

UNIVERSITY OF KENT

DOCTORAL THESIS

**Reclaiming Profiler Ground Truth:
Determining and Improving the Accuracy
of CPU Profilers in the Presence of
Dynamic Compilation**

Author:

Humphrey Hugo Cosmo Burchell

Supervisor:

Prof. Dr. Stefan Marr

A thesis submitted in fulfillment of the requirements

for the degree of Doctor of Philosophy

in the

Programming Languages and Systems Research Group

School of Computing

5 June 2026

Abstract

In the presence of highly optimising compilation and modern hardware, understanding where a program spends its time is complex. Profilers are used to understand program execution, but they introduce their own complexity through the observer effect, the phenomenon where measuring something changes the thing being measured.

In this work, we empirically evaluate sampling- and instrumentation-based profilers. For sampling-based profilers, we evaluate how often results change across multiple runs, giving an indication of self-precision, and compare profilers against each other to determine their disagreement when profiling the same program execution. For instrumentation-based profilers, we investigate accuracy, overhead, and their effect on compilation through additional generated code and compilation statistics. To alleviate known issues such as the observer effect, we implement our own profiler, BuboC, that makes use of late-compiler-stage instrumentation to avoid interfering with compilation decisions and reduce the tool’s overhead.

We solve a key limitation in evaluating profilers: the absence of a known ground truth against which to compare results. Without knowing the correct execution profile, it is extremely difficult to determine whether a profiler is accurate, or to what degree. To address this, we introduce a novel approach based on deliberate machine-code slowdown. We insert additional instructions to induce a controlled slowdown, allowing us to evaluate a profiler based on its ability to correctly attribute this additional execution time. This provides a practical approach to measuring profiler accuracy and the degree of that accuracy.

Finally, we evaluate a loop-level instrumentation profiler using late-compiler-stage instrumentation. We find that this loop-level profiler provides more accurate measurement of how much time the CPU spends in loops than sampling-based profilers, while maintaining practical overhead and reduced observer effect. Overall, this work provides a clear identification of the issues present in modern profiling under highly optimising compilation and hardware complexity, as well as an approach to alleviate these issues and improve profiler accuracy.

Acknowledgements

Firstly, I would like to thank my examiners, Stephen Kell and Mark Batty, not only for their time, effort, and care in reading this work, but also for their thorough examination process. I would also like to thank the University of Kent and the PLAS group, as well as David Castro-Perez, Anna Jordanous, Rogério de Lemos, and Michael Vollmer. I am also grateful to the friends and colleagues I have encountered along the way. To name but a few, Sophie, Jonah, and Octave have all had a great impact on how I worked and on my journey throughout my PhD.

I do not think I would have been able to start this work more than three years ago without the persistent effort of my supervisor, Stefan Marr. His constant willingness to help, seemingly endless patience, and the time he has invested in me have been invaluable. I have greatly enjoyed my PhD and research, finding both engaging and entertaining. I think this was largely due to the way in which we conducted the PhD, which was shaped by Stefan's approach and the time he gave me. For this, I am deeply grateful to Stefan. I would also like to thank Clíodhna for her kindness and support along the way.

And finally, I would like to thank both of my parents for their support and encouragement. My father especially, someone who is objectively an engineer across many disciplines, has always amazed me with his ability to understand and discuss my work. Even when the subject is so specialised that, in many cases, only my supervisor and I would be expected to know it in depth, he somehow manages to follow, question, and converse about it as though he had already considered many of the same ideas and questions himself. I am deeply grateful to them both.

Contents

Abstract	iii
Acknowledgements	v
1 Introduction - Measuring an Unstable World	1
1.1 Thesis Problems	3
1.2 Research Goals	4
1.3 Key Insights	6
1.4 List of Publications	7
2 Background	9
2.1 Profilers and Profiling Techniques	9
2.2 Sampling-Based Profilers	10
2.2.1 CPU Sampling	10
2.2.2 Problems with Sampling-Based Profilers	11
Sampling Overhead	12
Sampling Observer Effect	12
Sampling Randomness and Safepoint Bias	13
Reducing Safepoint Bias	13
Non-determinism in JVMs	14
2.3 Instrumentation-Based Profilers	15
2.3.1 Instrumentation Overhead	16
2.3.2 Instrumentation Observer Effect	16
2.4 Accuracy, Precision, and Realism	16
2.5 Graal Compiler	17
2.5.1 Compilation Process	17
2.6 Profilers and Inlining	19

2.7	Basic Blocks, Pipelining, and Out-of-Order Execution	20
2.8	VTune, A Profiler Using Hardware Event Sampling	21
2.9	Are We Fast Yet and ReBench	21
2.9.1	Benchmarking Execution Time	23
2.10	Sampling-Based Profilers Evaluated in This Work	23
3	Inaccuracy in Sampling Based Profilers	25
3.1	Introduction	25
3.2	Research Questions and Methodology	27
3.2.1	Research Questions	27
3.2.2	Experimental Setup	28
3.2.3	The Chosen Profilers	28
3.2.4	Benchmarks	29
3.3	Results	30
3.3.1	Self-Comparison of Profilers	30
	Precision	30
	Reliability	33
3.3.2	Comparisons Across Profilers	35
	Average Comparison	35
	Unions	36
	Analysis of Individual Methods	38
3.3.3	Overhead	40
3.4	Discussion	41
3.4.1	A Closer Look at the Queens Benchmark	41
3.4.2	Groups of Profilers	43
3.4.3	Workload with Expected Profile	44
3.5	Related Work	44
3.6	Conclusion	45
4	Late-Stage Compiler Instrumentation	47
4.1	Introduction	47
4.2	State of the Art in Instrumentation in JIT Compilation	49
4.2.1	Research questions	49

4.2.2	Assessing Run-time Overhead	50
4.2.3	Assessing Impact on Compilation	51
4.3	BuboC: Late-Compiler-Stage-based Instrumentation	56
4.3.1	Late-compile-stage Instrumentation Overview	56
4.3.2	Implementation of the Low-tier Instrumentation	57
4.3.3	Reducing Profiling Overhead for Small Methods	58
4.3.4	Attributing Cycles to Inlined Methods	59
4.4	Research Questions and Methodology	60
4.4.1	Research Questions	60
4.4.2	Experimental Setup	60
	Profilers	61
4.5	Evaluation of Late-Compiler-Stage-based Instrumentation	61
4.5.1	Run-time Overhead	61
	BuboC vs Classic Instrumentation	61
	BuboC vs Sampling	62
4.5.2	Tradeoffs of Estimating Small Compilation Units	63
4.5.3	Precision of Late-Compile-Stage Instrumentation	65
4.6	Practicality and Limitations	67
4.7	Related Work	70
4.8	Conclusion	71
5	Approximating the Ground Truth with Machine Code Slowdown	73
5.1	Introduction	73
5.2	Existing Approaches to Assessing the Accuracy of Java Profilers	75
5.3	Divining Profiles: Approximating a Ground Truth with Basic-Block- level Slowdowns	76
5.3.1	Approximating the Ground Truth with Proportional Slowdowns	77
5.3.2	Slowing Down a Program Proportionally and Accurately	78
5.3.3	Implementation of the Divining Process with our GroundTruth Scheduler	79
5.3.4	Implementation of Compiler Phases in Graal	81

	Marker Phase	82
	Slowdown Phase	83
5.4	Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level	85
5.4.1	Requirements for Machine-code-level Slowdown	85
	Distributed Slowdown	86
5.4.2	Evaluating Candidates	86
	Methodology	86
	Candidate Instructions	87
	Required Number of Slowdown Instructions	88
	Run-time Overhead	88
	Distribution of Slowdown Among Blocks	89
	Performance behaviour with Slowdown	90
	Importance of Distributing Slowdown	91
	Best Candidate for Accurate, Side-Effect-Free, and Distributed Slowdown	92
5.5	Evaluating Machine-Code Slowdown for Approximating Ground Truth	93
5.5.1	Research Questions	94
5.5.2	Experimental Setup	95
5.6	Results	97
5.6.1	Experiment 1: The Effect of Slowdown on Program and Basic Block Run Time	98
5.6.2	Experiment 2: The Effect of Block Slowdown on Profilers Reported Methods	99
5.6.3	Experiment 3: Can Profilers Detect Individual Blocks Being Slowed Down?	102
	Experimental Setup	103
	Results	104
5.7	Discussion	106
5.7.1	Compiler-Generated Debugging Information	106
	Tradeoffs of DebugNonSafepoints	106
	Instructions Without Debugging Information	107

Challenges of Tracking Accurate Debugging Information	107
Debugging Information for Slowdown Instructions	108
5.7.2 Hardware- and JVM-Related Factors Affecting Accuracy	109
Biased Sampling Skid	109
Microarchitecture-Specific Slowdown	109
Accuracy of VTune and Sampling Hardware Counters	110
Inaccuracy for Small Blocks	111
Run-Time Overhead of VTune	111
Using a Static Performance Model Instead of an Instruction Profiler	112
Sampling and Safepoint Bias	112
5.7.3 State-of-the-Art JIT Compilers and Bytecode-level Slowdown Insertion	113
5.7.4 Generalizing to Other Systems and Languages	114
5.7.5 Overall Divining Time and Future Optimization Opportunities	114
5.7.6 Experimental Time and Effort	115
5.8 Related Work	115
5.9 Conclusion	117
6 Loop-Level Instrumentation: The BuboL Profiler	119
6.1 Introduction	119
6.2 Motivation for Loop-Level Profiling	121
6.2.1 Goal of This Chapter	121
6.2.2 Instrumentation of Loops	122
6.3 BuboL	124
6.3.1 Overview of BuboL	124
6.3.2 Late-Late-Compiler-phase Instrumentation	125
6.3.3 BuboL Probe	126
6.4 Evaluation of BuboL	128
6.4.1 Research Questions	128
6.4.2 Experimental Setup	129
6.4.3 Does BuboL change compiler optimisations?	132

	Placement of BuboL in the Graal pipeline	132
6.4.4	BuboL's Overhead	135
6.4.5	BuboL's effect on performance behaviour	138
6.4.6	BuboL's observer effect on loops	141
	Relationship between BuboL probe and loop frequency	144
	To what extent does BuboL change performance behaviour?	146
	Threats to Validity	147
6.4.7	Accuracy of profilers in measuring the proportion of CPU time spent in loops	148
	Methodology	148
	BuboL, Async and JFR's overall accuracy	150
	Async and JFR's accuracy	151
	Oversampling and Undersampling	151
	Safepoint Bias	154
	BuboL's accuracy	156
	RQ3: Do late late-instrumentation-based profilers and sampling- based profilers record the proportion of CPU time spent in loops accurately?	158
6.5	Related Work	158
6.6	Conclusion	161
7	Discussion of the Challenges Modern Profilers Face	163
7.1	Method Boundaries	164
7.1.1	How do profilers identify what method they are in during execution?	164
7.1.2	Issues with method boundaries	165
7.1.3	A core issue of profiler inaccuracy	168
7.1.4	A proposed strategy for instrumentation to more accurately record time spent in methods	169
7.2	What Is Useful to Report in a Profile	174
7.2.1	Related work on non-method-centric profiling	176
7.3	Conclusion	177

8 Conclusion & Future Work	181
8.1 Conclusion	181
8.2 Future Work	183
8.3 Closing Words	186
A Appendix	187
A.3 Chapter 3 Appendix	188
A.4 Chapter 4 Appendix	189
A.5 Chapter 5 Appendix	190
A.6 Chapter 6 Appendix	195
Bibliography	199

Chapter 1

Introduction - Measuring an Unstable World

Determining where a program spends its time during execution is not straightforward. Highly optimising compilation and modern hardware in today's systems make it difficult to relate a program's source code to its execution behaviour. Compilation extensively changes the structure of the source code. With optimisations such as inlining and loop transformations, methods are moved, merged or optimised away, and loops can be reduced to a constant or fused with other loops.

