Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems

Humphrey Burchell
University of Kent
Canterbury, United Kingdom
h.burchell@kent.ac.uk

Octave Larose
University of Kent
Canterbury, United Kingdom
o.larose@kent.ac.uk

Stefan Marr
University of Kent
Canterbury, United Kingdom
s.marr@kent.ac.uk

Abstract

Profilers are crucial tools for identifying and improving application performance. However, for language implementations with just-in-time (JIT) compilation, e.g., for Java and JavaScript, instrumentation-based profilers can have significant overheads and report unrealistic results caused by the instrumentation.

In this paper, we examine state-of-the-art instrumentation-based profilers for Java to determine the realism of their results. We assess their overhead, the effect on compilation time, and the generated bytecode. We found that the profiler with the lowest overhead increased run time by 82×. Additionally, we investigate the *realism* of results by testing a profiler's ability to detect whether inlining is enabled, which is an important compiler optimization. Our results document that instrumentation can alter program behavior so that performance observations are unrealistic, i.e., they do not reflect the performance of the uninstrumented program.

As a solution, we sketch late-compiler-phase-based instrumentation for just-in-time compilers, which gives us the precision of instrumentation-based profiling with an overhead that is multiple magnitudes lower than that of standard instrumentation-based profilers, with a median overhead of 23.3% (min. 1.4%, max. 464%). By inserting probes late in the compilation process, we avoid interfering with compiler optimizations, which yields more realistic results.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

ACM Reference Format:
Humphrey Burchell, Octave Larose, and Stefan Marr. 2024. Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3679007.3685058

MPLR '24, September 19, 2024, Vienna, Austria

1 Introduction

Profilers are the go-to tool for developers to identify the program part that takes up most time and may benefit from optimization. Instrumentation-based profilers insert probes into the program, which then collect information about the program's behavior. In contrast, sampling periodically interrupts the program to collect information, e.g., records the program stack, to derive a probabilistic picture of the program's behavior. Both approaches are widely used for ahead-of-time- as well as just-in-time-compiled language implementations [8, 15, 19, 20].

Unfortunately, both sample- and instrumentation-based profiling of just-in-time-compiled programs affect program execution, reducing the accuracy of profiling results [3, 10, 14, 21], i.e., results may not represent the true performance behavior. Instrumenting code at the source or bytecode level changes how it is optimized [10]. Sampling suffers from safepoint bias, which means profilers do not sample all program parts with equal probability. Safepoint bias can also lead to misinterpreted samples, since the profiler only has a partial understanding of how the compiler altered a program's structure to achieve better performance [3, 14].

The one benefit of instrumentation is its high precision. Results may not be accurate, but probes count or measure reliably. Thus, we want to better understand instrumentation-based profilers on the Java Virtual Machine (JVM). To this end, we measure their overhead, which is typically increasing run time by one or two orders of magnitude. Furthermore, we assess the impact of instrumentation on compilation, and find that they can not detect whether inlining is enabled or disabled, which means their results are unrealistic.

As a way forward, we propose a new approach to instrumentation-based profiling on top of JIT compilers that improves the accuracy of profiles compared to source- and bytecode-level instrumentation by only instrumenting the code late in the compilation process.

Inspired by Basso et al. [2], we insert our instrumentation with a compiler phase of the Graal JIT compiler. This avoids changing which methods are selected for compilation, which methods are inlined, and it minimizes the impact from how the compiler optimizes the code. This is similar to instrumenting binaries of ahead-of-time-compiled programs [18], but with a lower engineering effort and thus, we believe, a suitable way forward for just-in-time compilers.

2 Background

This section introduces profiling, our terminology, the Graal compiler, and the challenges profilers have with inlining.

2.1 Profilers and Profiling Techniques

Profiling allows developers to observe a program's execution. A profiler may record, e.g., a program's CPU, GPU, or memory utilization. Profilers help identify performance issues, e.g., by pinpointing sections of code that consume the most CPU time, and thus, they can guide optimization efforts.

