability and high execution performance. *Profile* data collected by JIT compilers at run-time can enable profile-guided optimizations (PGO) to customize the generated native code to different program inputs. AOT compilation removes the speed and energy overhead of online profile collection and dynamic compilation, but may not be able to achieve the quality and performance of customized native code. The goal of this work is to investigate and quantify the implications of the AOT compilation model on the quality of the generated native code for current VMs.

First, we quantify the quality of native code generated by the two compilation models for a state-of-the-art (HotSpot) Java VM. Second, we determine how the *amount* of profile data collected affects the quality of generated code. Third, we develop a mechanism to determine the accuracy or *similarity* for different profile data for a given program run, and investigate how the accuracy of profile data affects its ability to effectively guide PGOs. Finally, we categorize the profile data types in our VM and explore the contribution of each such category to performance.

CCS Concepts •**Software and its engineering → Just-in-time compilers; Runtime environments;** *Embedded software*;

Keywords Program profiling, Profile-guided optimizations

1. Introduction

Managed language platforms, such as Java, provide an accessible, secure, platform-independent and high-performance development and run-time environment, and are popular in many embedded and mobile domains. Programs written in these managed high-level languages are distributed in a *machine-independent* binary format that is designed to ease program execution in a virtual machine (VM) running on many different processor architecture and operating system configurations. Program emulation in VMs can be performed by *interpretation* or *binary/native* code execution. Since software interpretation is inherently slow, most performance conscious systems compile (portions of) the distributed code on the target machine prior to execution. Thus, current models ensure program portability while also providing secure sand-boxed and high-performance code execution.

Code compilation in such environments can occur at load-time or run-time. Load-time compilation happens when the program is first installed on the device, and is an instance of the so-called *ahead-of-time* (AOT) compilation model. Execution-time or dynamic or *just-in-time* (JIT) compilation typically occurs *during* (every) program execution, and may compile all or only the frequently executed (or *hot*) sections of the program.

AOT compilation based systems provide several benefits over JIT compilation based VMs. Most prominently, the compilation in such systems is conducted offline and happens only once, rather than during each program execution. This property eliminates the time and energy overhead of online compilation, and is particularly beneficial to short-running programs that have a relatively flat method hotness curve. Additionally, AOT based systems may also reduce the VM complexity by not needing the selective compilation, code cache, and related runtime infrastructure.

The JIT compilation model also has some unique advantages. For instance, VMs can collect *profiling* data to understand and exploit dynamic program behavior during every program run. The JIT compiler can utilize this profile data during profile-guided code optimizations (PGO) to customize the native code for each input and improve overall program performance. Such infrastructure also enables the VM to apply additional aggressive and potentially unsafe optimizations *speculatively*. Speculatively compiled code can be de-compiled if the speculative condition is invalidated later.

The distinction between AOT and JIT compilation models has gained further prominence after Google Android, one of the most popular mobile platforms, replaced its JIT based Java VM (Dalvik) with an AOT-only runtime (ART), and then recently added JIT capability back into ART [1]. Researchers have also explored the use of AOT compilation systems [17, 21, 27, 35] to reduce *startup* time for Java programs on embedded platforms. Despite these developments, the performance implications of AOT vs. JIT compilation are still not clear. This work aims to address this issue by shedding light on how design tradeoffs in managed language compilation systems impact code quality and program performance.

The goals of this research are to: (a) investigate, quantify, and highlight the role of profile data and dependent PGOs to improve generated code quality in managed language runtimes, and (b) understand the challenges that AOT based systems face to generate high-quality code without access to customized and accurate profile data. We develop a variety of innovative experiments and VM frameworks to resolve the following issues. (a) How does the amount of custom (from the same program run) profile data impact the effectiveness of PGOs? (b) How do the inaccuracies in profile data affect the quality of generated code? To quantify the impact of inaccurate *offline* profile data, we develop techniques to systematically introduce noise into the profile data. We also develop

LCTES'17, June 21–22, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081037

[1] See: *https://source.android.com/devices/tech/dalvik/jit-compiler.html*

a mechanism to calculate the *similarity* of pairs of profile data sets. (c) What kinds of profile information are most important to performance? We study the different types of profile data collected by the HotSpot VM and isolate their individual impact.