Determining the source of each executed machine-code instruction is also difficult, because compilers can overwrite, discard, or make source information inaccurate during optimisation (Stinnett et al. 2026). This can leave machine-code instructions with incorrect or missing mappings back to the original method or line (Assaiante et al. 2023; Kell et al. 2024), making it difficult to account for the time spent executing these instructions accurately.

In addition to these mapping challenges, compiler decisions are themselves sensitive to run-time behaviour. Systems such as the Java virtual machine (JVM), V8, and .NET use dynamic compilation to optimise programs during execution. Techniques such as profile-guided optimisation can influence which optimisations are applied, introducing variability in compilation decisions and causing both compilation and execution behaviour to differ between runs (Georges et al. 2008).

Modern hardware introduces additional complexity to program execution, leading to non-deterministic performance. CPUs employ out-of-order execution and pipelining, which change the order and timing of instruction execution. Out-of-order

execution results in instructions executing not in the sequential order in which they are generated, while pipelining allows instructions to execute simultaneously. Memory access latency is highly variable, causing the same instruction to take different amounts of time depending on cache and memory behaviour (Yi et al. 2020).

Although run-time profiling is one of the oldest approaches for analysing program execution, having existed for over 60 years (IBM 1964), profiling modern systems remains challenging. Profilers typically measure how a program uses resources such as CPU, GPU, and memory, and software developers use this information to identify performance bottlenecks and optimisation opportunities.

However, profilers do not simply observe a constantly changing system; they can also influence the execution they measure. Profilers can suffer from the observer effect (Machkasova et al. 2009; Zheng et al. 2015), where observing a system influences and changes the system being observed. In the context of profiling, this means that a profiler can alter program execution to such an extent that the measured behaviour no longer accurately reflects the program’s unprofiled behaviour. As a result, the collected measurements may no longer identify where the program actually spends most of its time.

The observer effect introduced by profilers is most prevalent with instrumentation (Machkasova et al. 2009), as it actively modifies the program code by inserting probes. Sampling-based profilers are an alternative choice, which periodically halt execution to collect samples of the program. However, a sampling-based approach relies on the probabilistic nature of program execution, assuming that all parts of the program have an equal chance of being sampled. However, this assumption is not guaranteed to hold, as safepoint bias, sampling skid, and other sources of sampling bias can make some parts of a program more likely to be sampled than others (Mytkowicz et al. 2010; Yi et al. 2020).

In the current computing landscape, large-scale systems, and high-demand services place increasing pressure on software to execute efficiently (Manner 2022). This pressure is likely to grow as LLM-based code generation becomes more popular, increasing the amount of inefficient generated code that developers may need to understand and improve (Li et al. 2026). However, given the complexity introduced by dynamic compilation and modern hardware, even determining where a program

spends its time is no longer straightforward.

Profilers are intended to provide this insight, acting as a core tool for understanding program performance behaviour. However, as discussed, they can be inaccurate and influenced by the systems they attempt to measure. As a result, developers may be presented with misleading information, leading to ineffective or misguided optimisation efforts.

This problem is further compounded by the lack of a known ground truth. Without a reliable reference for correct performance behaviour, it is difficult to determine whether a profiler is accurate, or even to what extent its results can be trusted. Consequently, performance optimisation risks becoming dependent on intuition rather than reliable measurement.

Ensuring the accuracy of profiling tools is therefore essential. Without accurate profilers, our ability to understand, evaluate, and improve program performance is fundamentally limited.

1.1 Thesis Problems

This thesis seeks to determine and improve the accuracy of CPU profilers in the presence of highly optimising compilation and modern hardware. In such conditions, determining accuracy exposes fundamental problems.

Lack of recent empirical studies of profiler accuracy Existing empirical studies on the accuracy and observer effect of JVM profilers are from more than a decade ago, with key studies by Machkasova et al. 2009 and Mytkowicz et al. 2010. Since then, dynamic compilation and modern hardware have continued to evolve, increasing the complexity of program execution. As a result, the accuracy of modern JVM profilers, when running on HotSpot, is not well established empirically.

Profiler inaccuracy and observer effect Both sampling-based and instrumentation-based profilers are subject to the observer effect, whereby the act of measurement alters program behaviour. Instrumentation-based profilers, in particular, can significantly change compilation decisions and execution characteristics. Consequently, the

performance behaviour reported by a profiler may not reflect the behaviour of the uninstrumented program.

Lack of ground truth A fundamental challenge in evaluating profilers is the absence of a known ground truth. Determining whether a profiler is correct requires comparison against the true execution behaviour of a program, which is difficult to obtain in modern systems. Without such a reference, it is not possible to reliably assess profiler accuracy or quantify the degree of error.

Unstable method-level attribution Profilers often use methods as the main abstraction for reporting where execution time is spent. However, in dynamically-compiled systems, method boundaries are unstable. Inlining, code motion, and other optimisations can mix, move, duplicate, or remove code from different methods, while incomplete source or debug information can make attribution less precise. As a result, method-level profiles may not accurately reflect where execution time is spent.

Unclear accuracy trade-off between sampling and instrumentation Instrumentation-based profilers can collect a larger variety of information, and at greater granularity, than sampling-based profilers. However, in highly optimised compiled systems, it remains unclear whether this additional detail offsets the observer effect and overhead introduced by instrumentation, or whether instrumentation-based profilers can match or exceed the accuracy of sampling-based profilers.

This thesis addresses these challenges by evaluating existing profiling techniques, identifying the causes of their inaccuracies, and developing methods to approximate ground truth and improve profiling accuracy.

1.2 Research Goals

This thesis sets out to achieve the following research goals:

Conduct an empirical evaluation of state-of-the-art sampling-based and instrumentation-based profilers. We examine a range of sampling-based and instrumentation-based profilers to assess their accuracy, precision, and run-time overhead. For sampling-based profilers, the goal is to determine how consistently they report

the same results across repeated deterministic benchmark executions, and whether different tools agree on where execution time is spent. For instrumentation-based profilers, the goal is to assess the observer effect they introduce, including their impact on compilation and the extent to which their reported behaviour reflects realistic program execution. Across both profiling approaches, we also examine the run time overhead introduced by profiling.

Develop a methodology for evaluating profiler accuracy in the absence of a known ground truth. In the presence of dynamic compilation and modern hardware, determining the exact ground truth of an execution is impractical. Therefore, we aim to develop a methodology for approximating ground truth through deliberate machine-code slowdown. The goal is to determine whether this approach can accurately and precisely introduce targeted slowdowns at the machine-code basic-block level, while preserving the program’s underlying performance behaviour.

Evaluate a set of x86 machine-code instructions for controlled slowdown. A set of candidate x86 machine-code instructions is evaluated to determine which are most suitable for introducing controlled slowdown. Each instruction is assessed according to its ability to modify basic-block execution time accurately, consistently, and with fine granularity.

Apply the slowdown methodology to evaluate the accuracy of sampling-based profilers. The slowdown methodology is used to construct controlled execution scenarios in which the expected performance change is known. These scenarios are used to assess whether sampling-based profilers can identify the targeted slowdown and accurately report the degree of slowdown introduced, providing a basis for measuring profiler accuracy.

Evaluate late-compiler-stage instrumentation as a way to reduce observer effect and overhead. We explore whether instrumentation placed late in the Graal compilation pipeline, after major optimisation decisions have been made, can reduce overhead and minimise the observer effect on compiler decisions.

Evaluate a loop-level instrumentation-based profiler. Finally, we investigate whether loop-level profiling with an instrumentation-based profiler can provide more accurate attribution of execution time than sampling-based profilers, while maintaining practical overhead and minimising the observer effect.

1.3 Key Insights

The key insights of this thesis are as follows:

Sampling-based profilers can disagree on where execution time is spent. In [Chapter 3](#), we show that across six state-of-the-art Java profilers, the identified hottest methods can differ, even when profiling the same deterministic benchmark. While individual profilers appear self-consistent in the majority of runs, the variation between profilers and across runs shows that their results are not always reliable for guiding optimisation.

Instrumentation-based profilers can significantly change program behaviour. In [Chapter 4](#), we demonstrate that state-of-the-art instrumentation profilers introduce substantial overhead and alter compilation decisions, leading to execution behaviour that significantly differs from the uninstrumented program.

Late-compiler-stage instrumentation reduces overhead and observer effect. In [Chapter 4](#), we show that moving instrumentation to a late stage in the Graal compiler reduces both overhead and interference with compilation, producing behaviour that more closely matches uninstrumented execution.

Machine-code slowdown enables practical evaluation of profiler accuracy. In [Chapter 5](#), we demonstrate that deliberate slowdown at the machine-code basic-block level can introduce controlled performance changes, allowing us to evaluate whether profilers correctly attribute additional execution time.

NOP and MOV provide the most accurate and stable slowdown behaviour. In [Chapter 5](#), we show that NOP and MOV allow for precise and well-distributed slowdown while introducing minimal unintended side effects, making them suitable for approximating ground truth.