Instrumentation-based profilers insert probes into a program to record the information. For example, a probe at the beginning of each method can count how many times a method is called. This finds frequently called methods and can enable developers to identify underlying performance problems. Probes can be inserted at the source, bytecode, or native code level. We will evaluate JProfiler,[1] VisualVM,[2] and YourKit[3] as state-of-the-art instrumentation-based profilers.

CPU sampling interrupts the program to gather a snapshot of the current state of both hardware and software. This includes recording the call stack, instruction pointer, memory usage, and thread state, which are used to construct a profile. The interrupts occur at regular intervals, but could be random [14]. Safepoints [1] are crucial for garbage collection and to ensure that the VM is in a state where the stack can be correctly read and the program counter can be used to indentify the currently executing method. Thus, samples are typically interpreted based on the closest safepoints. However, this can lead to inaccurate profiles [14].

2.2 Accuracy, Precision, and Realism

Accuracy and precision are often used interchangably. However, in our work we use two distinct meanings.

Accuracy measures how close the reported profiler results are to what happens during executions without a profiler, i.e., how close the results are to the ground truth, which we unfortunately cannot determine directly. *Precision* refers to how close measurements are to each other between runs. Thus, it measures how consistent a profiler gives the same, but not necessarily correct answer.

As *realism* we understand a weaker notion of accuracy, which measures how close the reported profilers results are to an execution with a profiler that does not affect run-time optimizations. This allows for the general impact and bias introduced by using a profiler, but is meant to allow us to assess how instrumentation influences optimization heuristics. For instance, we would consider a profiler unrealistic when it significantly changes the effectiveness of specific optimizations, because the normal execution would not be subject to this change, and would show different performance properties.

[1] https://www.ej-technologies.com/products/jprofiler/overview.html
[2] https://visualvm.github.io/
[3] https://www.yourkit.com/

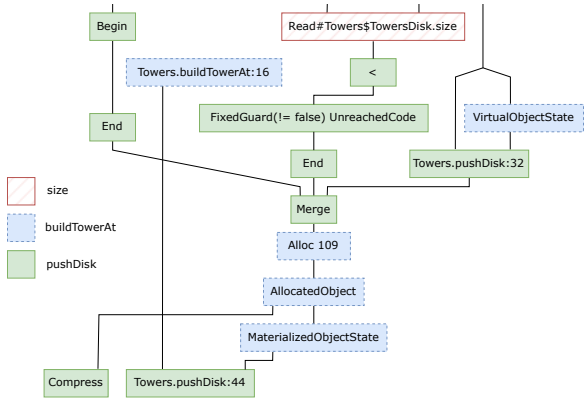

Figure 1. A GraalIR graph showing that nodes from inlined methods can end up being intermixed in a compilation unit.

2.3 Graal Compiler

The Graal compiler is a just-in-time (JIT) compiler implemented in Java. It compiles Java bytecode at run time to machine code. For this, it uses GraalIR, a graph-based *sea of nodes* [4] intermediate representation (IR) with explicit control flow edges [5]. It enables optimizations such as dead code elimination, loop transformation, and inlining by adding, transforming, or removing graph nodes. Each optimization is typically implemented in its own compiler phase.

2.4 Profilers and Inlining

Inlining replaces a method call with the body of the called method. This is a vital optimization, because it enables optimizations on the combination of caller and callee. Graal does inlining at the GraalIR level. After inlining the nodes from a callee, optimizations such as loop peeling can move and duplicate nodes and nodes from different methods can end up mixed together. Figure 1 illustrates this for the Tower benchmark [13] with nodes from the `buildTowerAt`, `size`, and `pushDisk` methods, each in a different color, without clear method boundaries between them. This makes it impossible to simply instrument the beginning and end of an inlined method, since they no longer exist.

Inlining thus complicates determining where time is spent. A sampler needs to know which method the currently executed instruction belongs to. For instrumentation, it depends on when probes are inserted. If they are inserted at the source or bytecode level, probes may be duplicated with the rest of the code, for instance during loop peeling, increase the overhead, and likely prevents optimizations, e.g., inlining [10].