We make the following contributions in this work limited to the VM (HotSpot) and benchmarks (DaCapo and SPEC-JVM) used.

1. We quantify the ability of custom profile data to generate higher-quality code for the HotSpot VM.
2. We find that even a small amount of accurate profile data can significantly benefit effectiveness of PGOs over no-profiling.
3. We show how the similarity (or *representative-ness*) between the inputs used during the *training* and later *measurement* runs (with *offline* profiling) directly and significantly impacts the quality of code generated by PGOs.
4. We find that making only a small percentage of profiling decisions incorrectly can induce PGOs to generate noticeably poorer quality code.
5. We find that only a small subset of the profile data types collected by the VM produces most performance gains in VMs.

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.

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 than use this profile to guide code optimization decisions. Offline profiling is used by static compilers like GNU gcc/g++ [10, 19, 24, 29]. 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 [6, 11, 28, 34]. Researchers have also developed static analysis techniques to estimate some run-time information for PGOs [36]. 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 [5, 15, 22, 28]. 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 [16], during profile-guided meta-programming [9], to improve code cache management in JVMs [30], to improve heap data locality in garbage collected runtimes [18], to guide object placement in partitioned hot/cold heaps to lower memory energy consumption [20], 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 [4, 12, 25]. 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 [26, 31]. 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 [17, 32, 35]. Instead, in this work we study the effect on generated code quality (i.e., steady-state performance) that is important to longer-running programs.

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 [28]. 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 [22]. 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 AOT and JIT configurations in this work allow deoptimized methods to be recompiled later if they regain hotness.

Benchmark Suites: Our experiments use benchmarks from the DaCapo [8] and SPECjvm2008 suites [33]. Four DaCapo benchmarks, `batik`, `eclipse`, `tradebeans` and `tradesoap` are excluded because they fail to run with the unmodified HotSpot-9.[2] 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* 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 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 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 [13].

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*' ($0<i<N$).

Benchmark	Steady-State run-time (ms)	Methods compiled		
		Startup	Steady	All
DaCapo benchmark suite (*default* input)				
avrora	5710.50	345	507	4152
fop	449.50	554	1301	7460
h2	6927.00	712	995	5151
jython	2917.00	1189	1489	7469
luindex	874.90	259	470	4004
lusearch	2124.90	304	443	3366
pmd	5026.70	901	1526	6121
sunflow	2277.40	279	338	4717
tomcat	6085.40	906	2134	25870
xalan	1532.50	746	1359	5073
SPEC JVM 2008 benchmark suite (*startup* configuration)				
compress	1475.50	48	56	2462
crypto.aes	3658.00	64	79	3197
crypto.rsa	371.70	143	291	3202
crypto.signverify	746.50	115	163	3060
derby	915.60	798	1017	7859
mpegaudio	2631.80	93	107	2602
scimark.monte_carlo	1444.60	29	30	2411
serial	2277.70	270	457	3311
sunflow	1342.80	239	307	3882
xml.transform	1008.20	868	1474	6654
xml.validation	667.90	419	838	4532

Table 1. Relevant benchmarks properties

4. Constructed Experimental Frameworks

We implement many new mechanisms in the HotSpot VM to correctly and fairly conduct our experiments for this study. 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 in the literature, we make a small update to implement this functionality in the VM [23].

We add an empty *VM-indicator* method to the DaCapo and SPEC harness 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.

[2] `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 [7].

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* [14]) 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.

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 did not yet attempt to resolve or predict 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.

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 build 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 [3].

The representative-ness or similarity of two collected profile data instances is a factor of the dependent PGO. We identified 60 *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. These results indicate the challenges that AOT compilation systems, whether based on offline profiling or static analysis, face to achieve the performance of JIT compilation systems that have access to profile information from the current run.