Loop-level instrumentation provides more accurate attribution than sampling. In [Chapter 6](#), we show that our loop-level profiler, BuboL, which instruments at a late stage in the Graal compiler, produces more accurate measurements of CPU time spent in loops than the sampling-based profilers we evaluate, while maintaining a practical level of overhead.

1.4 List of Publications

This thesis is based on the following publications.

The work in **Chapter 3** is based on *Don't Trust Your Profiler: An Empirical Study on the Precision and Accuracy of Java Profilers*, which evaluates the accuracy and precision of sampling-based profilers.

Chapter 4 builds on *Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems*, which examines how instrumentation affects program behaviour and introduces a compiler-level instrumentation profiler.

Chapter 5 is primarily based on two publications. *Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown* proposes a methodology for approximating ground truth through controlled slowdown and was accompanied by an evaluated artefact. *Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level* identifies suitable instructions for precise and low-side-effect slowdown.

Additionally, our understanding of language tooling, the benchmark suite, and compiler behaviour used throughout this thesis is informed by contributions to *Execution vs. Parse-Based Language Servers: Tradeoffs and Opportunities for Language-Agnostic Tooling for Dynamic Languages*.

Publications

1. Burchell, H., O. Larose, S. Kaleba, and S. 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, pp. 100–113. ISBN: 9798400703805. DOI: [10.1145/3617651.3622985](https://doi.org/10.1145/3617651.3622985)
2. Burchell, H., O. Larose, and S. 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 2024. Vienna, Austria: ACM, pp. 82–89. ISBN: 9798400711183. DOI: [10.1145/3679007.3685058](https://doi.org/10.1145/3679007.3685058)

3. Burchell, H. and S. Marr (Oct. 2025a). “Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown”. In: *Proceedings of the ACM on Programming Languages*. OOPSLA’25 9.OOPSLA2. ISSN: 2475-1421. DOI: [10.1145/3763180](https://doi.org/10.1145/3763180)
4. Burchell, H. and S. Marr (Aug. 2025b). *Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown (Artifact)*. DOI: [10.5281/zenodo.16911348](https://doi.org/10.5281/zenodo.16911348). URL: <https://doi.org/10.5281/zenodo.16911348>
5. Burchell, H. and S. Marr (Oct. 2025c). “Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level: Towards Supporting Fine-grained Slowdown for Advanced Developer Tooling”. In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL’25. Singapore: ACM, p. 8. DOI: [10.1145/3759548.3763374](https://doi.org/10.1145/3759548.3763374)

Additional relevant work

- Marr, S., H. Burchell, and F. Niephaus (2022). “Execution vs. Parse-Based Language Servers: Tradeoffs and Opportunities for Language-Agnostic Tooling for Dynamic Languages”. In: *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 1–14. ISBN: 9781450399081. DOI: [10.1145/3563834.3567537](https://doi.org/10.1145/3563834.3567537). URL: <https://doi.org/10.1145/3563834.3567537>

Chapter 2

Background

This section provides the background relevant to this thesis, including profiling techniques, sampling and instrumentation, as well as known issues with profilers. We then discuss just-in-time (JIT) compilation with regard to the Graal compiler (Duboscq et al. 2013), x86, and VTune as a CPU instruction profiler. We will also detail the Are We Fast Yet benchmarks (Marr et al. 2016), as well as Rebench (Marr 2023).

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. The aspects of execution collected by a profiler depend on the tool and user configuration, and may include CPU, GPU, and memory usage, as well as system-level metrics such as hardware performance counters, among others.

Profilers can be utilized to identify code segments that can be optimized. Processor vendors such as AMD have developed profilers for their various products including AMD RGP¹ for AMD GPUs and AMD uProf² for CPUs, while Intel offers VTune.³ A number of other profilers exist, often specialised for particular use cases. For example, CodeGuru⁴ from Amazon uses machine learning to help improve applications running on Amazon services, while AppDynamics, built by Cisco, targets IT infrastructure monitoring. Different programming languages typically have their

¹<https://gpuopen.com/rgp/> (All URLs cited in this thesis were verified in August 2026.)

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/>

own specific profilers, too. Examples include ARMmap⁵ for C and C++, JProfiler⁶ and VisualVM⁷ for Java and dotTrace⁸ for .NET. These examples show that profiling is not constrained to a single type of hardware or a single purpose. Across these different domains, the core role of a profiler is to monitor and record some aspect of execution, then provide feedback that helps users understand what happened.

2.2 Sampling-Based Profilers

While we use sampling-based profilers in all chapters, this background is most relevant for the empirical study in [Chapter 3](#), and for determining the accuracy of profilers in [Chapter 5](#).

2.2.1 CPU Sampling

To collect a profile, a sampling profiler repeatedly observes what a program is doing at different points during its execution. At each sampling point, the profiler briefly interrupts or suspends a selected thread and records its current instruction pointer, i.e., the machine-code instruction being executed. It then attempts to unwind the thread's stack to reconstruct the current call chain. In Java code the profiler relies on metadata produced by the JIT compiler to map the sampled machine-code location to a Java method and bytecode index. Where line-number information is available, the bytecode index can then be mapped to a source line. By collecting many of these samples and grouping them by method or call stack, the profiler approximates the proportion of execution time spent in each part of the program.

The sampling profilers used in this thesis do not all perform this process in the same way. Async-profiler can use `perf_events` to trigger a sample after a configured time or hardware-event period.⁹ When the event occurs, Async-profiler uses `AsyncGetCallTrace`. `AsyncGetCallTrace` is an internal HotSpot function for reconstructing a thread's active Java call stack. It can inspect the thread at the point where

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/>

⁹https://man7.org/linux/man-pages/man2/perf_event_open.2.html

it is interrupted, without waiting for it to reach a JVM safepoint. It does this using information recorded when the thread was interrupted, including where it was executing, together with HotSpot's internal knowledge of Java stack frames and JIT-compiled code.¹⁰ JFR does not use `perf_events`. JFR instead uses its own dedicated thread to periodically select and suspend application threads before recording their stack traces.¹¹

Early forms of profiling Profiling dates back to the 1970s. For example, the IBM S/360 (IBM 1964) used program status words, which included 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 a profiling tool known as PROF (Thompson et al. 1974), one of the first tools for performance analysis, with the ability to output how much time each individual function took to execute.

This history is important because profiling is not a new or immature technique. Profilers have been developed, refined, and integrated into mainstream systems for decades. Modern examples, such as Linux `perf`,¹² are actively maintained and widely used as part of production performance analysis. We should therefore expect profilers to provide reliable insight into program execution. The concern motivating this thesis is that, despite this long history, modern profiling still struggles with the complexity introduced by highly optimising compilers and modern hardware.

2.2.2 Problems with Sampling-Based Profilers

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

¹⁰See the Async-profiler implementations of `perf_events` and stack collection at https://github.com/async-profiler/async-profiler/blob/master/src/perfEvents_linux.cpp and <https://github.com/async-profiler/async-profiler/blob/master/src/profiler.cpp>.

¹¹<https://github.com/openjdk/jdk21u/blob/jdk-21.0.2-ga/src/hotspot/share/jfr/periodic/sampling/jfrThreadSampler.cpp>

¹²<https://github.com/torvalds/linux/tree/master/tools/perf>

Sampling Overhead

Overhead is the additional time incurred due to profiling. In this work, we calculate it as the increase in total wall-clock time from the start of compilation to the completion of program execution, relative to the corresponding execution without profiling.

The amount of overhead varies depending on the profiler and the technique it employs. In general, sampling-based profilers tend to incur lower overhead, as they periodically interrupt execution and therefore only introduce cost at discrete sampling points. Consequently, their overhead is tied to the sampling rate, leading to a trade-off between the amount of information collected and the overhead introduced.

Overhead is a critical consideration when selecting a profiling approach. Low overhead is particularly important in production environments, where excessive overhead can reduce use cases and render profiling impractical. Furthermore, high overhead increases the risk of altering program behaviour, contributing to the observer effect (see [Sampling Observer Effect](#)).

We assess the overhead introduced by Java profilers in [Section 3.3.3](#).

Sampling Observer Effect

Profiling can change how a program executes (Seelemaann 1995), which is also called the *observer effect*. Attaching a profiler to an execution may alter its behaviour. In the worst case, the program's behaviour can change significantly, causing the recorded results to no longer reflect the behaviour of the unprofiled execution.

CPU sampling interrupts program execution, which can alter performance. For instance, it may affect CPU cache behaviour or interfere with multi-threaded execution (Orton et al. 2021). If the samples themselves change a program's performance behaviour, then they may report a region that, in an unprofiled run, would not contribute significantly to execution time. 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.

Sampling Randomness and Safepoint Bias

Sampling-based profilers rely on the assumption that instructions are sampled in proportion to the time spent executing them. Thus, sufficient samples will give a probabilistic view of where programs spend their time. *Safepoints* (Agesen 1998), sometimes called *yield points* (Alpern et al. 1999), are inserted into the program during compilation. They are required to enable operations such as garbage collection and ensure that execution reaches a well-defined state in which it is safe to interrupt a thread. Interrupting at safepoints ensures that the VM is in a state in which the stack can be read correctly; consequently, many profilers collect samples at, or in close proximity to, these safepoints. Unfortunately, sampling at safepoints can lead to inaccurate profiles, as shown by Mytkowicz et al. (2010). The compiler moves and eliminates safepoints to minimise the overhead they cause, which can restrict which parts of a program are visible to a profiler, as sampling may only be possible at these safepoints. As a result, profilers sample only a subset of program execution rather than at uniform intervals, leading to biased samples. This leads to profilers inaccurately attributing runtime across different parts of a program.

Reducing Safepoint Bias

Several of the profilers we use in this work¹³ aim to reduce the impact of safepoint bias. Honest Profiler¹⁴ and Async-profiler¹⁵ 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.¹⁶ Therefore, this approach does not fully eliminate safepoint bias, but only reduces it.