If probes are added after inlining, then it may require many probes to isolate the different parts, and correctly attribute the execution time. This would be needed in our example in Figure 1 to accurately distinguish the methods. A simple profiler may only instrument the root method and attribute the time of the whole compilation unit to this method. However, a major part of the time may be spent on inlined code.

3 State of the Art in Instrumentation on JIT-Compiling JVMs

We will now analyze instrumentation-based profilers by examining their overhead, assessing their impact on the compilation process, and evaluating the realism of their results.

3.1 Experimental Setup

We ran our experiments with Graal on top of the HotSpot JVM in OpenJDK 21.0.2.[4] By using Graal's *libgraal* variant, we ensure that Graal itself is ahead-of-time compiled, which ensures the best possible compilation times from the start. All benchmarks were run on a machine with an AMD Ryzen 5 36000 6-core processor, which uses the Zen 2 architecture, 32 GB DDR4 RAM, and Rocky Linux 9.4 with a Linux kernel version 5.14.0. We chose to use the Are We Fast Yet benchmarks [13], because they are well-understood and deterministic. The suite includes 5 macro-benchmarks and 9 micro-benchmarks. We configured the benchmarks so that a single iteration takes about 100ms and we run each benchmark for 300 iterations. In this configuration, the benchmarks quickly reach a stable state and it is a good trade-off between overall run time and the number of measurements collected. All profilers profile immediately from the benchmark start. The benchmarks are executed with ReBench [12], which adapts the system's settings to minimize interference and noise.

3.2 Assessing Run-time Overhead

To assess the overhead instrumentation-based profilers introduce, we measure for each profiler its default settings using full instrumentation, i.e., without excluding any packages. This means for example that Java's standard library is instrumented, too. As a consequence of full instrumentation, we ran the benchmarks for only 10 iterations, because the high overhead made running the benchmarks for longer impractical. We verified that the relevant JIT compilation still occurs within the first iteration of the benchmark run.

Unfortunately, VisualVM does not seem to be scriptable and we could not execute our benchmarks automatically. This made it impractical run all experiments. Though, we ran the DeltaBlue benchmark, which is roughly in line with the other profilers, and we report results for it where relevant.

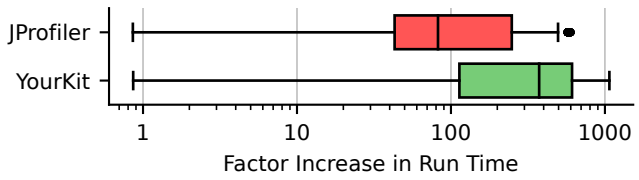

Figure 2. Overhead of instrumentation-based profilers for the Are We Fast Yet benchmarks. The overhead is expressed as a factor over the uninstrumented run time.

[4]We used a Graal from April 2024: https://github.com/oracle/graal/commit/249d3e4abd2f357461c5ceb682791e22b2c8a92f

Figure 2 reports the run-time overhead over the uninstrumented execution of all benchmarks. While fully instrumenting a program is expected to result in high overhead, the extent of this overhead can be substantial enough to render the use of a instrumentation-based profiler impractical. For VisualVM, which is not in the figure, profiling DeltaBlue increases the run time by 485×. This overhead is similar to the median overhead we found for YourKit and JProfiler.

3.3 Assessing Impact on Compilation

To understand why these profilers incur such high overhead, we assess the impact of instrumentation on the amount of code generated, and how it affects inlining. We extracted these details from Graal's compilation log, which is enabled with `-Djdk.graal.PrintCompilation=true`.

Impact on Code Size. For each profiler, we collect the time spent in compilation, the total amount of bytecode, the size of the generated native code, and the memory allocations that occurred during compilation for each of our benchmarks.

The overhead we saw in Figure 2 is likely due to the added instrumentation itself. The increased bytecode and native code size seen in Figure 3 suggests that the probes cause the additional code and that the compiler is unable to optimize the instrumented code effectively.