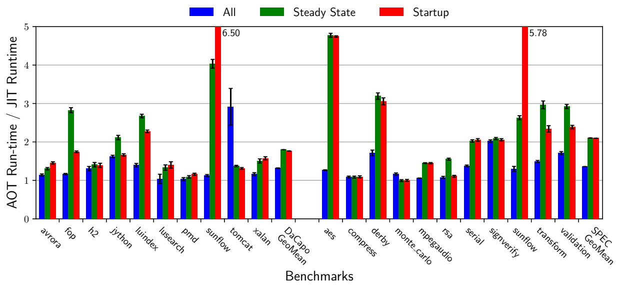

Figure 1. Profile data and PGOs have a significant impact on program performance on the HotSpot JVM

5.1 Impact of Profiling on Generated Code Quality

Our first set of experiments are designed to quantify the impact of profiling information on generated code quality. Program execution time serves as our metric for measuring code quality. We prepare five distinct HotSpot configurations to compare the behavior and performance of AOT and JIT compilation systems.

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 third 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 second 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.

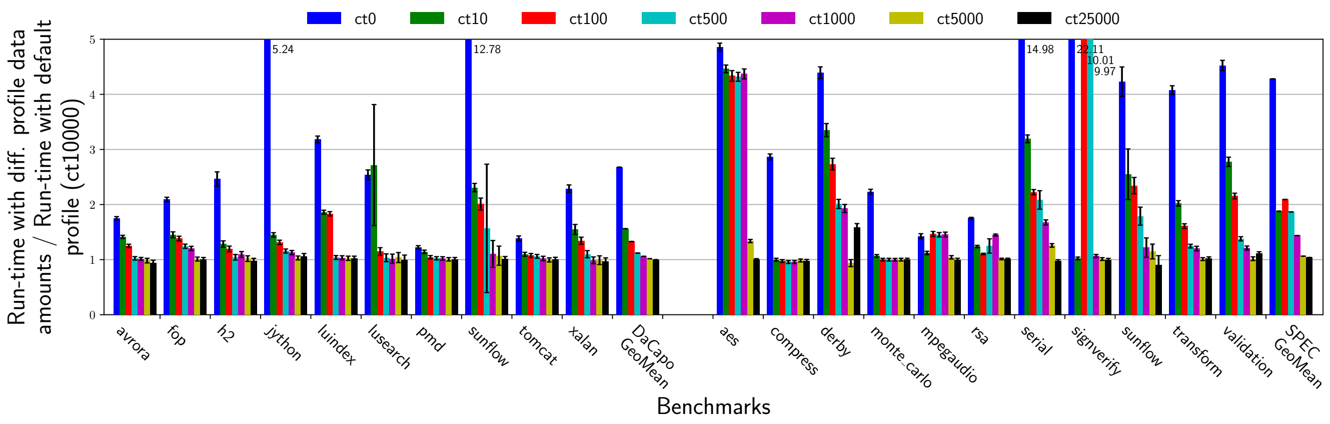

Figure 2. A small amount of profile data from the current program run is sufficient to effectively guide PGOs on the HotSpot JVM

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, 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 the program profile behavior, which is extensively used by PGOs in current VMs for languages like Java, significantly influences the quality of generated code. AOT compilers may have to rely on mechanisms like offline profiling or static analysis to address this potential loss in performance. However, these alternative mechanisms have other limitations. In later sections we attempt to study the challenges that AOT compilers may need to overcome when using these alternative mechanisms to estimate profile information.

5.2 Impact of Profile Data *Amount* on Code Quality

Offline profiling that can be used to drive PGOs in an AOT compiler can collect profile data for the entire duration of the *training* program run, or even over multiple offline runs using different training inputs. In contrast, JIT compilation systems need to balance the amount of profile data collection with the delay in making optimized code available to the emulation engine. 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.