The async-profiler additionally collects stack traces with `perf_events`. `perf_events`¹⁷ is a Linux kernel interface that exposes hardware performance counters, enabling low-level profiling and providing insight into system and application behaviour at

¹³Which we evaluate in Chapter 3

¹⁴<https://github.com/jvm-profiling-tools/honest-profiler>

¹⁵<https://github.com/async-profiler/async-profiler>

¹⁶*Why JVM modern profilers are still safepoint biased?*, Jean-Philippe Bempel, <https://jpbempel.github.io/2022/06/22/debug-non-safepoints.html>

¹⁷https://man7.org/linux/man-pages/man2/perf_event_open.2.html

the kernel and hardware levels. 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 Recorder¹⁸ uses its own code, independent of `AsyncGetCallTrace`, to access the stack traces without requiring safepoints, but has similar limitations because it relies on nearby safepoints. 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 safely access memory. The HotSpot¹⁹ JVM supports the `-XX:+DebugNonSafepoints` flag to help ensure that metadata is available when execution is not in a safepoint, which can help with identifying the correct line number or bytecode index of a stack sample. However, this still relies on the compiler providing accurate and available metadata, which may not always be the case, particularly under aggressive optimisations.

Overall, only limited information is available for the top-most stack frame, which restricts a profiler's ability to precisely identify the executing code. As a result, safepoint bias is reduced but not eliminated in sampling profilers.

Non-determinism in JVMs

Multiple runs of the same Java program on the same JVM may result in different performance. This is because JVMs perform just-in-time compilation in parallel to program execution. For example, variation can arise when the compiler accesses profile-guided optimization information, that is, run time data such as execution frequencies and type information collected to guide optimisation decisions.

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.

Mapping from native to source code Sample-based profilers periodically halt execution to collect, for instance, the current instruction pointer. The instruction

¹⁸<https://openjdk.org/jeps/328>

¹⁹<https://openjdk.org/groups/hotspot/>

pointer can then be used to look up compiler-generated debugging information to determine the executing method. Profilers such as Async-profiler and JFR use the aforementioned `AsyncGetCallTrace` API²⁰ for this. When `AsyncGetCallTrace` encounters instructions without any debug information, it attempts to attribute them by identifying the nearest method boundary.²¹ Although this approach is accurate given correct debugging information, compiler optimizations may impact the correctness and completeness of debug information (Assaiante et al. 2023; Kell et al. 2024). For instance, compilers may fuse loops or merge methods into a single optimized operation, complicating or eliminating debug attribution entirely. JVM compilers also generate implementation-specific backend code, e.g., garbage collection barriers, which do not directly correspond to the source code, making instruction-level source attribution more complex. Kell et al. (2024) highlight that code reordering and transformation is important for performance, but can degrade the accuracy of debugging information, causing tools such as profilers to misattribute execution time and confuse identification of hot source lines.

2.3 Instrumentation-Based Profilers

Instrumentation-based profilers 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 et al. (1994) 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, making them available for program analysis. Instrumentation can be added automatically (Misurda et al. 2005) or manually (Romer et al. 1997), and can be performed either at compilation time (De Bus et al. 2004) or at run time (Luk et al. 2005).

²⁰See discussions: <https://jpbempel.github.io/2022/06/22/debug-non-safepoints.html>, <https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html>

²¹instruction pointer Fallback: <https://github.com/openjdk/jdk/blob/0460978e7c769624cacdb528277a99914b327e30/src/jdk.hotspot.agent/share/classes/sun/jvm/hotspot/code/NMethod.java#L287>

In [Chapter 4](#) we will evaluate JProfiler,²² VisualVM,²³ and YourKit²⁴ as state-of-the-art instrumentation-based profilers for the JVM.

2.3.1 Instrumentation Overhead

Instrumentation-based profilers are more likely to introduce higher overhead than sampling-based profilers, because they execute additional logic on every instrumented event rather than observing execution at intervals like a sampling based profiler. We evaluate this overhead in [Chapter 4](#).

2.3.2 Instrumentation Observer Effect

Any code instrumentation intrinsically alters program behaviour and may therefore change execution times. In addition to the direct cost of executing probes, instrumentation can also change how a JIT compiler optimises a program, for example by changing inlining or altering code layout (Zheng et al. 2015). As a result, the measured execution can differ from the execution of the uninstrumented program. We evaluate the observer effect incurred by instrumentation on the JVM in [Chapter 4](#).

2.4 Accuracy, Precision, and Realism

Accuracy and precision are often used interchangeably. However, we treat these as distinct concepts:

Accuracy measures how close the reported profiler results are to the ground truth, that is, what happens during execution.

Precision refers to how close measurements are to each other between runs. Thus, it measures how *consistent* a profiler is in producing the same extracted data points across repeated runs, regardless of whether those results are correct.

Realism is a measurement of a tool's observer effect, how closely the performance behaviour observed during a profiled execution resembles that of an equivalent

²²<https://www.ej-technologies.com/products/jprofiler/overview.html>

²³<https://visualvm.github.io/>

²⁴<https://www.yourkit.com/>

unprofiled execution that does *not* affect run-time optimizations. This therefore captures the extent to which a profiler changes the resulting performance behaviour, whether through instrumentation bias or through effects on compiler optimisations caused by the profiler's presence. For instance, we would consider a profiler *unrealistic* when it significantly changes the behaviour of specific optimizations, because the normal execution would not be subject to this change.

Therefore, a profiler can be accurate but unrealistic. This occurs when it correctly measures what happened during the profiled execution, but the presence of the profiler causes the performance behaviour of that execution to be significantly different from an execution without a profiler attached.

2.5 Graal Compiler

The Graal compiler (Duboscq et al. 2013; Würthinger et al. 2013) is a just-in-time (JIT) compiler, a system designed to perform compilation during program run time. JIT compilers leverage run-time information, for instance, about types and method calls, but also CPU and system-specific details which they use to optimize performance. Once the JVM identifies a *hot* method, i.e., frequently executed code, it compiles and optimizes it with the available information to achieve optimal performance.

2.5.1 Compilation Process

The Graal compiler uses two intermediate representations (IRs) before emitting final machine code. These can be understood as three main compilation stages: a higher-level IR, a lower-level IR, and the final generated machine code.

1. **Higher Intermediate Representation (HIR):** A higher-level abstraction of the Java bytecode enabling analyses and optimizations.
2. **Lower Intermediate Representation (LIR):** A lower-level abstraction closer to machine code enabling further optimizations and (for instance) register allocation.

3. **Machine Code:** The final output of the compilation process, tailored for execution on the specific hardware.

In the Graal compiler, the higher intermediate representation is a graph of nodes and edges, where nodes represent operations and edges indicate dependencies (Click et al. 1995; Duboscq et al. 2013). On this HIR, the compiler applies a range of optimizations, such as loop transformations, inlining, and dead code elimination. Additionally, while the program is represented in the HIR, Graal organises its optimisation phases into the High Tier, Mid Tier, and Low Tier. These tiers mark stages as the graph is progressively lowered towards machine code. An overview of the compilation pipeline is shown in Figure 2.1.

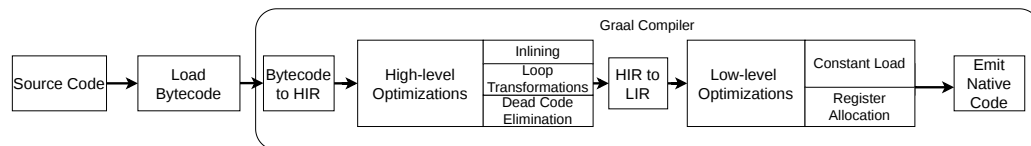


FIGURE 2.1: Overview of the Graal compilation pipeline, showing the progression from source code and bytecode through HIR and LIR to emitted native code. The diagram highlights where high-level and low-level optimisations are applied during compilation.

Graal LIR After the Graal HIR stage, the code is transformed into the LIR form. Here, the compiler organizes the code into blocks, similar to machine code basic blocks. These LIR blocks share key characteristics with basic blocks in that they have a single entry point and a single exit point. Once execution enters an LIR block, it is expected to execute all the LIR instructions sequentially until reaching a jump at the end, making LIR blocks a well-defined unit for analysis and instrumentation.

Compilation unit In Graal, a compilation unit is the highest-level organisation of code within a single compilation. It originates from one root method and may expand to include code from methods called by the root method when they are inlined. The root method and the code from its inlined callees are then optimised and compiled together.

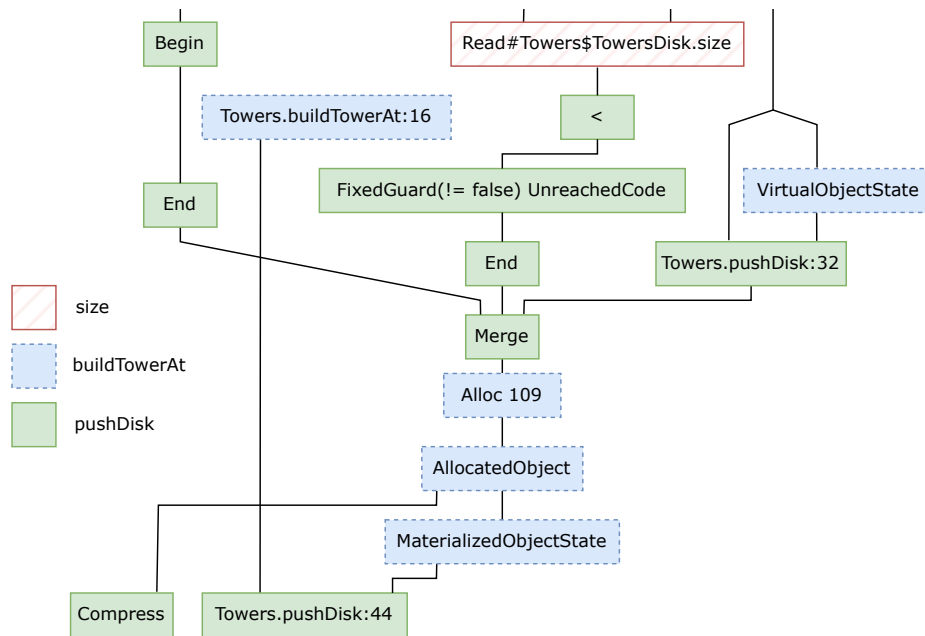


FIGURE 2.2: A Graal IR graph showing IR nodes originating from multiple inlined methods: *size*, *buildTowerAt*, and *pushDisk*. The coloured boxes identify the source method of each IR node, while the edges between nodes represent dependencies. The figure illustrates how compilation and inlining can result in operations from different source methods becoming intermixed within a single compilation unit, rather than forming distinct regions corresponding to individual methods.