When YourKit is attached, the added instrumentation causes the native code size to increase by a median of 341%,

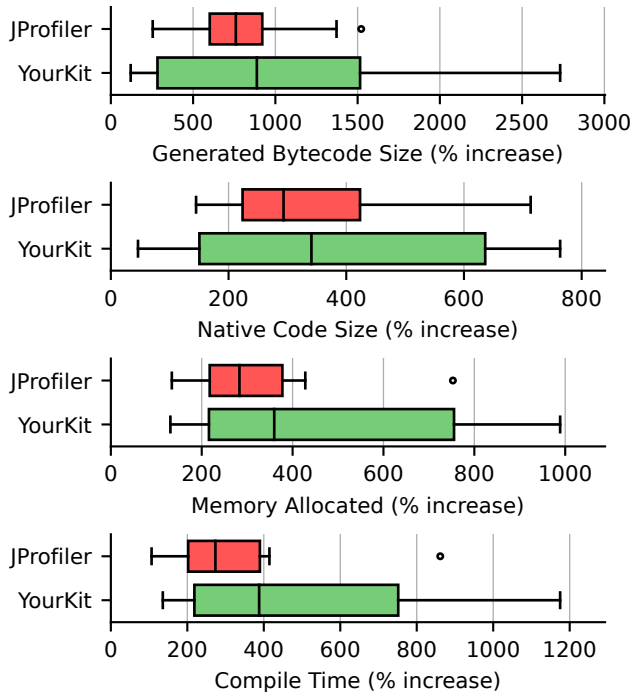

Figure 3. Increase of bytecodes, native code size, memory allocation, and compile time for attached instrumentation-based profilers on the Are We Fast Yet benchmarks.

which contributes to the overhead of 374×. The instrumentation likely also changes inlining and optimization decisions, further contributing to the slowdown. For example, instrumentation can cause some methods to become too large to be inlined, thus affecting the overall performance [10]. The increased code size and the resulting changes to optimization decision suggests that any results obtained from such profilers are likely less realistic than for other types of profilers.

Impact of Instrumentation on Inlining. We have measured how inlining statistics change when instrumentation profilers are attached. The data for Figure 4 was collected from Graal's inlining log. When running the benchmarks without a profiler, the number of inlined methods is much lower, indicating that the instrumentation requires many more instrumentation-related methods to be inlined, which explains the earlier seen increase in native code size.

These numbers also suggest that the instrumentation is likely to dominate the execution of the benchmarks. For instance, small methods will end up consisting of more instrumentation code than application behavior. At the same time, the compiler is not able to remove the instrumentation, because it is indistinguishable from normal application code. As a result, the behavior of an instrumented program likely correlates with method activation counts. This is a strong indication that the profiled behavior does likely not resemble the normal program behavior, making profiling results "unrealistic," since optimizations do not give the same benefit anymore, but often change the performance behavior of a program significantly.

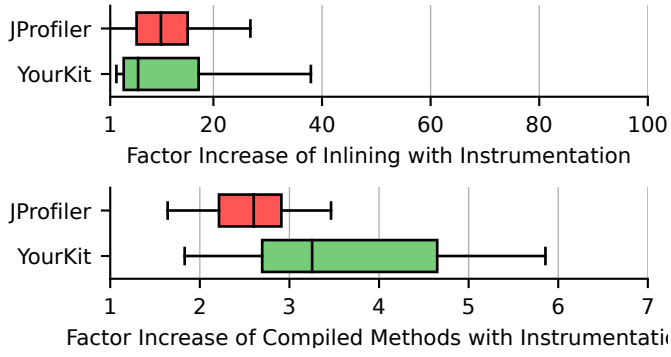

Figure 4. The increase in compiled methods and inlining caused by attached instrumentation profilers for AWFY benchmarks. The increase is expressed as a factor compared to the results of uninstrumented executions.

To understand better how much instrumentation changes the benchmark behavior, we ran the DeltaBlue benchmark with each instrumentation profiler, once normally, and once with inlining disabled.[5] We would expect a drastic change in behavior between inlining enabled and disabled.

[5]Inlining is disabled with the Graal compiler flag `-Dgraal.Inline=false`

Table 1. Profiling results for Async and JProfiler, with and without inlining enabled.

