Don't Trust Your Profiler

An Empirical Study on the Precision and Accuracy of Java Profilers

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

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

Sophie Kaleba
University of Kent
United Kingdom
s.kaleba@kent.ac.uk

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

Abstract

To identify optimisation opportunities, Java developers often use sampling profilers that attribute a percentage of run time to the methods of a program. Even so these profilers use sampling, are probabilistic in nature, and may suffer for instance from safepoint bias, they are normally considered to be relatively reliable. However, unreliable or inaccurate profiles may misdirect developers in their quest to resolve performance issues by not correctly identifying the program parts that would benefit most from optimisations.

With the wider adoption of profilers such as async-profiler and Honest Profiler, which are designed to avoid the safepoint bias, we wanted to investigate how precise and accurate Java sampling profilers are today. We investigate the precision, reliability, accuracy, and overhead of async-profiler, Honest Profiler, Java Flight Recorder, JProfiler, perf, and YourKit, which are all actively maintained. We assess them on the fully deterministic Are We Fast Yet benchmarks to have a stable foundation for the probabilistic profilers.

We find that profilers are relatively reliable over 30 runs and normally report the same hottest method. Unfortunately, this is not true for all benchmarks, which suggests their reliability may be application-specific. Different profilers also report different methods as hottest and cannot reliably agree on the set of top 5 hottest methods. On the positive side, the average run time overhead is in the range of 1% to 5.4% for the different profilers.

Future work should investigate how results can become more reliable, perhaps by reducing the observer effect of profilers by using optimisation decisions of unprofiled runs or by developing a principled approach of combining multiple profiles that explore different dynamic optimisations.

CCS Concepts: • **General and reference → Measurement**; • **Software and its engineering → Interpreters**; *Just-in-time compilers.*

Keywords: Profiling, CPU sampling, Profiler comparison, Analysis tools, Profiler precision

ACM Reference Format:
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 '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3617651.3622985

1 Introduction

There are many tools to analyse the efficiency of a program, with run-time profiling perhaps being the oldest one, having existed for over 50 years [9]. Profilers typically measure how a program uses resources such as CPU, GPU, and memory. Software developers use this information to identify performance bottlenecks and optimisation opportunities.

Unfortunately, modern computer systems are highly complex. Many systems rely on feedback-driven dynamic optimisations, caching, and possibly predictive optimisations. Languages such as Self, Java, JavaScript, Ruby, and PHP, use just-in-time (JIT) compilers, which compile and optimise a program based on run-time feedback.

Given this complexity, back in 2010 Mytkowicz et al. [16] evaluated the accuracy of sampling Java profilers. One of the many things they showed is that four Java profilers disagreed on which method took the highest percentage of a program's run time. Given that this is typically one of the key questions developers want to be answered when using profilers, this result called the reliability of profilers into question.

Sampling profilers encounter many challenges on modern computer systems. The sampling itself relies on the assumption that every point in the program has the same likelihood of being observed by the profiler. Thus, sufficient samples will give a probabilistic view of where programs spend their time. Unfortunately, in language virtual machines as used by Java, safepoints [1], which are sometimes also called yield points [2], are needed to enable for instance garbage collection in JIT-compiled systems, yet can bias what points in a program are visible to a profiler [16], because sampling might only be possible at these safepoints. Safepoints and optimisations may also lead to profilers incorrectly assigning percentages of the run time to wrong parts of the program, perhaps because the profiler did not have the necessary information to identify which source code a section of highly optimised native code originates from.

Given all these issues, and the wider adoption of profilers such as async-profiler and Honest Profiler that are designed to avoid the safepoint bias, we wanted to know how precise and accurate today's profilers are, 13 years after Mytkowicz et al.'s work.

With the probabilistic nature of sampling-based profiling, we are interested in the accuracy and precision of the profiles that profilers produce. To have a good intuition of what an accurate profile should look like, we chose the Are We Fast Yet benchmarks [14]. Since they are relatively small, designed to be fully deterministic, and contain many classic benchmarks, it is easier to judge profiles than it would be the case for larger benchmarks as in DaCapo [4] or Renaissance [18].

With our experiments, we assess whether Java profilers still disagree with each other when identifying the parts where a program spends most of its time. While the data of Mytkowicz et al. [16] suggests that a profiler typically agrees with itself, we verified this, too. Furthermore, we assess the run-time overhead on the execution.

We find that profilers identify the same method as hottest relatively reliably when looking at them in isolation. Though, on two benchmarks this was not the case, suggesting that the precision of profilers might be application-specific. Furthermore, the run time attributed to a method may vary, for YourKit the median difference is quite large with 8%.

When comparing the profiles among profilers, we unfortunately find that they often disagree with each other. We found this effect for the hottest method as well as the set of the five hottest methods.

With this disagreement among profilers, it is not clear how directly actionable the results of a single profiler are. Relying on a single profile may misdirect optimisation efforts and not yield the expected results. However, async-profiler, Java Flight Recorder and Honest Profiler provider very similar profiles for all but two benchmarks. We discuss this in detail in Section 5.2.

The overhead of profiling is on average 1% to 5.41% depending on the profiler.

The contribution of this paper is an evaluation of the precision, accuracy, and overhead of six actively maintained Java profilers, both commercial and open source.

2 Background

In the following section, we give a wider background on profiling and then discuss the known issues with sampling profilers.

2.1 Profilers and Profiling Techniques

Code profiling is a form of dynamic analysis, which enables developers to measure various aspects of a program's execution. What is measured depends on the profiler and the intention of the user. It can include CPU and memory usage, or other specific system resources and hardware counters. The most common code profiling techniques are code instrumentation and CPU sampling.

The motivation behind utilizing a profiler is often to find code segments that can be optimised. Processor vendors such as AMD have developed profilers for their various products including AMD RGP[1] for AMD GPUs and AMD uProf [2] for CPUs, while Intel offers VTune.[3] The many other profilers include CodeGuru[4] from Amazon, which makes use of machine learning to help improve applications running Amazon services, and AppDynamics owned by Cisco for IT infrastructure monitoring. Different programming languages typically have their own specific profilers, too. Examples include ARMmap[5] for C and C++, JProfiler[6] and VisualVM[7] for Java and dotTrace[8] for .NET.

2.1.1 Instrumentation. Instrumentation-based profilers generally add probes to a program. Such probes can take the form of code instructions at the source, bytecode, or native code level, which record the desired information. A classic example is to insert a counter at the start of every method, which increases whenever its associated method is called, allowing developers to identify possible performance bugs. Srivastava and Eustace [22] proposed a platform called ATOM (Analysis Tools with OM), which inserts code into a program at compilation time, with this inserted code gathering program attributes and making them available for program analysis. Instrumentation can be added automatically [15] or manually [19], and it can be performed either at compilation time [5] or at run time [12].

2.1.2 CPU Sampling. CPU sampling interrupts the CPU to gather a snapshot of the current state of hardware and given software on the machine, for instance recording the call stack, instruction pointer, memory usage, and thread

[1]https://gpuopen.com/rgp/
[2]https://www.amd.com/en/developer/uprof.html
[3]https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[4]https://aws.amazon.com/codeguru/
[5]https://docs.linaroforge.com/22.1.3/userguide-forge.pdf
[6]https://www.ej-technologies.com/products/jprofiler/overview.html
[7]https://visualvm.github.io/
[8]https://www.jetbrains.com/profiler/

activities. These interrupts are typically done at regular intervals, but sometimes at random intervals [16], usually many times during the execution of a program to build an accurate profile.