2.6 Profilers and Inlining

Inlining replaces a method call with the body of the target method. This is a vital optimisation, as it enables optimisations to be performed across the combined caller and callee code. Graal does inlining at the HIR level. After inlining the nodes from a callee, subsequent optimisations can move and duplicate nodes which can lead to nodes from different methods ending up mixed together. Figure 2.2 illustrates this mixing for the Tower benchmark (Marr et al. 2016) with nodes from the *size*, *buildTowerAt*, 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 an instrumentation-based profiler, 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. They may also prevent optimizations, such as inlining (Machkasova et al. 2009). 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 2.2 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. A further discussion of the extent to which inlining leads to mixing of methods is given in Section 7.1.2.

2.7 Basic Blocks, Pipelining, and Out-of-Order Execution

Machine-code basic blocks A machine code basic block is a sequence of instructions that are executed sequentially, from start to finish, without any internal jumps or branches. The only exception is a single jump instruction at the end of the sequence, which determines the next block to be executed (Bakhvalov 2024; Hennessy et al. 2017). For our work, we rely on all instructions within a block being executed to keep our approach manageable.

Pipelining & OOO Modern processors utilize pipelining and Out-of-Order (OOO) execution to improve instruction throughput and overall performance. Pipelining allows multiple instructions to be processed simultaneously by breaking execution into distinct stages, such as fetch, decode, execute, and writing. OOO execution enables the processor to reorder instructions to maximize efficiency. The CPU can look ahead and determine whether future instructions share dependencies with currently executing instructions. If no dependency exists, the CPU can reorder independent instruction sequences to optimize execution and minimize idle cycles. However, this dynamic reordering complicates performance analysis, as the observed execution order may not match the program's original instruction sequence, making it challenging to attribute instruction-level execution time accurately in profiling.

2.8 VTune, A Profiler Using Hardware Event Sampling

Intel’s VTune²⁵ profiler is a run-time analysis tool for systems with Intel (and some AMD²⁶) CPUs. It monitors hardware and system usage through sampling and hardware counters, and can reveal instruction-level hotspots, identifying the specific machine instructions where execution time is spent. We use VTune’s hardware event-based sampling mode, which periodically samples performance counters that track metrics such as CPU cycles, instruction executions and instruction dispatch. These counters can be programmed for specific events (Bakhvalov 2024), such as cache misses, branch mispredictions, pipeline stalls. However, counters are *sampled* rather than recorded continuously, meaning they have limited granularity due to missing some events. They are also subject to *skid*, where delays in halting execution introduce mismatches, and out-of-order execution further complicates accuracy (Xu et al. 2019).

Precise Event-Based Sampling Some CPUs have access to precise event-based sampling (PEBS)²⁷ (Intel Corporation 2016 Vol. 3B, p. 18-57), which reduces the attribution error inherent in conventional hardware sampling by recording the architectural state at the instruction that triggered the sampled event. Unlike standard hardware sampling, where samples may suffer from skid and be attributed to nearby instructions, PEBS can provide more accurate instruction-level attribution, particularly in tight loops and vectorised code, because the CPU records the snapshot at the event rather than having an interrupt handler sample execution later. However, PEBS is not universally precise, because a hardware delay can bias samples towards certain instructions in tight, regular sequences (Yi et al. 2020).

2.9 Are We Fast Yet and ReBench

For chapters 3, 4, 5, and 6, where we evaluate profilers, we use the Are We Fast Yet benchmarks (Marr et al. 2016). We chose these benchmarks because they are well-understood and deterministic.

²⁵<https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>

²⁶Hardware Requirements - Supported CPUs <https://www.intel.com/content/www/us/en/developer/articles/release-notes/vtune-profiler/2026.html>

²⁷Sometimes called *Processor* Event-Based Sampling

TABLE 2.1: 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

Table 2.1 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 profilers by removing sources of variation.

However, choosing the Are We Fast Yet benchmarks also means that they are less representative of large Java applications. While benchmark suites such as Da-Capo (Blackburn et al. 2006) and Renaissance (Prokopec et al. 2019) 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 which allows Java code to call native C or C++ libraries, or other features that can be challenging for profilers due to

execution outside the JVM.

2.9.1 Benchmarking Execution Time

While each chapter uses a different number of executions, a common factor across all our Are We Fast Yet experiments is that we configured the benchmarks such that a single iteration takes about ~ 100 ms. We run each benchmark for 300 iterations in [Chapter 3](#) and 500 iterations in chapters 4, 5, 6. In this configuration, the benchmarks quickly reach a stable state, typically within the first second of execution, resulting in the majority of run time being spent at peak performance and producing more stable measurements. All profilers are attached at startup and begin profiling immediately.

To execute the benchmarks with and without profilers, we rely on ReBench ([Marr 2023](#)). ReBench adapts the system's settings to minimize interference and noise. ReBench automates benchmark execution and records profiler configurations to facilitate reproducibility. It also ensures a consistent system environment across runs. This helps reduce noise and system nondeterminism that might otherwise mislead the analysis.

2.10 Sampling-Based Profilers Evaluated in This Work

We evaluate the following sampling-based profilers across this thesis. Chapters use different subsets depending on the experiment, which we state where relevant.

Sampling-based profilers

- **Async-profiler**²⁸ An open-source project aimed at providing low-overhead sampling without safepoint bias.
- **Honest Profiler**²⁹ An open-source project that also aims for low-overhead sampling without safepoint bias.
- **Java Flight Recorder (JFR)**³⁰ Part of the OpenJDK, it allows the collection of run-time statistics and includes a sampling-based profiler.

²⁸<https://github.com/async-profiler/async-profiler>

²⁹<https://github.com/jvm-profiling-tools/honest-profiler>

³⁰<https://openjdk.org/jeps/328>

- **perf**³¹ A Linux tool that supports sample-based profiling of Java applications.
- **JProfiler**³² A commercial tool that supports both CPU sampling and instrumentation.
- **YourKit**³³ A commercial profiler designed to support low-overhead profiling.

³¹https://perf.wiki.kernel.org/index.php/Main_Page

³²<https://www.ej-technologies.com/products/jprofiler/overview.html>

³³<https://www.yourkit.com/>

Chapter 3

Inaccuracy in Sampling Based Profilers

3.1 Introduction

Given the complexity of modern computer systems, including dynamic compilation and modern hardware with its own level of optimisations, Mytkowicz et al. (2010) 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 on managed runtimes such as the JVM can produce biased or misattributed profiles because sampling is not uniformly possible across execution (for example, it may be constrained by safepoints), and aggressive JIT optimisations and incomplete metadata can make it difficult to reliably map optimised native code back to its source locations (Agesen 1998; Alpern et al. 1999; Mytkowicz et al. 2010).

Given all these issues, the wider adoption of profilers such as *async-profiler* and *Honest Profiler* that are designed to avoid safepoint bias, as well as the continued development in compilation since Mytkowicz et al.'s work, we wanted to know how precise and accurate today's profilers are, 16 years later.

The contribution of this chapter is an evaluation of the precision of sampling based profilers, examining how reliably they identify the same methods during execution and measuring the run time overhead of six actively maintained Java profilers, both

commercial and open source. This establishes, for this thesis, how accurate sampling based profilers are, to what extent we should trust their results, and what run time overhead they incur. We also discuss the factors that can cause sampling based profilers to be inaccurate.

To have a good intuition of what an accurate profile should look like, we chose the Are We Fast Yet benchmarks (Marr et al. 2016). Since they are relatively small, designed to be fully deterministic, and contain many classic benchmarks, it is easier to interpret and assess the resulting profiles than would be the case for larger benchmarks as in DaCapo (Blackburn et al. 2006) or Renaissance (Prokopec et al. 2019).

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. (2010) 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 between runs. YourKit shows the largest median difference among the profilers we evaluated, at 8%.

When comparing the profiles among profilers, we unfortunately find that they often disagree with each other. This disagreement appears both for the first hottest method and for top five hottest methods reported by each profiler.

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* provide very similar profiles for all but two benchmarks. We discuss this in detail in [Section 3.4.2](#).

The overhead of profiling is on average 1.0% to 5.4% depending on the profiler.

3.2 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.

3.2.1 Research Questions

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

RQ1: Do sampling profilers give precise results? As argued in [Section 2.2.2](#), 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 precise, i.e. how close measurements are to each other between runs. Given the complex systems we are working with, accuracy cannot be assessed directly. Even when we design workloads where we expect where time should be spent, dynamic optimisations in both software and hardware make it difficult to quantify the true performance behaviour precisely.¹

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.

¹In [Chapter 5](#) we show that it is possible to approximate a reference performance behaviour, allowing us to reason about accuracy relative to a known performance behaviour rather than an unknowable ground truth. But for this chapter, instead we focus on whether the results are precise. This means, we expect measurements to be tightly grouped.

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.

3.2.2 Experimental Setup

For our experiments, we chose six profilers, evaluating them on the Are We Fast Yet benchmarks (Marr et al. 2016), and performed the following high-level steps for each profiler:

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 2.2.2](#)), and other external influences. Across these runs, 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.

3.2.3 The Chosen Profilers

For our experiments, we chose six actively maintained profilers (see [Section 2.10](#)). 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 safepoint bias as identified by Mytkowicz et al. (2010). Since the Java Flight Recorder (JFR) 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. (2010), but they are unfortunately no longer actively maintained.

3.2.4 Benchmarks

All benchmarks were executed on a machine with Rocky Linux 9, and a Linux kernel version 5.14.0. The machine has an AMD Ryzen 5 3600 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 representation and then method names are normalised² to enable comparison.

²Profilers format method names differently. For example, some include the fully qualified class name, such as `package.Class.method`, while others append parentheses to denote a method. We therefore

3.3 Results

In this section, we present results on profiler precision across repeated runs, agreement between profilers on the hottest methods, and runtime overhead.

3.3.1 Self-Comparison of Profilers

We first assess a profiler's precision and reliability.

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 the average run time of the program. This minimises the noise in the results and likely represents what developers are interested in when trying to optimise a program.