Profiler	Inline	Method	%
Async	yes	deltablue.Plan	23.3
		Vector.forEach	17.7
		ScaleConstraint.execute	4.9
		Vector.append	3.7
		ScaleConstraint.recalculate	3.5
	no	EqualityConstraint.execute	19.9
		ScaleConstraint.execute	8.2
		DMH.newInvokeSpecial	4.5
		Plan$$Lambda.apply	3.7
		Variable.getValue	3.6
JProfiler	yes	EqualityConstraint.execute	14.0
		Plan.lambda$execute$0	10.0
		Vector.forEach	9.0
		Variable.getValue	6.0
		ScaleConstraint.execute	5.0
	no	EqualityConstraint.execute	14.5
		Plan.lambda$execute$0	10.0
		Vector.forEach	8.0
		Variable.getValue	6.0
		ScaleConstraint.execute	5.0

We ran this experiment also with the Async-profiler, which uses sampling instead of instrumentation. We assume that sampling provides results closer to the ground truth, since it does not alter program behavior as much as instrumentation.

Table 1 shows the results for Async and JProfiler. The full results are in the appendix in Table 2. These tables show that the profiles with and without inlining are close to identical for the instrumentation-based profilers. For Async, the sampling profiler, this is however not the case. Here enabling inlining drastically changes the profile as we would expect.

JProfiler reports the methods `Plan.lambda$execute$0`, `Vector.forEach`, and `EqualityConstraint.execute` as the ones taking most time, with and without inlining. Without inlining, Async reports `EqualityConstraint.execute`, `ScaleConstraint.execute`, and `newInvokeSpecial` from the JVM's method handle system as most important. Though with inlining, it reports `deltablue.Plan`, `Vector.forEach` and `ScaleConstraint.execute`, which suggests that inlining and subsequent optimizations change the observable behavior significantly.

For VisualVM and YourKit, inlining has also no major effect on the profiles. To us, this means that the instrumentation prevents us from seeing the impact of inlining, which itself enables many subsequent optimizations.

With this, we conclude that the probes used in state-of-the-art instrumentation approaches change the application behavior to such a degree, that the run-time behavior becomes unrealistic.

3.4 Instrumentation Bias towards Activation Count

When instrumentation alters an application's behavior, to such a degree that profiles strongly correlate with activation counts, they may no longer provide actionable guidance. Simply optimizing the most activated method may not be feasible or provide the desired performance gains, because the compiler may have already realized these benefits.

Figure 5 illustrates a worst-case scenario where a profile based on activation counts may misdirect optimization efforts. In this example. the `execute()` methods of `ActionA` and `ActionB` could be identified as most called. However, optimizing them is likely fruitless. A realistic profile would likely direct attention to the part of the program that dominates run time after optimizations, e.g., the bubble sort.

```
class ActionA { int id; void execute() {} }
class ActionB { int id; void execute() {} }
var actions = getMixOfManyActions();
bubbleSortById(actions);
framework.execute(actions);
```

Figure 5. Worst-case scenario for instrumentation profilers. Highest activation counts may misguide optimization efforts.

4 Improving Realism with Late-compiler-phase-based Instrumentation

In this section, we sketch late-compiler-phase instrumentation to improve realism of profiles and account for inlining.

4.1 Late-compiler-phase Instrumentation

Late-compiler-phase instrumentation inserts probes into compilation units in the latest practical JIT compiler phase to avoid interfering with optimizations. When we insert probes, most optimizations are already applied, which minimizes the observer effect and run-time overhead.

At the high-level, the compilation process remains unchanged, too. The JVM uses its normal heuristics to select a method for JIT compilation and the Graal compiler optimizes it with its many phases. Our implementation adds two phases to the process. The first is placed late in the highest tier, where high-level information is still available, which we use to collect details about the compiled method and methods that have been inlined. We also prepare the resolution of a memory address for our second phase. Though, our first phase does not insert any instrumentation nodes, which avoids interfering with optimizations.