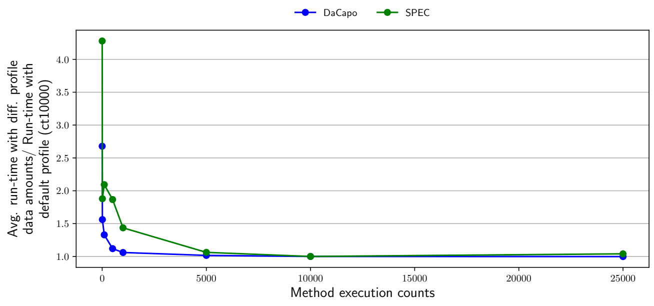

Figure 3. Average program performance quickly improves and reaches saturation with small increases to the amount of collected program profile information.

Figures 2 and 3 show the results of this experiment. We observe slightly different trends for DaCapo and SPECjvm benchmarks. Overall, one surprisingly 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 5000 with SPEC. These results suggest that offline profiling conducted over long time intervals may not have much of an advantage over traditional online profiling based JIT compilation systems.

We also find that, unlike the DaCapo 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*). Preliminary analysis shows 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

A fundamental limitation of AOT compilation systems is that they cannot customize the single statically generated binary to all program inputs possible at run-time. These systems can still employ static analysis or offline profiling to guide PGOs. However, several issues remain unresolved. 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

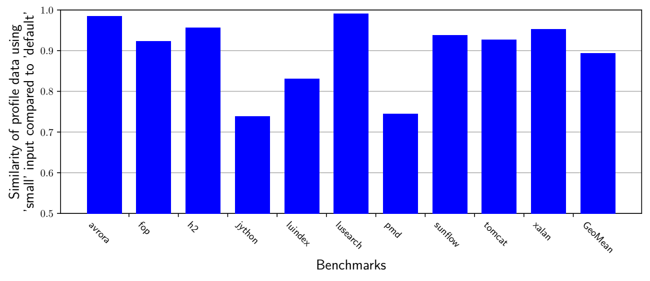

Figure 4. DaCapo's *small* input can closely represent the program behavior of a run with DaCapo's *default* input for guiding HotSpot's PGOs.

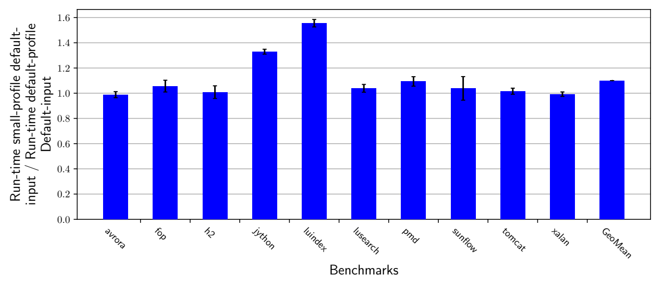

Figure 5. In most cases, offline profiling using DaCapo's *small* input produces a binary that achieves good performance with a later evaluation run with the *default* input

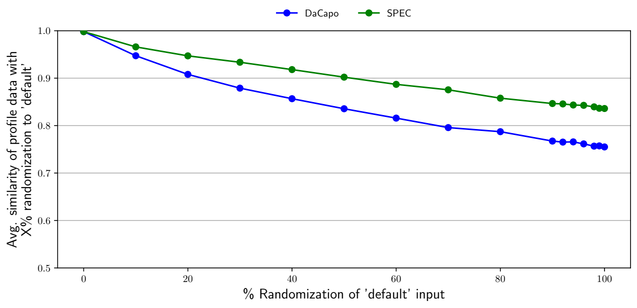

Figure 6. Average representative-ness of the profile trace for various randomization configurations as compared to HotSpot's default *reactive* configuration

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 DaCapo *small* Input

The DaCapo suite provides two distinct input settings that are common for each benchmark program, called *small* and *default*. In this section, we evaluate the effectiveness of recording the program behavior with the *small* input, and using that data to guide PGOs during an evaluation run with the *default* input set.

We use the setup described in Sections 4.2 and 4.3 for these experiments. We compare the run-times from two configurations for each benchmark. The first configuration instantiates the training run for each benchmark with DaCapo's *default* program input. The profile data and method compilation order collected by the training run are exported. The evaluation runs again use the same *default* input size. At the end of the 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 application resumes after all the compilations finish. We allow the program run to stabilize over the next few iterations before recording the benchmark run-time.