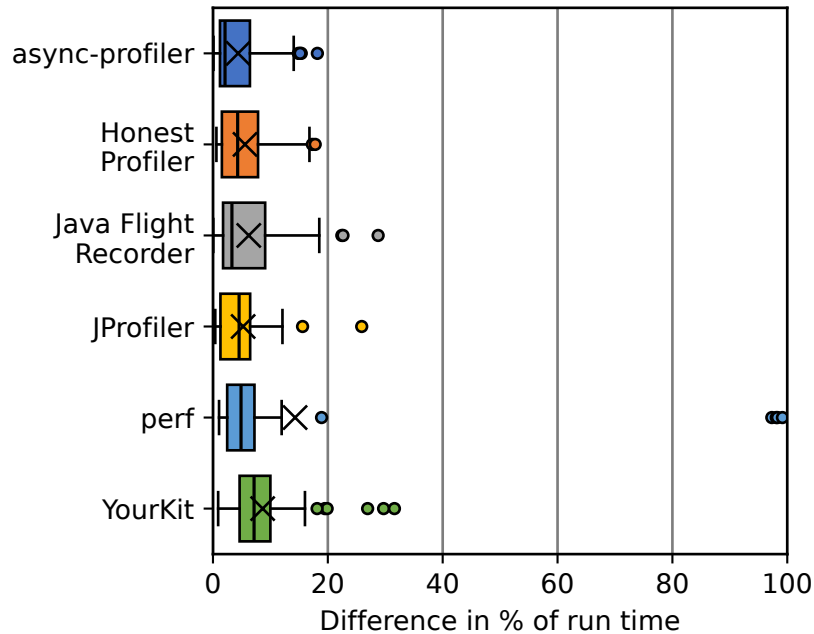


FIGURE 3.1: Difference in % of run time for every method, for each profiler. The boxplot shows the difference between the minimum and maximum percentage of run time for each method. The $\times$ indicates the mean. The median difference is between 2 and 8%, suggesting a high precision.

handle each profiler's output format separately and extract a consistent representation of each method name.

Figure 3.1 shows the distribution of differences per profiler. For instance, a value of 10% means the difference between the minimum and the maximum reported 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 four outliers methods within the NBody and Permute benchmarks that show a difference of 99%. 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.

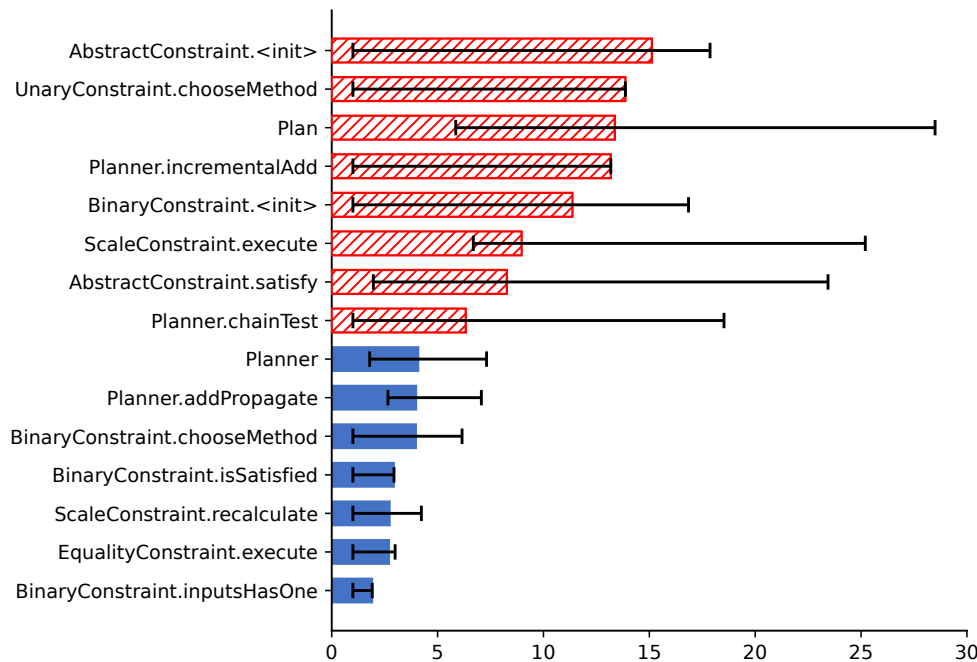


FIGURE 3.2: Bar chart for all methods in the DeltaBlue benchmark identified by Java Flight Recorder. A bar represents the average percentage of run time for a method, with error bars being the minimum and maximum values found for that method. Striped bars are methods that were the hottest method at least once. The large error bars representing the minimum/maximum show that methods can easily switch places in the ranking.

To investigate the impact of the difference, we created plots for each benchmark

showing the average run time attributed to a method. [Figure 3.2](#) shows a worst-case example, 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 runs, which was the case for 10 different methods on this benchmark. In addition, the large minimum/maximum ranges for these methods suggest that we cannot fully rely on profiles from Java Flight Recorder on DeltaBlue to guide optimisation decisions.

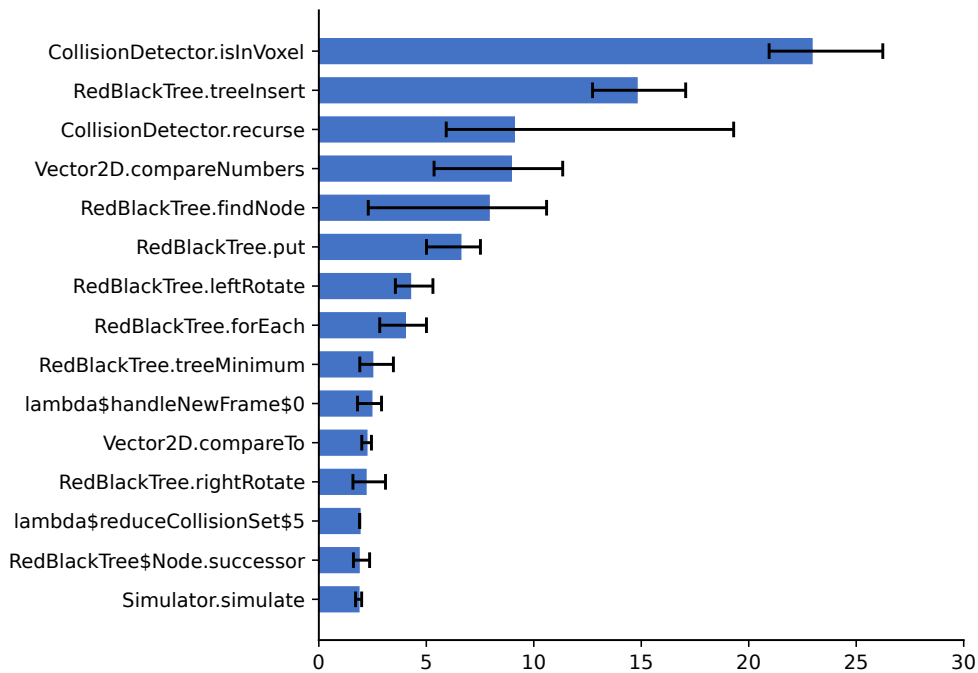


FIGURE 3.3: Bar chart for all methods in the CD benchmark identified by YourKit. A bar represents the average percentage of run time for a method, error bars are the minimum and maximum values found for that method. The large range between minimum/maximum shows that runs can be significantly different from one another.

As a second example in [Figure 3.3](#), 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, 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 rather than relying on a single result to guide optimisation.

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 3.1](#).

TABLE 3.1: 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.

Profiler	Hottest Methods		
	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

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 runs for that profiler, the same method was identified as hottest. Since the numbers are aggregated across all benchmarks, the median indicates that the profilers generally identify

the same hottest method across repeated runs. The maximum number of methods identified as hottest suggests that JProfiler and YourKit are having the fewest 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 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 run time, 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 3.4](#).

Each method plotted was identified as the hottest method at least once in all of the 30 runs. [Figure 3.4](#) plots the percentage of run time over the median run time of iteration for that profiler run. The method with the highest percentage of 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 3.1](#), 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.

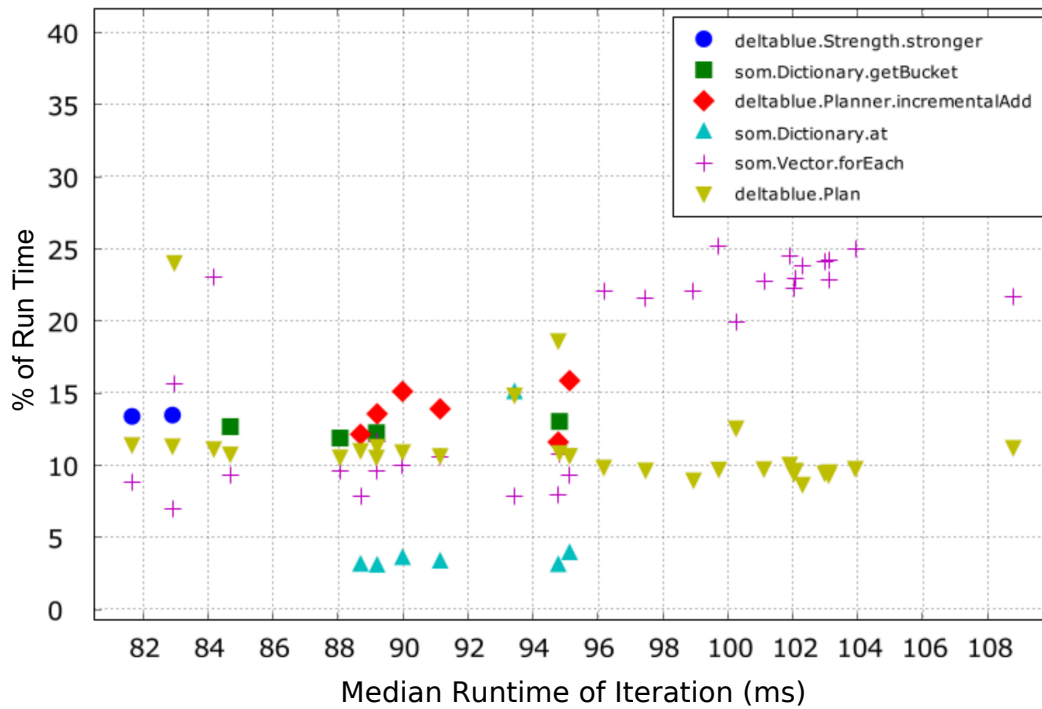


FIGURE 3.4: Percentage of run time over median run time for the six hottest methods for the DeltaBlue benchmark profile from async-profiler. We observe multiple changes in what the first hottest method is (the furthest up) for each iteration.

3.3.2 Comparisons Across Profilers

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.

Average Comparison

Looking back to [Figure 3.1](#), 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.

Unions

Mytkowicz et al. (2010) compared the cardinality of the union of the first hottest method over all profilers, which means they counted how many different methods they see. We will use their approach, too. As mentioned in Section 3.2.1, assessing the accuracy of a profiler is not directly possible, a challenge we overcome in Chapter 5, 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.