In the 1970s, the IBM S/360 [9] used program status words, which included the program counters to enable an early form of sampling. In timer-based interrupts, the developer wants to see what the CPU was doing at each interruption. In 1974 UNIX introduced PROF [10], one of the first tools for performance analysis. It can output how much time each individual function took to execute.

2.2 Problems with Profilers

While profilers are widely used in both industry and research, they have some well-documented issues.

2.2.1 Overhead. All instrumenting profilers induce some overhead in the execution of the program [11]. Sometimes this overhead is minimal, but often it increases the run times by several factors. The amount of overhead varies depending on the profiler itself as well as the code profiling technique it uses. This overhead is problematic, for example when profiling a program which takes several hours to run, as a high overhead can then increase this total run time to several days. Thus, sampling-based profilers are often preferred since overhead is limited to the points when a program is interrupted for profiling.

Low overhead is especially desirable when profiling systems in production, since it can negatively affect user experience and excessive overhead can make profiling of production systems impractical. Thus, we assess the overhead Java profilers add to a program's execution time in Section 4.3.

2.2.2 Observer Effect. Profiling can change how a program executes [20], which is also called the probe effect [6]. This may change or bias the results that a profiler reports. For example, any code instrumentation intrinsically alters program behaviour and may therefore change execution times. Furthermore, the instrumentation code can also change how a JIT compiler optimises a program [23]. Similarly, CPU sampling with interrupts may also change the performance of program, for instance by changing CPU caches or interacting with multi-threaded executions [17]. In the worst case, this may lead to attributing execution time to parts of the program which get minimal benefits from developers trying to optimise them.

2.2.3 Sampling Randomness and Safepoint Bias. Safepoints [1], sometimes called yield points [2], are needed to enable for instance garbage collection, and guarantee that the system is in a known state, where it is safe to interrupt the thread. Interrupting at safepoints limits the observer effect and ensure that the VM is in a state in which a stack can be read correctly, samples are typically collected at these safepoints. Unfortunately, this can lead to inaccurate profiles, as shown by Mytkowicz et al. [16].

The compiler moves and eliminates safepoints to minimize the overhead they cause, which can result in sections of hot code not being fully sampled, if there are few or no safepoints in that section. This results in sampling profilers not sampling at uniform intervals, but instead sampling only a reduced part of the program, resulting in biased samples. This leads to profilers incorrectly assigning percentages of the run time to the wrong parts of a program.

2.2.4 Reducing Safepoint Bias. Several of the profilers we are testing aim to reduce the impact of safepoint bias.

Honest Profiler and async-profiler aim to minimise bias by using `AsyncGetCallTrace`, an OpenJDK internal function, which allows them to record stack traces without requiring a thread to be in a safepoint. However, to be able to correctly interpret the obtained data for the top stack frames, they may need to rely on nearby safepoints.[9] Therefore, this approach does not fully eliminate safepoint bias, but it reduces it.

The async-profiler additionally collects stack traces with `perf_events`. By combining them with the stack traces collected with `AsyncGetCallTrace`, it can get profiling information for the Java code as well as any native code, i.e., in the VM implementation or any of the used native libraries.

Java Flight Recorder uses its own code, independent of `AsyncGetCallTrace`, to access the stack traces without requiring safepoints, but has similar limitations as the other two profilers. Since the VM assumes that a thread's stack is only accessed in a safepoint, it does not normally produce metadata outside of these safepoints to safe memory. The HotSpot JVM supports the `-XX:+DebugNonSafepoints` flag to ensures that metadata is available also when execution is not in a safepoint, which can help with identifying the correct line number or bytecode index for a stack sample.

Overall, this means there is only a limited amount of information available for the top most stack frame, which limits tools in identifying the specific bit of code executed. Thus, it remains a limited safepoint bias for sampling profilers.

2.2.5 Non-determinism in JVMs. Multiple runs of the same Java program on the same JVM may show different performance. This is because JVMs perform just-in-time compilation in parallel to program execution. For example, this can cause variation when the compiler accesses profile information. Access to profile information is designed so that multiple threads can read and write it without requiring synchronization to avoid overhead for application threads. However, it may lead to different optimization decisions, which may result in different run times between executions.

[9] *Why JVM modern profilers are still safepoint biased?*, Jean-Philippe Bempel, https://jpbempel.github.io/2022/06/22/debug-non-safepoints.html

3 Research Questions and Methodology

This section details our research questions and describes our experimental setup, metrics, the benchmarks, and how we collect data to evaluate each profiler.

3.1 Research Questions

To focus our inquiry, we formulate the following research questions.

RQ1: Do sampling profilers give precise results? As argued in Section 2.2.3, sampling profilers are inherently probabilistic and face many challenges, which may bias and distort results. Thus, our first question to investigate is whether today's profilers are accurate, i.e. give correct results, and precise, i.e. whether they agree with each other. Given the complex systems we are working with, assessing accuracy is not directly possible: even when crafting workloads where we have an expectation of where time is going to be spent, with dynamic optimisations in soft- and hardware, we cannot precisely quantify this. Instead, we focus on whether the results are precise. This means, we expect measurements to be tightly grouped.

To this end, we assess the percentage of run time that a profiler attributes to various methods, and compare it with percentages reported by the same profiler across 30 runs. We also compare these reported values with results reported across the different profilers.

RQ2: Do sampling profilers reliably identify the same hottest methods in an execution? Related to precision, but more focused on whether the results are actionable for developers, we will assess whether the profilers correctly identify the same hottest methods. Often, a developer may only need a ranking of the methods to effectively focus their attention. The precise run time a method takes may only be of secondary importance. Thus, here we investigate the method rankings that profilers create based on where time is spent and assess their consistency between different profiler runs and across profilers.

RQ3: How high is the run-time overhead of sampling profilers? Since the overhead of profiling may interfere with program execution and high overhead may make profilers impractical for certain use cases, we also assess their impact on the overall run time of a benchmark.

To assess the overhead, we run the benchmarks both with and without profilers attached, and compare the overall run time between both sets of runs.

Since all experiments were run 30 times with 300 iterations each, we compute the run time by taking the median over the $30 \cdot 300$ data points. To avoid our numbers being skewed by the warmup phase, we rely on the median run time to effectively characterize run-time overhead.

3.2 Experimental Setup

For our experiments, we chose six different profilers, evaluate them on the Are We Fast Yet benchmarks [14], and perform the following high-level steps for each profile:

1. We run each of the 14 benchmarks 30 times for 300 iterations, with no profiler attached, to determine the baseline performance.
2. We then run all 14 benchmarks 30 times with a profiler attached for the 300 iterations to record profiles.
3. From the recorded profiles, we extract the 10 hottest methods, the percentage of the total execution run time spent in each method, and the total run time.

We run everything 30 times to compensate for the probabilistic nature of sampling profilers, the aforementioned non-determinism of JVMs (see Section 2.2.5), and other external influences. We expect that 30 runs are sufficient for a reliable profile to emerge.

We discuss the details of this setup in the remainder of this section.

3.3 The Chosen Profilers

We rely on the following profilers:

- async-profiler[10] is an open source project. It aims for low-overhead sampling without safepoint bias.
- Honest Profiler[11] is an open source project. It also aims for low-overhead sampling without safepoint bias.
- Java Flight Recorder[12] is part of the JDK and allows the collection of run time statistics and includes a sampling-based profiler.
- JProfiler[13] is a commercial tool and supports CPU sampling and instrumentation.
- perf[14] is a Linux tool which also supports sample-based profiling of Java applications.
- YourKit[15] is a commercial tool that aims to support profiling with low overhead.