The second *offline-profiling* configuration follows a setup very similar to the first. In this case though the training run is instantiated with DaCapo's *small* program input, and it exports the profile data collected. The evaluation run loads the profile data exported by this training run, but uses the method compilation order from the first configuration to compile the same set of hot methods. The program run-time is again recorded at the end of 12 benchmark iterations.

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 program inputs. Figure 4 uses this mechanism to quantify the representative-ness of DaCapo's *small* input set compared to the *default* input for each benchmark. We find that with an average similarity score of almost 90%, program behavior with DaCapo's *small* input is quite representative of its behavior with the *default* input, with regards to guiding the PGOs in the HotSpot JVM.

Figure 5 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 the exceptions of *jython* and *luindex*, the high similarity of the *small* and *default* DaCapo inputs does indeed translate to good performance for the offline-profiling configuration. On average, performance with offline-profiling only shows a degradation of 10% compared to the first configuration that has access to profile data from the same run.

5.3.2 Offline Profiling with *Randomized* Program Input

Although DaCapo's *small* input set generates a representative profile 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.

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`

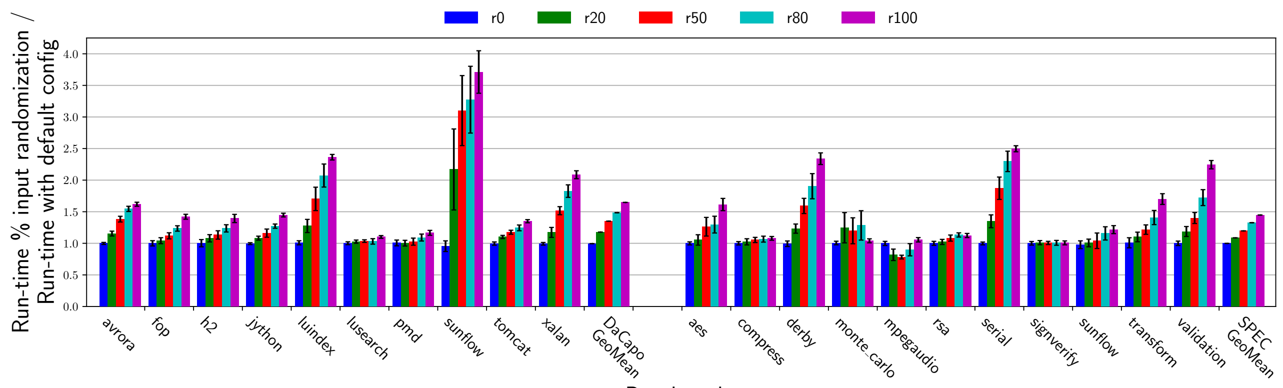

Figure 7. Impact of varying profile data inaccuracy on *individual* program performance on the HotSpot JVM

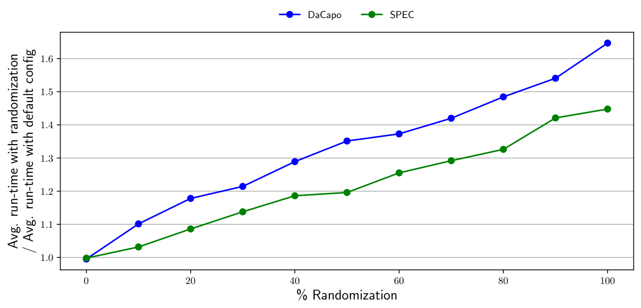

Figure 8. Impact of varying profile data inaccuracy on *average* program performance on the HotSpot JVM

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 (76% for DaCapo and 84% for SPEC), indicating that even vastly different profiles result in the compiler making similar decisions 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 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 programs notice some jitter. We believe this effect is again a result of the nature of the SPEC benchmarks that have fewer and more prominent hotspots. 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, which we will study in future work.

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 60 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 `All_traps` profile-sites (see Table 2) produces high instability in HotSpot and causes a very high num-

[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.