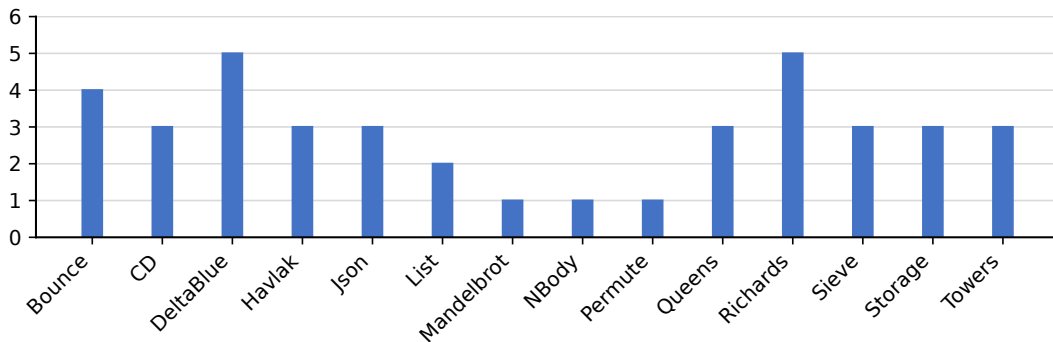


FIGURE 3.5: Union₁, i.e., count of different first hottest methods across all profilers.

When looking at the results in Figure 3.5, 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 with at least one other profiler, 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 3.6 shows the results for the union for the 5 hottest methods. Here we

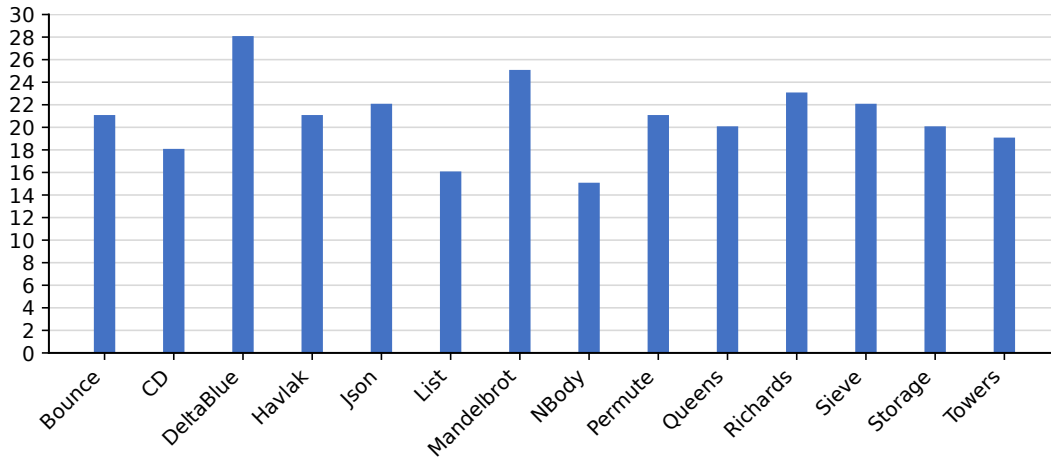


FIGURE 3.6: Union₅, i.e., count of different methods reported for the five hottest methods across all profilers.

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.

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 3.6](#) 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 3.2](#), there is high imprecision on this benchmark resulting in all but two pairs of profilers agreeing on a method.

Mandelbrot should only have two hot methods (see [Table 2.1](#)). Although only two methods dominate Mandelbrot, the union of the six profilers' top-five lists contains more than 12 distinct methods, including methods involved in producing output and parsing bytecode.

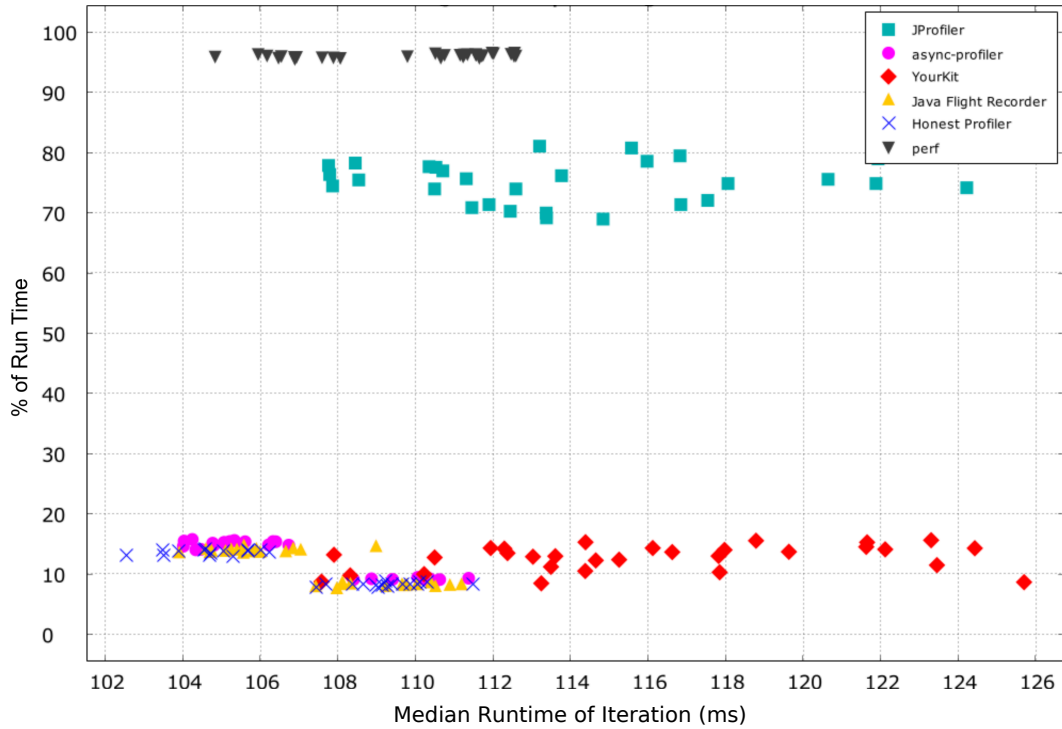


FIGURE 3.7: Percentage of run time against median iteration run time for the method `Queens.placeQueen`, for all 6 profilers. We observe clearly different amounts of percentage of run time between many profilers, showing clear disagreement.

To answer RQ2, i.e., how reliable profilers are at identifying the same hottest methods, [Figure 3.5](#) shows that profilers reliably identify the first hottest method for few benchmarks, and there are benchmarks where six profilers report five different methods. When considering [Figure 3.6](#), reliability further decreases when comparing the 5 hottest methods even without taking ordering into account.

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.

Our first example in [Figure 3.7](#) 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.

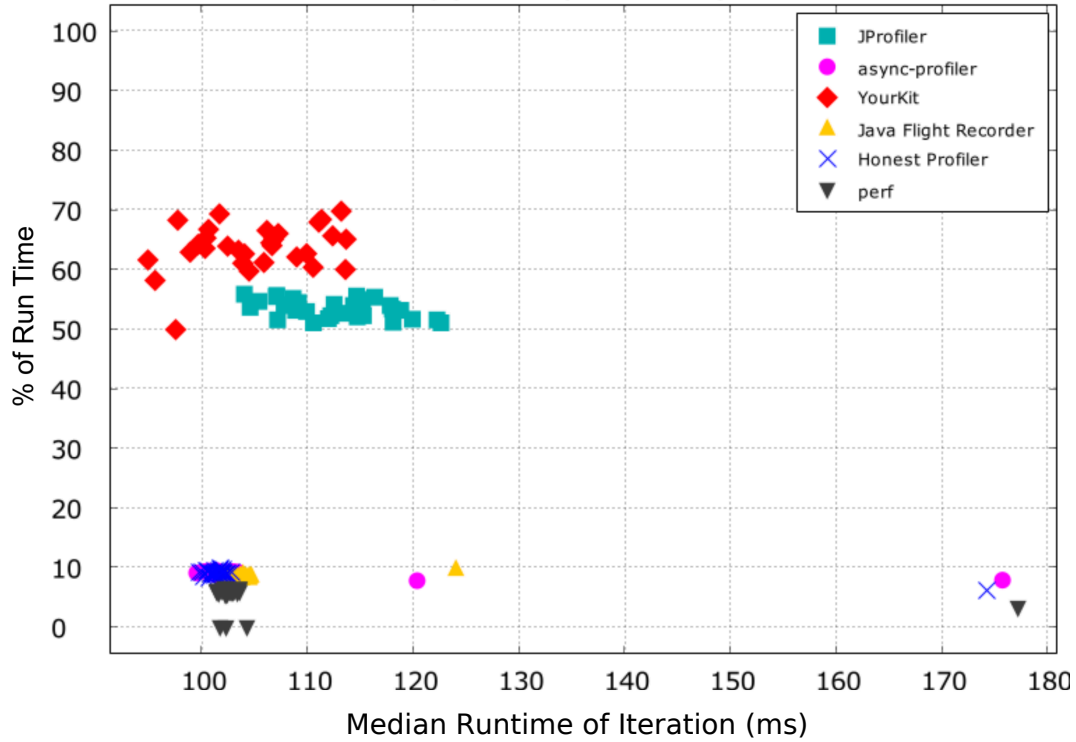


FIGURE 3.8: Percentage of run time over median iteration run time for the method Bounce.benchmark for all 6 profilers. We observe clearly different amounts of percentage of run time between many profilers, showing clear disagreement

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 3.8 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 reported percent

of run time.

perf (▼) is in close agreement with the other three profilers, but YourKit (◆) and JProfiler (■) attribute 50% to 70% 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 3.7 and Figure 3.8 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.

3.3.3 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 3.9 shows the distribution across all 14 benchmarks as a boxplot for each profiler.