For our experiments, we chose six actively maintained profilers. This includes the two commercial products JProfiler and YourKit, as well as four open source profilers either specifically designed for Java, or in the case of perf, with Java support. We included async-profiler and Honest Profiler, since they are specifically designed to avoid the issue of safepoint bias as identified by Mytkowicz et al. [16]. Since the Java Flight Recorder is directly integrated in the Java Development Kit (JDK) and the HotSpot JVM, we also include it in our investigation. We considered to also include Xprof and Hprof since they were used by Mytkowicz et al. [16], but they are unfortunately no longer actively maintained.

3.4 Benchmarks and Experimental Setup

As previously mentioned, we evaluate the profilers on the Are We Fast Yet benchmarks [14]. We choose them since they contain relatively well-understood classic benchmarks, which encapsulate a variety of common program behaviours, with 9 microbenchmarks and 4 slightly larger benchmarks.

Table 1 provides a brief overview of key metrics. Most relevant for our experiments is the last column with *hot* methods, i.e., methods that are executed in each iteration, may eventually be JIT compiled, and are candidates for being included in a profile. Given the size of our benchmarks, it is also possible for us to inspect the code and compare obtained profiles with our own intuition.

We also chose these benchmarks because they are designed to be fully deterministic. Thus, they show identical behaviour across iterations by avoiding using for instance Java's `HashMap`, which can rely on object hashes that are not the same across multiple program executions. Using benchmarks with fully deterministic behaviour gives us a more reliable foundation for assessing sampling profilers.

However, choosing these benchmarks also means they are less representative of large Java applications. While benchmark suites such as DaCapo [4] and Renaissance [18] would be more representative, we aimed to give the profilers a workload that can be considered a best-case scenario for them. Thus, the benchmarks are pure single-threaded Java and do not rely on for instance the Java Native Interface or other features that could be a challenge for the profilers.

Table 1. Basic metrics for the Are We Fast Yet Benchmarks

Benchmark	Executed Lines	Classes	Executed Methods	Hot Methods
CD	356	16	43	41
DeltaBlue	387	20	99	75
Havlak	421	18	110	87
Json	232	14	56	56
Richards	279	12	47	47
Bounce	42	5	11	11
List	30	2	9	9
Mandelbrot	39	0	2	2
NBody	105	3	14	14
Permute	33	3	13	13
Queens	36	3	13	13
Sieve	22	3	9	9
Storage	23	4	10	10
Towers	42	2	12	12

To execute the benchmarks with and without profilers, we rely on ReBench [13]. This tool automates the benchmark execution and documents the configuration of the profilers to facilitate reproducibility.

All benchmarks were executed on a machine with Rocky Linux 9, and a Linux kernel version 5.14.0. The machine has a AMD Ryzen 5 36000 6-Core processor and 32 GB DDR4 RAM. We use the Java HotspotVM 17.0.7.

To give the profilers the best possible chance for accurate profiles, we run each benchmark for 300 iterations. None of the initial arguments set will change during the iterations. Since all benchmarks reach peak performance early on, this gives the profilers a lot of iterations to determine an accurate profile. Furthermore, this should minimise external factors, such as noise caused by interference from the operating system. All profilers are configured to take a sample every 10ms. While such a high sampling rate may incur higher overhead, it should also ensure that the profilers have a good chance to get a detailed picture of the benchmark execution. To put this sampling rate into perspective, the median run time spent executing an iteration across our benchmarks is 102ms. The minimum is 92ms, and the maximum 152ms. Therefore, we expect samples to happen between 9 and 15 times per iteration, with between 2,700 and 4,500 samples per benchmark execution.

With all these settings, a complete run of all experiments takes about 30 hours to complete.

The profilers typically create output files, which we process to collect all needed information, including the 10 hottest methods, total run time, and the percentage of run time each method took.

Since output formats and details differ widely between profilers, the data is first converted into a uniform JSON representationand then method names are normalised to enable easy comparison.

4 Results

In this section, we analyse the obtained data and answer our research questions. We analyse a profiler first by comparing it to itself, to determine how precise and reliable its results are over multiple runs. Afterwards, we examine how reliable profilers are in identifying the hottest methods by comparing them to each other. Finally, we assess the overhead of each profiler.

4.1 Self-comparison

We first assess a profiler's precision and reliability.

4.1.1 Precision. To assess how precise a profiler is in determining the percentage of run time for each method in a benchmark, we take the difference between the maximum and minimum reported run time percentage. This gives us an idea of how much a profiler disagrees with itself. Since sampling profilers are probabilistic by nature, we limit the analysis to methods that contribute to at least 5% of their average run time to the program. This minimises the noise

[10] https://github.com/async-profiler/async-profiler
[11] https://github.com/jvm-profiling-tools/honest-profiler
[12] https://openjdk.org/jeps/328
[13] https://www.ej-technologies.com/products/jprofiler/overview.html
[14] https://perf.wiki.kernel.org/index.php/Main_Page
[15] https://www.yourkit.com/

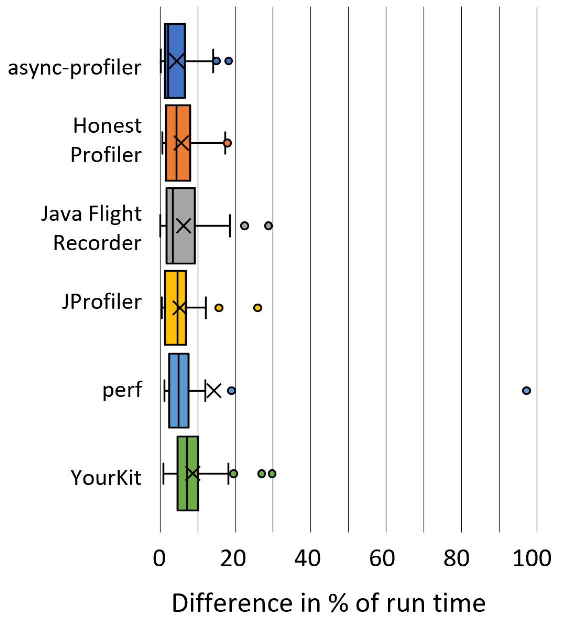

Figure 1. Difference in % of run time for every method, for each profiler. The boxplot shows the difference between the minimum and maximum percentage of run time for each method. The × indicates the mean. The median difference is between 2 and 8%, suggesting a high precision.

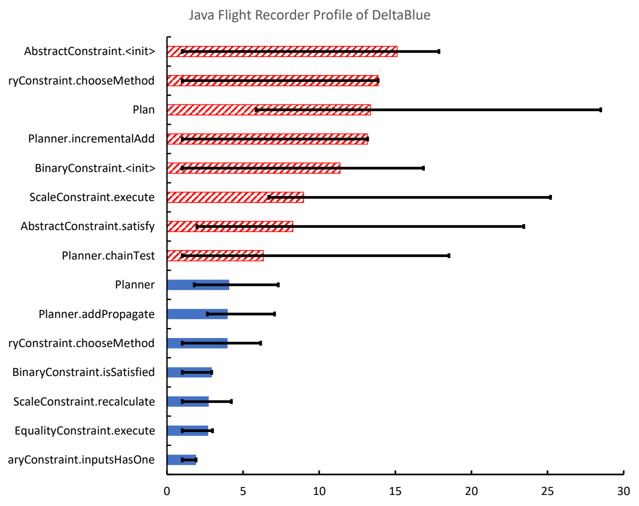

Figure 2. Bar chart for all methods in the DeltaBlue benchmark identified by Java Flight Recorder. A bar represents the average percentage of run time for a method, with error bars being the minimum and maximum value found for that method. Striped bars are methods that were the hottest method at least once. The large error bars representing the minimum/maximum shows that methods can easily switch places in the ranking.

in the results and likely represents what developers are interested in when trying to optimise a program.