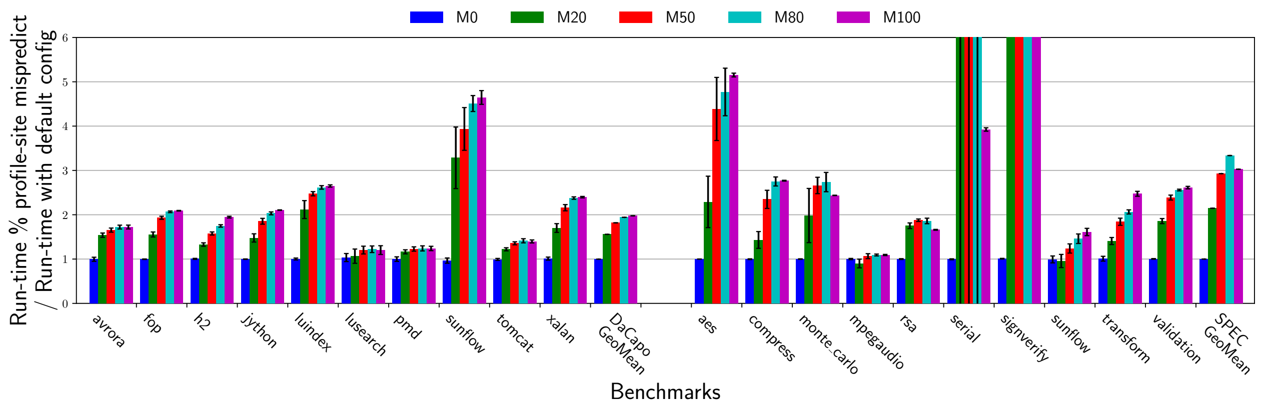

Figure 9. Increasing the probability of mispredicting at a profile-site branch increases the negative impact on the quality of generated code on the HotSpot JVM (individual benchmark view)

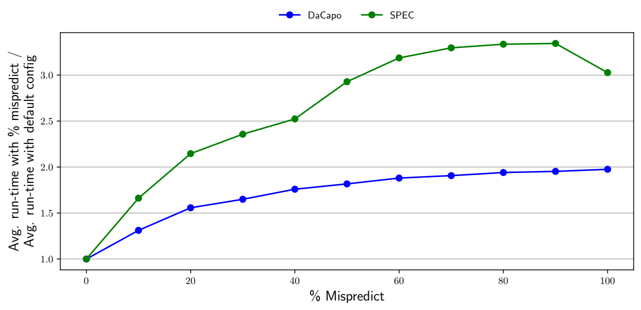

Figure 10. Impact of varying the probability of mispredicting at a profile-site branch on *average* program performance

ber 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.

5.4 Contribution of Profile Data Types to Performance

In this section, we investigate the relative importance of different profile data types to performance. If certain profile types are more important to PGO effectiveness, then researchers may be able to focus their efforts to more precisely predict those values using static analysis or other techniques.

We manually studied the profiling data types used by HotSpot and categorized them into eight sets. Our eight profile data-type categories are described in Table 2. The first column gives a name to each category, and the second column provides a short description of data values in each category. The third column in Table 2 gives the number of profile sites that use a data-type from a particular category to make optimization decisions in the compiler.

To quantify the performance impact of each category of profile data-types, we designed experiments that execute benchmarks with certain sets of profile sites *disabled*. We disable a particular profile site by forcing the compiler to take the path that would be taken if no profile data were available at that point. Otherwise, if the site is enabled, the VM simply uses the online profiling information collected earlier in the same program run.

Our first experiment runs the benchmarks with all of the profile sites disabled (`enable_none`). Ideally, the `enable_none` configuration should produce performance similar to that achieved by simply disabling all profiling in HotSpot (`disable_HS_prof`).[4] The first bar in Figure 11 plots the performance achieved by the `enable_none` configuration as compared to that obtained by `disable_HS_prof`. For many benchmarks, execution time with `enable_none` is very close to that obtained by `disable_HS_prof`. However, the performance varies for some benchmarks, which may be due to uses of profile data that we do not catch or other interactions within HotSpot that we will investigate in future work.