Our second phase is added as late as possible to the low tier and adds our instrumentation nodes. These nodes record the CPU cycles at the start and each exit from the compilation unit. It also instruments calls into non-inlined methods. Further nodes are inserted to compute the CPU cycles taken directly by this compilation unit, and to add the result to the unit's entry in a global array for all compilation units. For this, we use the previously prepared memory address, which minimizes the run-time computations. The array is processed right before the JVM shuts down, to compute and output the overall profile.

Since we instrument code in the JIT compiler, only methods that are compiled will collect profiling information. Methods that are interpreted only thus will not be profiled. However, for many use cases, the methods relevant for performance are invoked often and therefore get JIT compiled.

4.2 Attributing Cycles to Inlined Methods

As discussed in Section 2.4, inlining is a major challenge for profilers when it comes to correctly attributing where a program spends its time, because inlining and subsequent compiler optimizations may cause an inlined method to be arbitrarily intermixed with parts from other methods.

To attribute time to inlined methods after all optimizations, we estimate the cycle cost for the remaining elements. We know for each GraalIR node from which method it originates. Based on branch probabilities and loop counts collected at run time, we then estimate which fraction of the overall compilation unit comes from a specific method. This allows us to identify which methods make up the hottest compilation units and avoids the overhead of instrumenting each remaining part of an inlined method separately.

4.3 Evaluation

To evaluate our late-compiler-phase-based instrumentation, we compare it with sampling- and the other instrumentation-based profilers. The setup is the same as in Section 3.1, which gives both instrumentation-based profilers and samplers enough time to profile the program in a stable state. However, some minor engineering issues prevented us from using libgraal for our implementation, which we call Bubo. Thus, all experiments with Bubo use *jargraal*, i.e., the Java version of Graal that is subject to JIT compilation itself. We believe this has no major impact on our results. In the worst case, it disadvantages Bubo compared to other profilers.

Comparison with Classic Instrumentation. As shown in Figure 2, the median overhead for the instrumentation profilers is in the range of 82× to 374× over all benchmarks. This means at the median, JProfiler causes programs to take 82× more time compared to their uninstrumented version.

In contrast, Figure 6 shows that Bubo has a median overhead of only 23.3% (min. 1.4%, max. 464%), and having a generally lower impact on the execution of a program.

Comparison with Sampling. Sampling profilers are expected to have a lower overhead than instrumentation-based profilers, since their overhead is proportional to the sampling frequency, instead of incurring a constant overhead for every instrumented method. Figure 6 shows that Bubo has a higher

median overhead with 23.3% (min. 1.4%, max. 464%) than the sampling profilers. Bubo's median overhead of 23.3% is also higher than the 75th percentile of the sampling profilers, as indicated by the right edge of the boxes. Nonetheless, Bubo's overhead is much closer to that of sampling-based profilers than that of instrumentation-based ones.

We believe these first results show that the approach could make instrumentation-based profiling more practical and realistic.

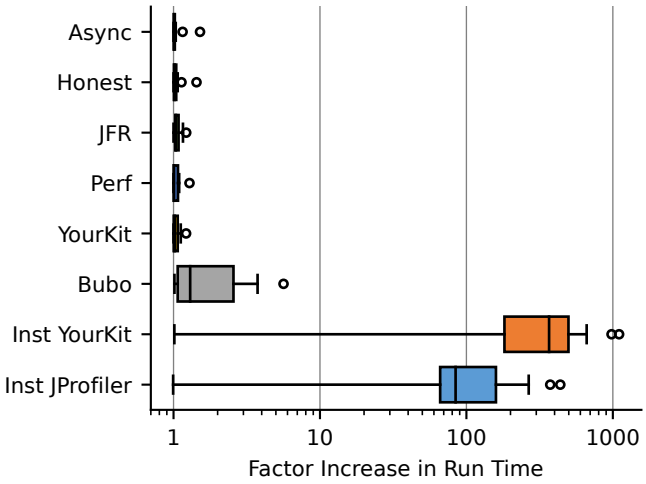

Figure 6. Run-time overhead for each profiler expressed as factor over the uninstrumented run time. Async has the lowest median overhead with 1% (min. 0.1%, max. 52%). Bubo's median overhead is 23.3% (min. 1.4%, max. 464%).