Figure 1 shows the distribution of differences per profiler. For instance, a value of 10% means the difference between the minimum and the maximum run time is 10%.

For all 6 profilers, the median ranges between 2% and 8%. This suggests that they have relatively high precision for most methods. However, the maximum differences found for each profiler are about 18% for async-profiler, 18% for Honest Profiler, 29% for Java Flight Recorder, 26% for JProfiler, 99% for perf, and 32% for YourKit. These high differences indicate that each profiler has a rather low precision for at least some methods or benchmarks.

DeltaBlue and Richards were the two benchmarks on which profilers struggled to get good precision. Additionally, perf has a group of 4 outliers that show a difference of 99% on the NBody and Permute benchmarks. perf attributed drastically different run times to two methods between runs. However, this effect is unique to perf, as the other 5 profilers have good precision for the NBody and Permute benchmarks.

To investigate the impact of the difference, we created plots for each benchmark showing the average run time attributed to a method. Figure 2 shows a worst-case example with the DeltaBlue benchmark using Java Flight Recorder. The error bars show the minimum and maximum values. The striped red bars identify the methods that were at least once reported as the hottest method in one of the 30 profile runs, which was the case for 10 different methods on this benchmark. In addition to the large minimum/maximum range indicated for the methods, this suggests that we can not fully rely on profiles from Java Flight Recorder on DeltaBlue to guide optimisation decisions.

As a second example in Figure 3, we chose one benchmark on a profiler with a median amount of imprecision. It shows the aggregate over the 30 profiles for the CD benchmark from YourKit. It being a median case, there is only one method that was identified as the hottest in all 30 runs, but both `RedBlackTree.treeInsert` and `CollisionDetector.recurse` have a large minimum/maximum range, which means they can easily switch places in the ranking. Methods 3–5 could also appear in different orders based on their minimum/maximum range.

These large differences suggest that a single profile run even on a benchmark with good precision will still leave doubt on where a program spends its time.

As an example with little self-imprecision, we saw the NBody benchmark on the Honest Profiler. Here the maximum difference seen for the 10 hottest methods was at most 3% and no method had enough difference to overlap with another method.

The results presented here indicate that profile runs can significantly differ from one to the next. In the worst case,

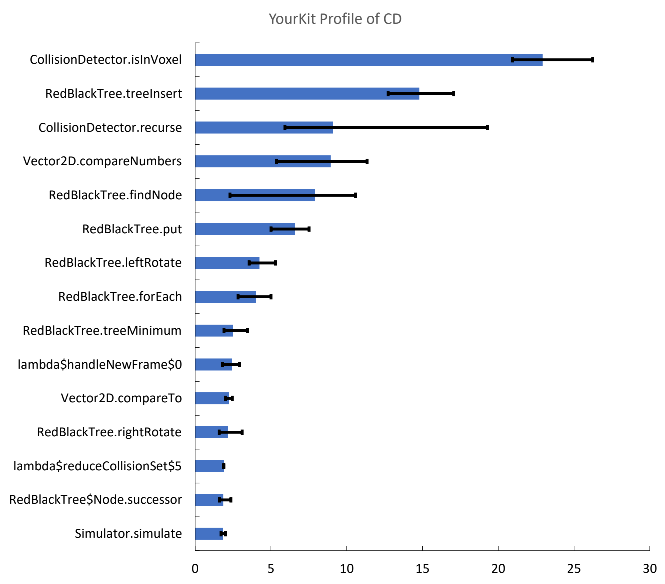

Figure 3. Bar chart for all methods in the CD benchmark identified by YourKit. Bar represents the average percentage of run time for a method, error bars are the minimum and maximum value found for that method. The large range between minimum/maximum shows that runs can be significantly different from one another.

this makes the profiles highly unreliable, but even in the median case, it suggests that one would want to collect at least a second profile if the precise ordering of hottest methods is of interest.

To answer our RQ1, of whether sampling profilers give precise results, it seems this depends on the particular application in question. If the precision matters, for instance to prioritise which areas of a program to investigate for optimisations, it seems necessary to collect multiple profiles to assess the precision on a case-by-case basis.

4.1.2 Reliability. To get a more complete overall impression of how reliable profilers identify the same hottest method among their 30 profiles, we look at the statistics in Table 2.

The *Hottest methods* columns give the median and maximum number of different methods reported as hottest. A value of one means that across all 30 profiler runs, the same method was identified as hottest. Since the numbers aggregate across all benchmarks, the median reliability to identify the hottest method seems to be given. The maximum number of methods identified as hottest suggests that JProfiler and YourKit are having the least issues with imprecision, identifying at most two methods as hottest.

While the hottest methods are typically of most interest, we also had a look at the full set of the top 10 methods and how stable it is across the 30 runs. The column *Unstable Methods* counts the methods that do not appear in the top 10 for all 30 runs. To reduce the noise of methods that contribute

Table 2. Identification of the first hottest method in a given run, for each profiler. We show the median number of such methods for each benchmark, as well as the maximum number of first hottest methods observed. *Unstable Methods* refers to the number of methods that do not appear in the top 10 hottest for all 30 runs, but appear in it for at least one run.

	Hottest Methods		
Profiler	Median	Max	Unstable Methods
async-profiler	1	6	35
Honest Profiler	1	9	38
Java Flight Recorder	1	10	43
JProfiler	1	2	16
perf	2	4	16
YourKit	1	2	74

minimally to the overall run time, we only consider methods that contribute at least 1% of the total run time on average. An *Unstable Methods* count of 0 would mean the profiler had the same top 10 methods over all 30 runs for all 14 benchmarks. A higher value means higher unreliability.

To summarise, with a median number of one identified first hottest methods for all but perf, the profilers identify the hottest methods reliably between runs. However, the high maximum number of the first hottest methods is still concerning and indicates that async-profiler, Honest Profiler, Java Flight Recorder, and perf can be unreliable for certain benchmarks. Additionally, the *Unstable Methods* count reveals a poor reliability of the profilers. For instance, YourKit sees 74 methods that are not consistently in the top 10. Considering we discard from this metric the methods that contribute less than 1% of the runtime, we argue that the profilers are unreliable at identifying the same set of hot methods.

An example of change in the first hottest method happening from profiler to profiler can be seen with Figure 4.

Each method plotted was identified as the hottest method at least one in all of the 30 runs. Figure 4 plots the percentage of run time over the median run time of iteration for that profiler run. The method with the highest amount of percent of the run time (the furthest up on the y-axis) was identified to be the hottest method for that iteration. Every method that is plotted was the hottest method at least once, which correlates with the results showed in Table 2, with DeltaBlue being the benchmark with the maximum number of methods identified as hottest for async-profiler. Methods like `som.Vector.forEach` can, during one iteration, be reported to take up 28% of run time, yet on another iteration take up only 8% of run time.

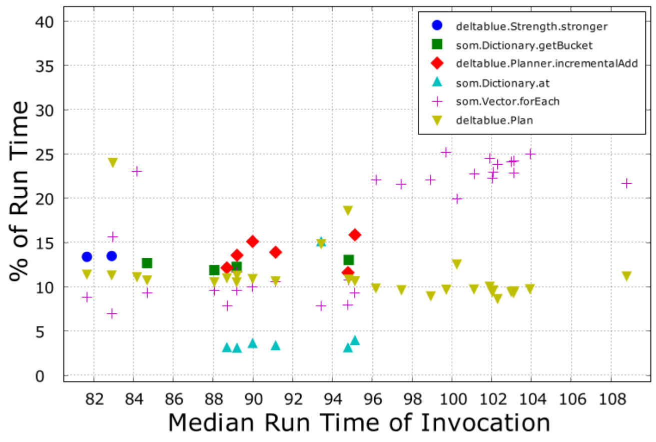

Figure 4. Percentage of run time over median run time for the six hottest methods for the DeltaBlue benchmark profile from async-profiler. We observe multiple changes in what the first hottest method is (the furthest up) for each iteration.