Our next set of experiments evaluates the impact of each individual profile data-type category by disabling all profile sites except for the sites belonging to a single category. We find that except for `call_site_count`, none of the other categories significantly affect program performance, when enabled in isolation. The second bar in Figure 11 shows the performance benefit of enabling only the profile-sites in the `call_site_count` category.

Next, we conduct experiments that simultaneously enable profile-sites from `call_site_count` and one other profile-data category at a time. These experiments reveal that the `null_at_BCI` and `receiver_class_count` show a positive performance effect when enabled in combination with `call_site_count`.

Our final experiment enables all 16 profile-sites belonging to these three important profile-site categories. The third bar in Figure 11 plots the result of this experiment (`enable_imp_flags`) compared to `disable_HS_prof`. Additionally, the final bar in Figure 11 compares program run-time of HotSpot's default *reactive* configuration to `disable_HS_prof`. The `enable_imp_flags` configuration obtains performance that is only 7.9% and 14.1% worse than *reactive* for DaCapo and SPEC benchmarks respectively, on average. Thus, only enabling these three important profile-site categories (16 out of 60 profile-sites) achieves 95.3% of the potential performance benefit provided by PGOs in HotSpot for both DaCapo and SPEC benchmarks, on average.

[4] `ProfileInterpreter` and `ProfileTraps` command-line flags can disable all interpreter profiling in HotSpot.

Name	Description	# sites
All_traps	Determines whether a trap event, such as an array-out-of-bounds exception, occurs at a particular BCI or in a given method.	30
Null_at_BCI	Whether or not a null was observed at a particular BCI. Affects implicit null-check optimizations.	7
Unique_receiver_class	Which subclass is the dynamic type of `this` for a virtual call. Affects type speculation and inlining of virtual calls.	2
Receiver_class_count	The count of receivers for a virtual call, if multiple were observed. Affects type speculation.	1
Klass_for_call	The dynamic types of the arguments and/or return values for a call. Affects type speculation.	3
Inv_loop_counters	Influences "warm-call", or relatively but not absolutely hot call, inlining.	5
Branch_data	Whether or not a particular branch was taken. For `switch` structures, which `case` was taken.	4
Call_site_count	The number of times a particular call was executed. Affects inlining.	8

Table 2. Categories of Profile Data Types in the HotSpot VM

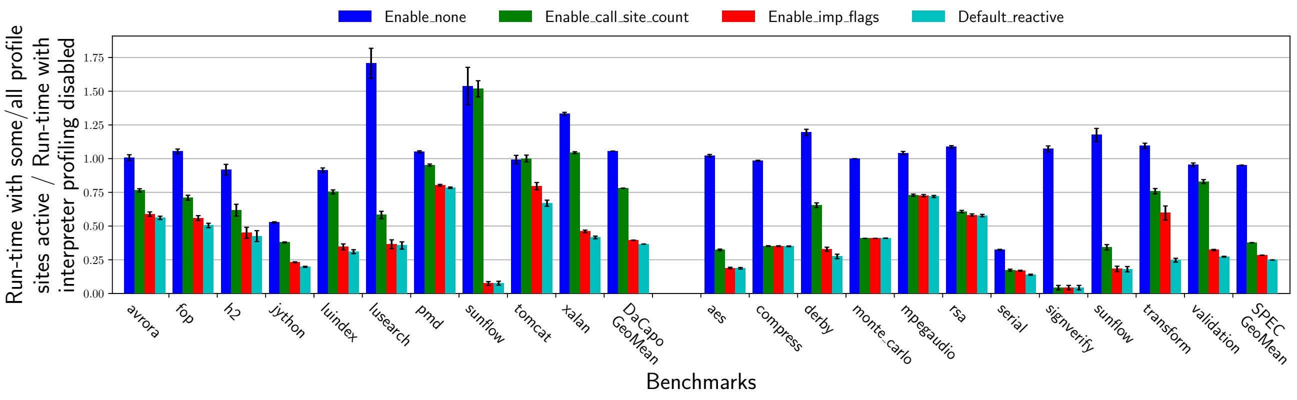

Figure 11. A small subset of the different types of profile data produce the major impact on the quality of generated code. `Enable_none` simultaneously disables all 60 profile sites. `Enable_call_site_count` only enables the 8 flags belonging to that category. `Enable_imp_flags` enables the 16 flags belonging to the *null_at_BCI*, *receiver_class_count* and *call_site_count* categories. Enabling the 16 *important* flags achieves performance very close to that produced by the default *reactive* HotSpot configuration.