5 Related Work

The most relevant work is on instrumentation at the compiler level and binary rewriting.

Most notably, Zheng et al. [21] and Basso et al. [2] inspired our late-compiler-phase instrumentation. Zheng et al. [21] illustrate the impact of JIT compiler optimizations on, for instance, object allocations and method invocations, demonstrating the need to work alongside the JIT compiler to understand which of these operations are still present after optimization. Both [2, 21] focused on specific compiler optimizations and compiler events to better understand the compiler and performance issues with it. We on the other hand focus on profiling applications.

Much earlier work such as gprof [7], also instrumented programs as part of ahead-of-time (AOT) compilation. Today's compilers such as GCC and LLVM also support it, e.g., to enable profile-guided optimizations [16].

However, it seems more common today for application profiling to instrument binaries after compilation to avoid interfering with optimizations. Dynamic binary instrumentation can be used to make instrumentation very targeted for use in production, e.g., by sampling programs using instrumentation at configurable frequencies set by the user [11]. Others optimize instrumentation by combining multiple probes into one to reduce the overhead without losing information [9] or by using self-modifying instrumentation [17].

6 Conclusion

In this work, we show that state-of-the-art instrumentation-based profilers for the JVM have high overhead and report unrealistic results. We found that the lowest median overhead is 82× across the Are We Fast Yet benchmarks. The overhead can be explained with the cost of instrumentation, which is visible in the increased bytecode size, native code size, and amount of inlining. We further argue that the reported profiles are unrealistic, because they do not change when inlining is turned off, which indicates that the instrumentation negates most compiler optimizations.

To overcome these issues, we proposed late-compiler-phase-based instrumentation. It minimizes interference with compiler optimizations and as a result delivers more realistic profiles than other instrumentation-based profilers.

In our prototype implementation, it reduces the median profiler overhead on the Are We Fast Yet benchmarks to 23.3% (min. 1.4%, max. 464%), which is more similar to sampling-based profilers. Furthermore, we attribute the cycles for a compilation unit to the methods fragments that remain in the compilation unit after optimization to account for inlining. However, our prototype does not support multithreading.

Future Work. The main benefit of instrumentation-based profiling over sampling is its precision, i.e., that it gives consistent results (see Section 2.2). However, it's not generally possible to assess the accuracy of profilers and samplers often disagree with eachother [3, 14]. Thus, an important open question is how to better approximate the ground truth profile for any given program. One could possibly use hardware simulators to determine the ground truth and thereby assess the accuracy of profilers [6].

One could also consider combining sampling and late-compiler-phase instrumentation to reduce the overhead, and gain precision for specific parts of a profile. A hybrid solution would also allow profiling of executions in the interpreter.

Other future work is more engineering focused. At this point, Bubo instruments all methods, but perhaps one would want to select manually which methods to instrument as in classic instrumentation-based profilers. Adding support for multithreaded application and ensuring the profile data is collected correctly would be needed to make Bubo work with most JVM applications.

Acknowledgments

This work was supported by a grant (EP/V007165/1) and studentship (2619006) from EPRSC as well as a Royal Society Industry Fellowship (INF\R1\211001). The authors would also like to thank Matteo Basso and the GraalVM community for their advice and discussions.

References

[1] Ole Agesen. 1998. *GC Points in a Threaded Environment.* Technical Report SMLI TR-98-70. Sun Microsystems.

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

[3] Humphrey Burchell, Octave Larose, Sophie Kaleba, and Stefan Marr. 2023. Don't Trust Your Profiler: An Empirical Study on the Precision and Accuracy of Java Profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2023)*. ACM, 100–113. https://doi.org/10.1145/3617651.3622985

[4] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. In *IR '95: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, United States). ACM, 35–49. https://doi.org/10.1145/202529.202534

[5] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[6] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. ACM, 15–27. https://doi.org/10.1145/3466752.3480058

[7] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. gprof: a Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. ACM, 120–126. https://doi.org/10.1145/800230.806987