4.2 Across Comparison

Even if we have identified a profiler as precise and reliable so far, the methods it identifies could be incorrect. To assess the accuracy, we now compare the identified methods between profilers to see if there is disagreement.

4.2.1 Average Comparison. Looking back to Figure 1, there are some observations that can be made in comparing the profilers against each other.

YourKit has the highest average difference of 7.1% and async-profiler has the lowest average difference of 2.1%. This indicates a large difference in precision between these two profilers. The most likely explanation for this difference is that they use different sampling techniques. Unfortunately, YourKit is closed-source. Thus, we were unable to investigate this theory further.

For the maximum difference, perf has the previously mentioned outliers in the 99% range. No other profiler exhibited such extreme outliers. We assume this might be caused by some issue in perf's Java support.

4.2.2 Unions. Mytkowicz et al. [16] compared the cardinality of the union of the first hottest method over all profiles, which means they counted how many different methods they see. We will use their approach, too. As mentioned in Section 3.1, assessing the accuracy of a profiler is not directly possible, so we assess the disagreement between profilers instead. We first determine the hottest method per benchmark for each profiler across the 30 runs based on the highest average percentage of run time. We then normalise the names, since different profilers will report names in a different format, and finally count how many different first hottest methods are reported for each benchmark. In the best case, all profilers agree and the size of the union is one. In the worst case all profilers report a different method, and we get a union of six.

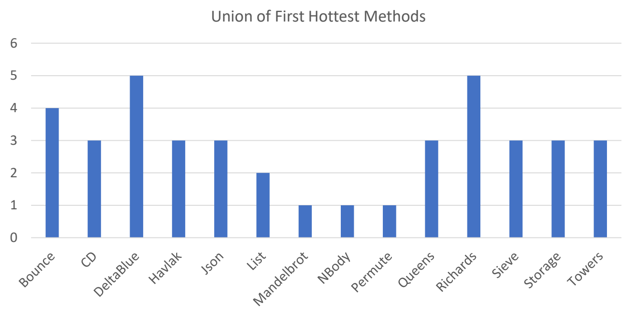

Figure 5. Union$_1$, i.e., count of different first hottest methods across all profilers.

When looking at the results in Figure 5, we see that our six profilers agree only on Mandelbrot, NBody, and Permute completely and we see exactly one first hottest method. For all other benchmarks, we see that there were 1–4 profilers that identified a different method as hottest. DeltaBlue and Richards show here the least consistency, with only two profilers agreeing on the hottest method, i.e., we see five different hottest methods.

If at least one profiler disagrees, then that one profiler could be inaccurate. Since developers may use the ranking to prioritise optimisation projects, the observed high disagreement suggests that prioritisation decisions may be based on unreliable data.

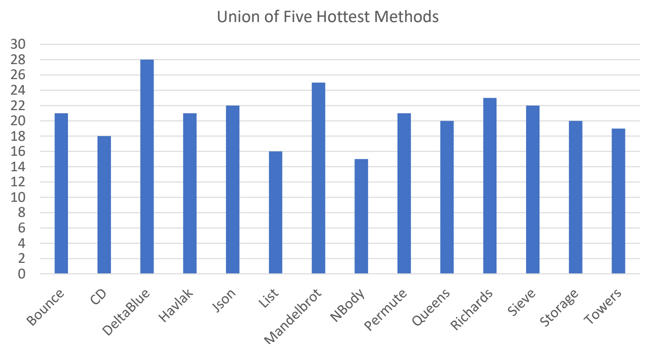

Figure 6. Union$_5$, i.e., count of different methods reported for the five hottest methods across all profilers.

Figure 6 shows the results for the union for the 5 hottest methods. Here we count the number of different methods identified as being in the top 5, ignoring their ordering. Thus, in the best case, we count the same five methods. In the worst case, each of the six profilers would report five different methods, giving us a count of 30 different methods, indicating complete disagreement.

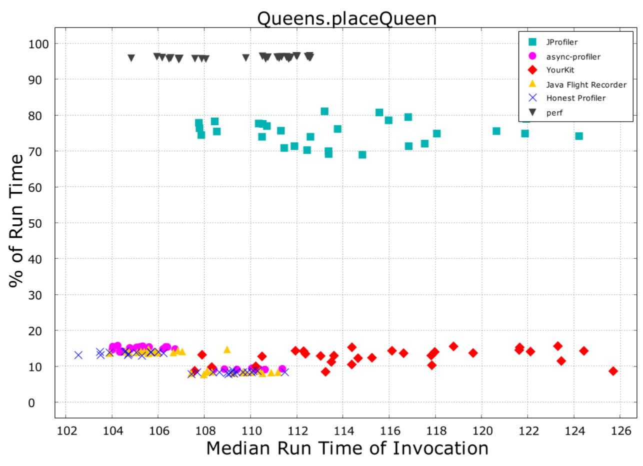

Figure 7. Percentage of run time over median iteration run time for the method `Queens.placeQueen`, for all 6 profilers. We observe clearly different amounts of percentage of run time between many profilers, showing clear disagreement.

Even though there was good self-reliability, these results suggest that profilers struggle to identify the same set of 5 hottest methods. Since the identified hottest methods differ from each other, at least one of the profilers must incorrectly identify the hottest methods.

As Figure 6 shows, only NBody reaches the best score of 50% agreement on the 5 hottest methods between all profilers. This is because async-profiler, Honest Profiler, and Java Flight Recorder are in complete agreement. For the List benchmark, async-profiler, Java Flight Recorder and YourKit were in agreement except for one method, which lead to the union set of 16 methods.

DeltaBlue is the worst benchmark in terms of agreement with only two pairs of profilers in agreement. This is not surprising, as shown in Figure 2, there is high imprecision on this benchmark resulting in all but two pairs of profilers agreeing on a method.

For Mandelbrot, which should only have two hot methods (see Table 1), the profilers report a variety of other methods, including for output of results and the bytecode parser, causing more than the maximum of 12 expected methods.

To answer RQ2, i.e., how reliable profilers are at identifying the same hottest methods, Figure 5 shows that profilers reliably identifying the first hottest method for few benchmarks, and there are benchmarks where six profilers report five different methods. When considering Figure 6, reliability further decreases when comparing the 5 hottest methods even without taking ordering into account.

4.2.3 Analysis of Individual Methods.

The unions of the profilers gave a coarse summary of how profilers compare to one another. We will now take a closer look at selected examples, to see how the data looks in detail.

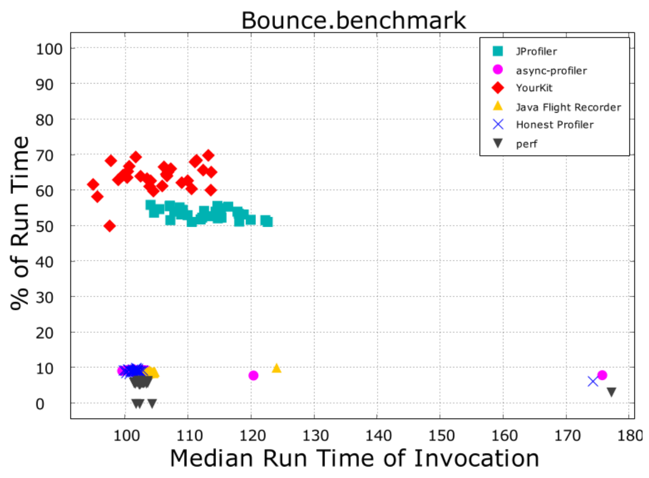

Figure 8. Percentage of run time over median iteration run time for the method `Bounce.benchmark` for all 6 profilers. We observe clearly different amounts of percentage of run time between many profilers, showing clear disagreement

Our first example in Figure 7 shows how the profilers attribute run time to the `Queens.placeQueen` method. It shows the median iteration time on the x-axis and the percentage of run time on the y-axis, for 30 runs. This allows us to see possible patterns caused for instance by optimisation decisions of the just-in-time compiler.

perf (▼) and JProfiler (■) attribute the majority of the run time to the `Queens.placeQueen` method, whereas the other profilers attribute much less time to it.