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 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. AOT compilation systems popular in embedded systems typically lack access to such reliable profile data, which can restrict their effectiveness. In this work we quantify the impact of profile knowledge on the quality of code produced by JIT optimization systems 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. In particular, we find that (a) even a very little amount of profile data can significantly benefit generated code quality as compared to no-profiling. (b) small imperfections in profile data can have noticeable performance implications, (c) a small fraction of profile-site mispredictions can significantly affect the performance of PGOs to generate high-quality code, and (d) although sophisticated VMs, like HotSpot, collect several varieties of profile data, only a few profile data types induce most of the benefits from dependent PGOs. 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.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback. This work is supported by the National Science Foundation, under CAREER award CNS-0953268, and grants CCF-1619140, CCF-1617954, and CNS-1464288.

References

[1] Dacapo batik benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/1.

[2] Dacapo eclipse benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/2.

[3] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the Symposium on Code Generation and Optimization*, CGO '05, pages 51–62, 2005.

[4] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the Symposium on Code Generation and Optimization*, pages 51–62, 2005.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. *SIGPLAN Notices*, 46(4):65–83, May 2011. ISSN 0362-1340.

[7] S. Blackburn, D. Frampton, R. Garner, and J. Zigman. dacapo-9.12-bach. http://dacapobench.org/RELEASE_NOTES.txt, 12 2009.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 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, pages 169–190. ACM, 2006.

[9] W. J. Bowman, S. Miller, V. St-Amour, and R. K. Dybvig. Profile-guided meta-programming. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 403–412, 2015.

[10] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21:1301–1321, 1991.

[11] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

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

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*, pages 57–76, 2007.

[14] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, 1982.

[15] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996. ISSN 0164-0925.

[16] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, 2013.

[17] S. Hong, J.-C. Kim, J. W. Shin, S.-M. Moon, H.-S. Oh, J. Lee, and H.-K. Choi. Java client ahead-of-time compiler for embedded systems. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 63–72, 2007.

[18] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 69–80, 2004.

[19] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7 (1-2):229–248, 1993. ISSN 0920-8542.

[20] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi. Cross-layer memory management for managed language applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 488–504, 2015.

[21] D.-H. Jung, S.-M. Moon, and H.-S. Oh. Hybrid compilation and optimization for java-based digital tv platforms. *ACM Trans. Embed. Comput. Syst.*, 13(2s):62:1–62:27, Jan. 2014.

[22] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8): 717–738, December 2000.

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

[24] M. Mock, C. Chambers, and S. J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the Symposium on Microarchitecture*, pages 291–302, 2000.

[25] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the Symposium on Code Generation and Optimization*, CGO '07, pages 198–208, 2007.

[26] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '10, pages 187–197, 2010.

[27] H.-S. Oh, J. H. Yeo, and S.-M. Moon. Bytecode-to-c ahead-of-time compilation for android dalvik virtual machine. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pages 1048–1053, 2015.

[28] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[29] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, 1990.

[30] F. J. Robinson, M. R. Jantz, and P. A. Kulkarni. 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*, pages 11–20, 2016.

[31] S. Rubin, R. Bodík, and T. Chilimbi. 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, pages 140–153. ACM, 2002.

[32] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A quasi-static compiler for java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 66–82, 2000.

[33] SPEC2008. Specjvm2008 benchmarks. http://www.spec.org/jvm2008/, 2008.

[34] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Transactions on Programming Languages and Systems*, 27(4):732–785, July 2005.

[35] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 15–24, 2011.

[36] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the Symposium on Microarchitecture*, pages 1–11, 1994.