[8] Berkin Ilbeyi and C. Batten. 2016. JIT-assisted fast-forward embedding and instrumentation to enable fast, accurate, and agile simulation. *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2016), 284–295. https://doi.org/10.1109/ISPASS.2016.7482103

[9] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. 2005. Low overhead program monitoring and profiling. *SIGSOFT Softw. Eng. Notes* 31, 1 (sep 2005), 28–34. https://doi.org/10.1145/1108768.1108801

[10] Elena Machkasova, Kevin Arhelger, and Fernando Trinciante. 2009. The observer effect of profiling on dynamic Java optimizations. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 757–758. https://doi.org/10.1145/1639950.1640000

[11] Scott Mahlke, Tipp Moseley, Richard Hank, Derek Bruening, and Hyoun Kyu Cho. 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494982

[12] Stefan Marr. 2023. *ReBench: Execute and Document Benchmarks Reproducibly.* https://doi.org/10.5281/zenodo.8219119 Version 1.2.0.

[13] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[14] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, 187–197. https://doi.org/10.1145/1806596.1806618

[15] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. 2007. JIT Instrumentation - A Novel Approach To Dynamically Instrument Operating Systems. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. ACM, 3–16. https://doi.org/10.1145/1272996.1273000

[16] Adam Preuss. 2010. *Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure.* Technical Report TRID-ID TR10-05. 1–16 pages. https://doi.org/10.7939/R3GF0MX64

[17] Amitabha Roy, Steven Hand, and Tim Harris. 2011. Hybrid Binary Rewriting for Memory Access Instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (Newport Beach, California, USA) *(VEE '11)*. ACM, 227–238. https://doi.org/10.1145/1952682.1952711

[18] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, 196–205. https://doi.org/10.1145/178243.178260

[19] Maja Vukasovic and Aleksandar Prokopec. 2023. Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code. *ACM Transactions on Programming Languages and Systems* 45 (2023), 1–64. https://doi.org/10.1145/3612937

[20] April W. Wade, P. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: impact of profile data on code quality. *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (2017). https://doi.org/10.1145/3078633.3081037

[21] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate Profiling in the Presence of Dynamic Compilation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, 433–450. https://doi.org/10.1145/2814270.2814281

Received 2024-05-25; accepted 2024-06-24

Table 2. A complete version of Table 1. Comparison of methods percentages across different profilers for the DeltaBlue benchmark with and without inlining.

Profiler	Inlining	Method	Percentage
Async	yes	deltablue.Plan	23
		Vector.forEach	17
		ScaleConstraint.execute	4
		Vector.append	3
		ScaleConstraint.recalculate	3
	no	EqualityConstraint.execute	19
		ScaleConstraint.execute	8
		invoke.DirectMethodHandle$Holder.newInvokeSpecial	4
		Plan$$Lambda.0x00007fcf5800d6c0.apply	3
		Variable.getValue	3
JProfiler	yes	EqualityConstraint.execute	14
		Plan.lambda$execute$0	10
		Vector.forEach	9
		Variable.getValue	6
		ScaleConstraint.execute	5
	no	EqualityConstraint.execute	14
		Plan.lambda$execute$0	10
		Vector.forEach	8
		Variable.getValue	6
		ScaleConstraint.execute	5
VisualVM	yes	Plan.lambda$execute$0	14
		Vector.forEach	12
		Variable.getValue	6
		Planner.addPropagate	4
		AbstractConstraint.satisfy	2
	no	Plan.lambda$execute$0	24
		Vector.forEach	12
		Variable.getValue	6
		Planner.addPropagate	4
		AbstractConstraint.satisfy	2
YourKit	yes	deltablue.Plan.lambda$execute$0	24
		som.Vector.forEach	17
		deltablue.EqualityConstraint.execute	10
		deltablue.Planner.addPropagate	3
		deltablue.AbstractConstraint.satisfy	2
	no	deltablue.Plan.lambda$execute$0	36
		som.Vector.forEach	17
		deltablue.EqualityConstraint.execute	10
		deltablue.Planner.addPropagate	4
		deltablue.AbstractConstraint.satisfy	2