For async-profiler (●), Java Flight Recorder (▲), YourKit (◆), and Honest Profiler (×), we see run times in the range between 8% and 15%. JProfiler attributes 68% to 82% of the run time to it and perf is quite stable at 87%.

For Honest Profiler (×) and async-profiler (●), we further see two clear groups separated by the median iteration time, which suggests that we may observe different optimisation decisions between runs. Though, for the other profilers, we do not see a similarly tight grouping.

To answer RQ2, perf and JProfiler disagree strongly with the other four profilers for this method, this suggests that either two of the profilers or all four are incorrect (as in, show the wrong amounts of percent of run time) for this benchmark. Our working hypothesis is that perf and JProfiler do not use information about inlining decisions when attributing run time.

Figure 8 is another example of profilers disagreeing with one another. Honest Profiler (×), async-profiler (●), Java Flight Recorder (▲) are grouped around 6% to 10% run time on the y-axis, with a few outliers which have a much larger iteration run time indicated on the x-axis, but still maintain the same amount of percent of run time.

perf (▼) is in close agreement with the other three profilers, but YourKit (◆) and JProfiler (■) attribute 50% to 70% percent of the run time to this method. This example again shows that different profilers can have large disagreements in the amount of percent of run time for specific methods.

To conclude this section, Figure 7 and Figure 8 both show that profilers can strongly disagree with each other. The amount of disagreement in percent of run time is large enough that we can assume that at least one of these profilers is incorrect in its assessment of these benchmarks. We demonstrated that profilers can be wildly inaccurate in their allocation in percent of run time, and therefore multiple profilers cannot reliably identify the hottest methods.

4.3 Overhead

To assess the overhead of our profilers, we recorded the time spent in each benchmark iteration with and without a profiler. We take the median iteration time to calculate the overhead. Figure 9 shows the distribution across all 14 benchmarks as a boxplot for each profiler.

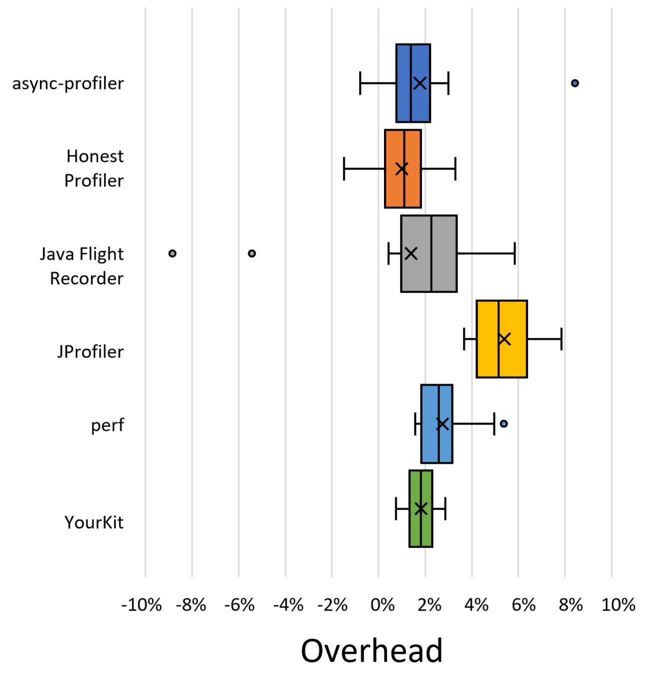

Figure 9. The average percentage of run time overhead induced by each profiler. The × indicates the mean.

The mean overhead, indicated with an ×, ranges from 1% to 5.4%, which means there is a significant difference depending on the profiler. JProfiler, which has the highest overhead of 5.4% may also be considered the most reliable based on Table 2. However, arguably it has five times the overhead than the best profiler, which may make it impractical for some use cases and may bias the result in other ways because of a non-negligible observer effect [17, 21].

There are also outliers where the overhead is negative. This suggests that these benchmarks ran faster with a profiler being attached than without it. A possible explanation for this is the observer effect. The presence of an attached profiler may change timings and subsequently cause the compiler to make different optimisation choices than it does normally when there is no profiler attached, causing a speedup.

With regards to RQ3, we found that the average overhead is practical for many applications. The worst case we observed was an overhead of 7.8%, which may be impractical for large long-running systems. However, this run time overhead seems acceptable for short-running programs.

5 Discussion

In this section, we discuss observations we made while collecting and processing our data.

5.1 A Closer Look at the Queens Benchmark

In our collection of results, we discovered a trend in the data for certain profilers and certain methods. This is exemplified in async-profiler and the Queens benchmark. Table 3 shows the three hottest methods and the minimum/maximum percentage of run time they each had across 30 runs. These three methods were the three hottest observed for every single run. For each of these methods, we observed significant differences between the minimum and maximum run time percentage: `getRowColumn` had a difference of 15.17%, `placeQueen` had 10.01% and `setRowColumn` had 6.88%.

Table 3. Minimum and maximum percentage of run time attributed to the three hottest methods for the Queens benchmark, for async-profiler, over 30 profiles.

Benchmark	Minimum	Maximum	Difference
getRowColumn	56.81%	71.98%	15.17%
setRowColumn	16.81%	26.82%	10.01%
placeQueen	8.92%	15.8%	6.88%

To investigate these differences, we compare the runs with one another in more detail. Figure 10 plots out all 30 runs of async-profiler, comparing the percent of the run time of a method over the median of execution for an invocation. Figure 10 displays a bimodal trend: between 107 and 109 milliseconds on the x-axis, there is a separation between the two groups of plots in the amount of percentage of run time, with `setRowColumn` and `placeQueen` both lowering in run time percentage as `getRowColumn` increases in this regard. The fact that some methods fall as others rise in run time percentage is logical, since the total sum of percentages for all methods must always be equal to 100%: therefore if the run time percentage increases for one method, it must decrease for another in turn.

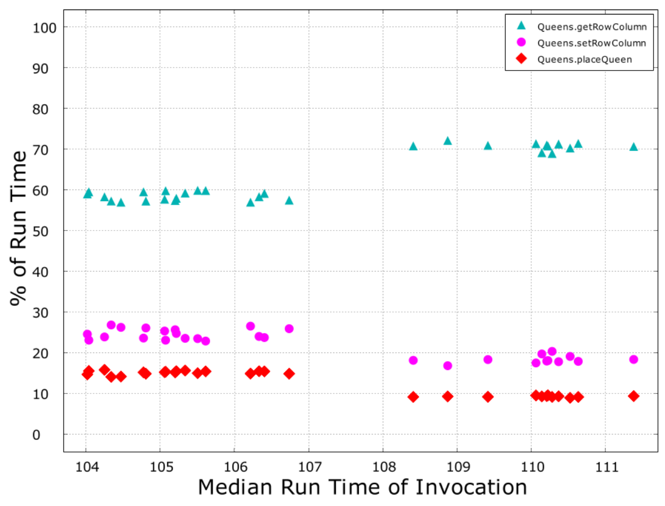

Figure 10. Percentage of run time over median run time of invocation for the three hottest methods in the Queens Benchmark with async-profiler. This displays a bimodal trend.

A possible reason for this bimodal trend may be different optimisation choices by the compiler. When the run time is between 108 and 109 milliseconds, the compiler could make a different choice in optimisation. This choice could cause a method to take a longer or shorter percentage of run time, affecting the results of the profile. This may all be caused by the observer effect: as profilers are adding their own overhead onto the running program, this may alter timings and cause the compiler to see different execution metrics resulting in different optimisations.

Returning back to Figure 7, Honest Profiler and Java Flight Recorder show a bimodal trend much like async-profiler, yet perf and JProfiler do not. For perf and JProfiler, a change in the median run time of a profile seems to have no consistent effect on the percentage of run time for both profilers. perf stays stable at 96%, while JProfiler exhibits variation in the 70%–80% range yet with no clear bisection in its results. For these three profilers, this change in percentage of run time could be considered not to be imprecision, but instead an accurate detection of a change in the performance of the program. As an alternative conclusion, if this is the result of a profiler fault, then all three share this same mistake.

5.2 Groups of Profilers

We have just highlighted similarities between three profilers: async-profiler, Java Flight Recorder and Honest Profiler. As it turns out, this is an observed trend: we found that throughout our results, these three had far more agreements with one another than other sets of profilers.

Figure 11 features the same results as Figure 5, with an added detail: we establish a distinction between these three profilers and the three others. The portion of the bars that

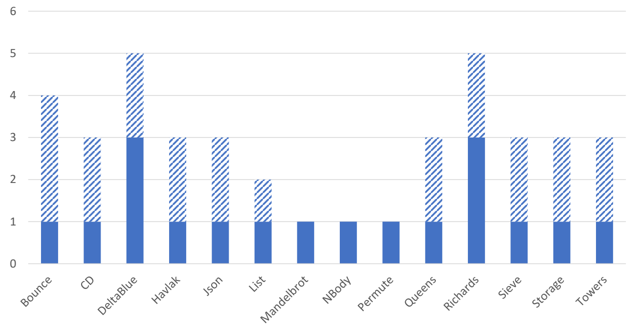

Figure 11. Union of first hottest method across all profiles, highlighting the agreements between async-profiler, Java Flight Recorder and Honest Profiler (solid blue)

is solid shows the union of the hottest method for async-profiler, Java Flight Recorder and Honest Profiler. The striped portion of the bar indicates the union over perf, YourKit or JProfiler. This highlights that async-profiler, Java Flight Recorder and Honest Profiler agree more often than the other 3 profilers. It is also striking that for 12 out of our 14 benchmarks, these three profilers are in total agreement with the first hottest method, i.e. having a union of size 1. However, since they do not fully agree with themselves on DeltaBlue and Richards (see Section 4.1), there is also no agreement among each other here.

5.3 Workload with Expected Profile

In some other work, specific workloads are crafted for which the expected profile is known [3]. We however assumed that this would be too unreliable in the context of a modern JVM with dynamic optimization on top of system with complex caching effects, a CPU that does, e.g., out of order execution and branch prediction. Our assumption here is that we can not reliably predict the run time distribution for a program to be usable as an oracle for what profilers should give.

6 Related Work

In addition to comparing to Mytkowicz et al. [16] in more detail, we also briefly discuss work on optimisations of profiling techniques, as well as some open source work.

The work of Mytkowicz et al. [16] was a major motivation for our study. They investigated the accuracy of four Java profilers, and their evaluation showed that profilers have major disagreements as to which methods are the hottest. As one of the possible reasons for this discord, they identify safepoint bias, which restricts stack sampling to GC safepoints and means not all program points have the same likelihood of being profiled. As a solution, they propose to collect samples truly randomly by using a dedicated thread with a randomized timing interval, which can also sample threads that are not in a safepoint.

async-profiler and Honest Profiler use HotSpot's support for such sampling independent of safepoints. Nonetheless, when comparing unions of agreement of profilers, we found similar disagreement as reported by Mytkowicz et al. [16]. In some ways, the situation may have gotten even worse, since they reported very small 95% confidence interval around the mean run time they reported for 30 runs.

Hofer et al. [8] propose techniques to improve stack sampling and reduce the overhead and possible bias of sampling profilers. They use partial safepoints, which can reduce the number of threads that need to participate in a safepoint and use incremental stack sampling, which minimizes the overhead. In their evaluation, they use the DaCapo benchmarks and assess the accuracy of calling context trees, which are merged stack traces. They focus on comparing the calling context trees and the presence of hot edges, which we consider weaker criteria than what we use here. However, for these calling context trees, they do not identify any major accuracy issues.

A very recent paper on a profiler for Python [3] uses a microbenchmark to assess the precision of its sampling-based approach compared to instrumentation-based Python profilers. Though, their evaluation is assuming the absence of just-in-time compilation. Thus, it can not be applied in our context. Since their work uses CPython, which only has an interpreter, we cannot directly compare the overhead they report either.

Outside of academic literature, the issue of profiler bias and issues around accuracy has also found wider attention and has been documented in several blog posts. This includes a series of posts by Nitsan Wakart,[16] who reported on technical details of profiler, limitations, and discussed the underlying issues of safepoint bias.

Similarly, Alexey Ragozin[17] assesses the accuracy issues on specific practical use cases, for instance also including use of Java's Native Interface, making the case that profilers have different tradeoffs in what they can profile.

7 Conclusion and Future Work

Sampling profilers are one of the main tools developers reach for to identify the source of performance issues. However, they have known limitations around sampling bias, for instance caused by the safepoint bias observed by Mytkowicz et al. [16], and a general observer effect, for instance caused by profiling overhead.

With the availability of profilers that aim to avoid specifically the safepoint bias such as async-profiler and Honest Profiler, in this work, we investigate whether profilers give precise results, reliably identify the same hottest methods, and assess what their overhead on the execution is. We study the six Java profilers async-profiler, Honest Profiler, Java Flight Recorder, JProfiler, perf, and YourKit. Instead of using largely opaque application-sized benchmarks, we use the Are We Fast Yet benchmarks [14] with 9 micro and 5 larger benchmarks with fully deterministic behaviour, which give us a stable foundation and a better chance to match profiles with our understanding of the benchmark code.

Our results show that a profiler normally identifies the same hottest method reliably across multiple runs. However, we observed major outliers where they could not identify a hottest method reliably, which may mean that their precision is application-dependent. Furthermore, we found that there is high disagreement between the profilers on what the hottest method of a benchmark is, and which set of methods should be contained in the five hottest methods.

Our results are concerning as they indicate that profilers are not correctly identifying the same methods as the hottest ones, and are not associating the correct amount of percent of run time to them. Since there is also no reliable consensus between a majority of them, this also casts doubt on how reliable we can consider sampling profilers on a modern JDK.

The main issues for future work are precision and reliability of sampling profilers. While they have largely good precision compared to themselves, benchmarks such as DeltaBlue and Richards suggests that there might be deeper issues. While we have not yet investigated the underlying issues in detail, one hypothesis could be a difference in dynamic optimisation decisions, which may be caused by the presence of the profiler or its overhead. To address the issue, replayable optimisation decisions [7] could be one possible solution. Another might be a principled approach to steer optimisation decisions, and possibly randomize them and aggregate profiles over multiple runs.

Future work should also investigate the different implementation techniques in more detail to identify avoidable causes for imprecision.

The experiment setup we have built for this paper is available as open source.[18] It can be extended to include additional Java profilers. Future work is invited to build on our system for instance to investigate novel sampling approaches or to integrate with JIT compilers to avoid any observer effects.

8 Acknowledgements

This work was supported by a grant from the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

References

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

[16] *The Pros and Cons of AsyncGetCallTrace Profilers*, Nitsan Wakart, https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html
[17] *Lies, darn lies and sampling bias*, Alexey Ragozin, https://blog.ragozin.info/2019/03/lies-darn-lies-and-sampling-bias.html

[18] https://github.com/HumphreyHCB/AWFY-Profilers

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. 1999. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, 314–324. https://doi.org/10.1145/320385.320418

[3] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. 2023. Triangulating Python Performance Issues with SCALENE. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (Portland, Oregon, USA). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[5] De Bus, Bruno and Chanet, Dominique and De Sutter, Bjorn and Van Put, Ludo and De Bosschere, Koen. 2004. The Design and Implementation of FIT: a Flexible Instrumentation Toolkit. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, 29–34.

[6] Jason Gait. 1986. A probe effect in concurrent programs. *Software: Practice and Experience* 16, 3 (1986), 225–233. https://doi.org/10.1002/spe.4380160304

[7] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java Performance Evaluation through Rigorous Replay Compilation. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA'08)*. ACM, 367–384. https://doi.org/10.1145/1449764.1449794

[8] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2015. Lightweight Java Profiling with Partial Safepoints and Incremental Stack Tracing. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. ACM, 75–86. https://doi.org/10.1145/2668930.2688038

[9] IBM. 1964. *IBM System/360 Principles of Operation*. IBM Press.

[10] D. M. Ritchie K. Thompson. 1974. *UNIX programmer's manual fifth edition*. Bell Telephone Laboratories, Inc.

[11] 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

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, 190–200. https://doi.org/10.1145/1065010.1065034

[13] Stefan Marr. 2023. *ReBench: Execute and Document Benchmarks Reproducibly*. Open Source Software. https://doi.org/10.5281/zenodo.1311762 Version 1.2 https://github.com/smarr/ReBench/.

[14] 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

[15] Jonathan Misurda, James A. Clause, Juliya L. Reed, Bruce R. Childers, and Mary Lou Soffa. 2005. Demand-Driven Structural Testing with Dynamic Instrumentation. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. ACM, 156–165. https://doi.org/10.1145/1062455.1062496

[16] 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

[17] Indigo Orton and Alan Mycroft. 2021. Tracing and Its Observer Effect on Concurrency *(MPLR 2021)*. Association for Computing Machinery, 88–96. https://doi.org/10.1145/3475738.3480940

[18] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 31–47. https://doi.org/10.1145/3314221.3314637

[19] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997* (Seattle, Washington) *(NT'97)*. USENIX Association, 1.

[20] Ilene Seelemann. 1995. Limiting the Probe Effect in Debugging Concurrent Object-Oriented Programs. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '95)*. IBM Press, 56–68.

[21] Isaac Sjoblom, Tim Snyder, and Elena Machkasova. 2011. Can You Trust Your JVM Diagnostic Tools? *(MICS 2011)*. Midwest Instruction and Computing Symposium. https://micsymposium.org/mics_2011_proceedings/mics2011_submission_26.pdf

[22] 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

[23] 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

A Raw Data

The following table contains part of our raw data to determine the hottest method to complement Figure 5 and Figure 11.

Profiler	Method	% of run time	Profiler	Method	% of run time
Bounce			*Permute*		
Async	Bounce$Ball.bounce	84.0%	Async	Permute.permute	58.3%
honest-profiler	Bounce$Ball.bounce	82.7%	honest-profiler	Permute.permute	51.1%
JavaFlightRecorder	Bounce$Ball.bounce	84.9%	JavaFlightRecorder	Permute.permute	55.9%
JProfiler	Bounce.benchmark	53.5%	JProfiler	Permute.permute	98.4%
Perf	Benchmark.innerBenchmarkLoop	74.3%	Perf	Permute.permute	49.5%
YourKit	Bounce$Ball.<init>	63.7%	YourKit	Permute.permute	55.7%
CD			*Queens*		
Async	cd.RedBlackTree.treeInsert	20.9%	Async	Queens.getRowColumn	63.2%
honest-profiler	cd.RedBlackTree.treeInsert	20.8%	honest-profiler	Queens.getRowColumn	62.5%
JavaFlightRecorder	cd.RedBlackTree.treeInsert	22.7%	JavaFlightRecorder	Queens.getRowColumn	64.5%
JProfiler	cd.Vector2D.<init>	44.2%	JProfiler	Queens.placeQueen	75.3%
Perf	cd.RedBlackTree.treeInsert	24.4%	Perf	Queens.placeQueen	96.2%
YourKit	cd.CollisionDetector.isInVoxel	22.9%	YourKit	Queens.setRowColumn	62.3%
Havlak			*Richards*		
Async	som.Vector.forEach	16.8%	Async	Richards.benchmark	36.9%
honest-profiler	deltablue.Planner.chainTest	15.4%	honest-profiler	richards.Scheduler.schedule	45.5%
JavaFlightRecorder	deltablue.AbstractConstraint.<init>	15.1%	JavaFlightRecorder	richards.Packet.<init>	13.0%
JProfiler	som.Vector.<init>	29.7%	JProfiler	richards.TaskControlBlock.runTask	84.1%
Perf	deltablue.Planner.chainTest	28.1%	Perf	richards.RBObject.<init>	37.0%
YourKit	som.Vector.forEach	36.1%	YourKit	richards.TaskControlBlock.runTask	38.2%
Json			*Sieve*		
Async	java.lang.String.equals	21.6%	Async	Sieve.sieve	89.6%
honest-profiler	java.lang.String.equals	21.5%	honest-profiler	Sieve.sieve	88.6%
JavaFlightRecorder	java.lang.String.equals	25.8%	JavaFlightRecorder	Sieve.sieve	90.6%
JProfiler	java.lang.String.substring	91.3%	JProfiler	Sieve.sieve	93.4%
Perf	json.JsonParser.readStringInternal	32.2%	Perf	Benchmark.innerBenchmarkLoop	84.7%
YourKit	java.lang.String.equals	23.4%	YourKit	Sieve.benchmark	88.4%
Mandelbrot			*Storage*		
Async	List.isShorterThan	99.6%	Async	Storage.buildTreeDepth	42.6%
honest-profiler	List.isShorterThan	98.0%	honest-profiler	Storage.buildTreeDepth	40.7%
JavaFlightRecorder	List.isShorterThan	100.0%	JavaFlightRecorder	Storage.buildTreeDepth	45.1%
JProfiler	List.isShorterThan	99.7%	JProfiler	som.Random.next	99.5%
Perf	List.tail	99.0%	Perf	java.util.Arrays.setAll	87.5%
YourKit	List.isShorterThan	99.5%	YourKit	java.util.Arrays.setAll	42.1%
NBody			*Towers*		
Async	nbody.NBodySystem.advance	42.3%	Async	Towers.pushDisk	35.3%
honest-profiler	nbody.NBodySystem.advance	40.9%	honest-profiler	Towers.pushDisk	33.8%
JavaFlightRecorder	nbody.NBodySystem.advance	41.0%	JavaFlightRecorder	Towers.pushDisk	38.2%
JProfiler	nbody.NBodySystem.advance	99.2%	JProfiler	Towers.moveDisks	94.2%
Perf	nbody.NBodySystem.advance	47.7%	Perf	Towers.popDiskFrom	72.5%
YourKit	nbody.NBodySystem.advance	44.5%	YourKit	Towers.pushDisk	33.0%

Received 2023-06-29; accepted 2023-07-31