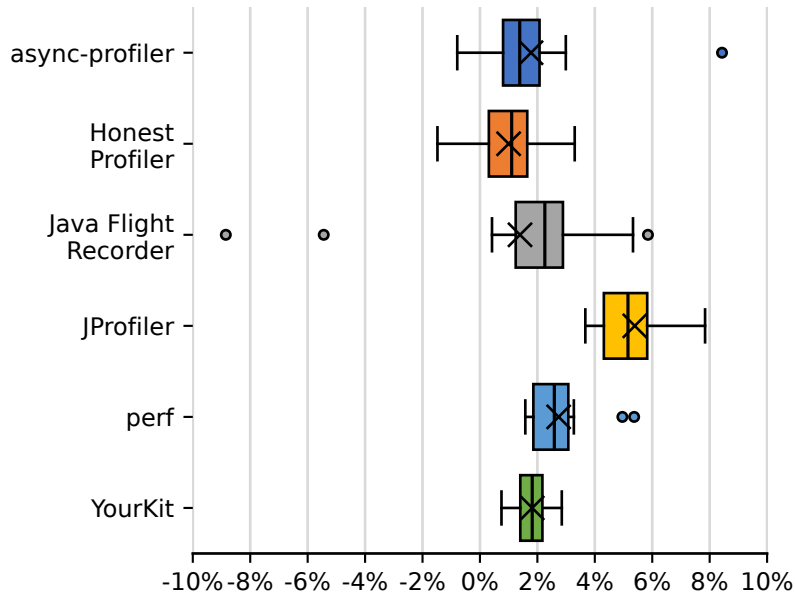


FIGURE 3.9: The average percentage of run time overhead induced by each profiler. The × indicates the mean.

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 3.1](#). 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 (Orton et al. [2021](#); Sjoblom et al. [2011](#)).

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.

3.4 Discussion

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

3.4.1 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 3.2](#) 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%.

To investigate these differences, we compare the runs with one another in more detail. [Figure 3.10](#) 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 iteration. [Figure 3.10](#) 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

TABLE 3.2: 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%

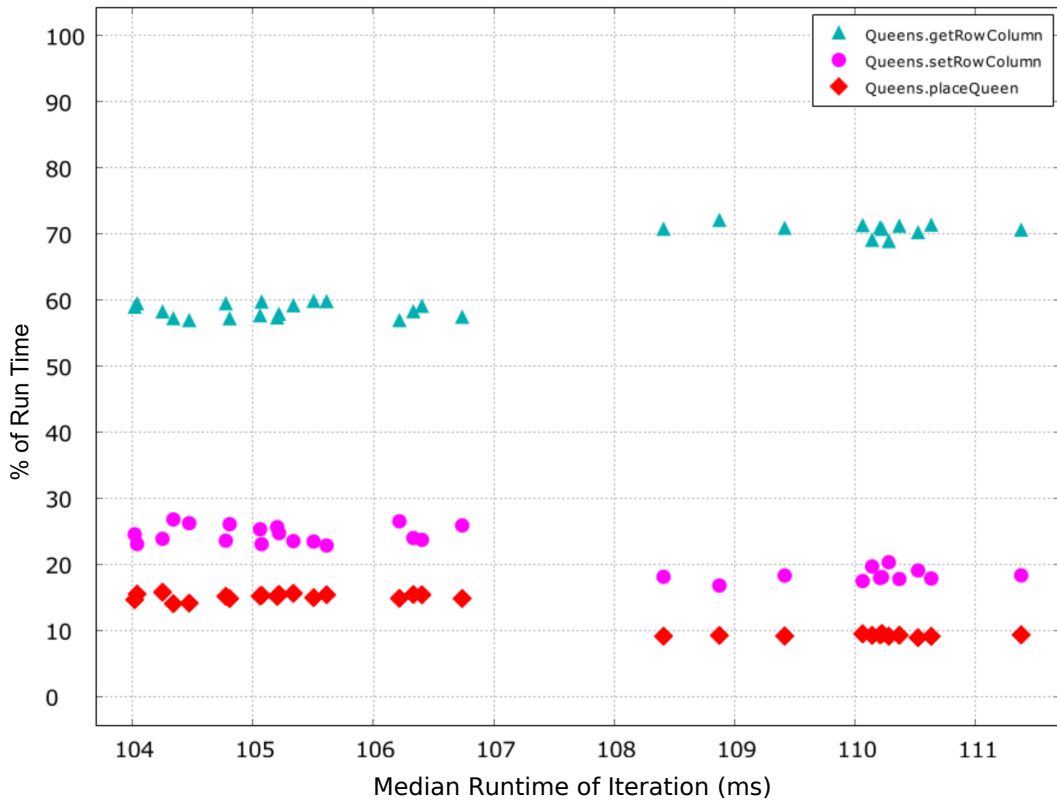


FIGURE 3.10: Percentage of run time over median run time of an iteration for the three hottest methods in the Queens Benchmark with async-profiler. This displays a bimodal trend.

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.

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 3.7](#), 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.

3.4.2 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.

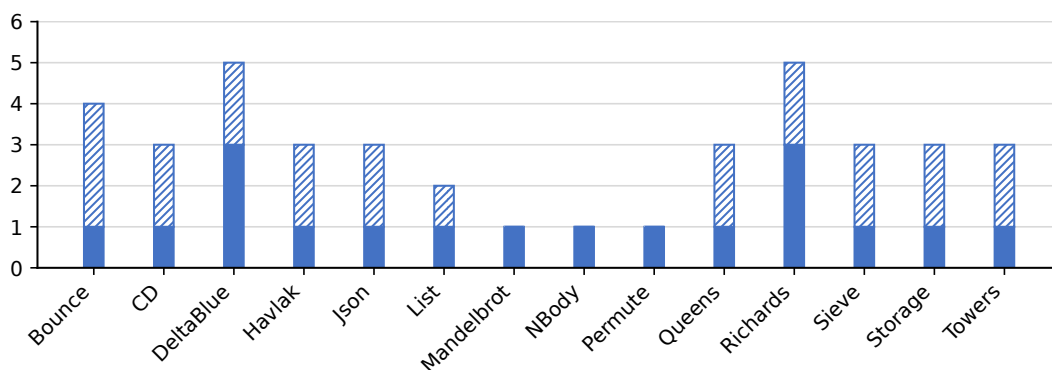


FIGURE 3.11: Union of first hottest method across all profiles, highlighting the agreements between `async-profiler`, Java Flight Recorder and Honest Profiler (solid blue)

[Figure 3.11](#) features the same results as [Figure 3.5](#), with an added detail: we establish a distinction between these three profilers and the three others. The portion of the bars that 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 3.3.1](#)), there is also no agreement among each other here.

3.4.3 Workload with Expected Profile

In other work, specific workloads are crafted for which the expected profile is known (Berger et al. [2023](#)). 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. However, in [Chapter 5](#), we explore an approach where the expected profile is known, giving us an approximated ground truth for what the profiler should report.

3.5 Related Work

In addition to comparing to Mytkowicz et al. ([2010](#)) in more detail, we also briefly discuss work on optimisations of profiling techniques, as well as open source work.

The work of Mytkowicz et al. ([2010](#)) 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 sampling independent of safepoints.³ Nonetheless, when comparing unions of agreement of profilers,

³But their retrieval of debug information can still rely on safepoints; see [section 2.2.2](#)

we found similar disagreement as reported by Mytkowicz et al. (2010). In some ways, the situation may have gotten even worse, since they reported a very small 95% confidence interval around the mean run time they reported for 30 runs.

Hofer et al. (2015) 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 identify no major accuracy issues.

A paper on a profiler for Python (Berger et al. 2023) 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 used a version of CPython which only has an interpreter, we cannot directly compare the overhead they report either.

Outside 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,⁴ who reported on technical details of profiler, limitations, and discussed the underlying issues of safepoint bias.

Similarly, Alexey Ragozin⁵ 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.

3.6 Conclusion

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,

⁴*The Pros and Cons of AsyncGetCallTrace Profilers*, Nitsan Wakart, <https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html>

⁵*Lies, darn lies and sampling bias*, Alexey Ragozin, <https://blog.ragozin.info/2019/03/lies-darn-lies-and-sampling-bias.html>

for instance caused by the safepoint bias observed by Mytkowicz et al. (2010), 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 (Marr et al. 2016) 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 on average 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 experimental setup used in this chapter is available as open source⁶ and can be extended to include additional Java profilers.

In the next chapter, we turn to instrumentation-based profilers. We examine how accurately they report execution time, how much overhead they introduce, and how they affect compilation. We also introduce our own instrumentation-based profiler, which is designed to evaluate and address the issues identified in next chapter.

⁶<https://github.com/HumphreyHCB/AWFY-Profilers>

Chapter 4

Alleviating Inaccuracy in Instrumentation-Based Profilers with Late-Stage Compiler Instrumentation

4.1 Introduction

In the previous chapter ([Chapter 3](#)), we showed that sampling-based profilers can produce inconsistent results across repeated runs, and can also disagree with other sampling-based tools when profiling the same program. The most likely causes are differences in profiling techniques, as well as the effects these tools can have on compilation behaviour.

Instrumentation-based profiling of just-in-time-compiled programs affects execution, reducing the precision and accuracy of profiling results as well (Machkasova et al. [2009](#); Mytkowicz et al. [2010](#); Zheng et al. [2015](#)). Instrumenting code at the source or bytecode level changes how the code is optimized (Machkasova et al. [2009](#)), leading to results that may not be representative of the actual performance of an application.

To better understand a known challenge with instrumentation-based profilers on the Java Virtual Machine (JVM), and to take a first step towards reducing it, we assess the state of the art in [Section 4.2](#). We measure runtime overhead, the impact instrumentation has on compilation, and whether instrumentation-based profilers

yield unrealistic results. We find that the overhead is typically one or two orders of magnitude, and the results can be considered wildly unrealistic.

As a solution, we propose a new approach to instrumentation-based profiling integrated into the 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. (2023), 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 on how the compiler optimizes the code. To reduce the run-time overhead of the instrumentation, we use two different types of probes. For compilation units that are smaller than the cost of probes that measure elapsed CPU cycles, we only count activations, and estimate the elapsed cycles based on Graal's cycle estimates, which are informed by the JVM's run-time profiling. Furthermore, we attribute the time during which a compilation unit executes the remaining fragments of inlined methods, instead of merely attributing it to the root method. Here, the root method is the method being compiled and the outermost caller in the compilation unit. Methods called by it, and potentially their callees, may be inlined into its compiled code.

BuboC, our implementation of late-compiler-stage-based instrumentation, is evaluated using the Are We Fast Yet (AWFY) benchmarks (Marr et al. 2016). The results show that our approach has significantly lower overhead compared to state-of-the-art instrumentation-based profilers for Java. We achieve a median overhead of 23.3% (min. 1.4%, max. 464%) when we instrument all JIT compiled compilation units.

To reduce the overhead of our tool we introduce an approach of estimating the elapsed cycles for small compilation units instead of fully instrumenting them. For the AWFY benchmarks, a threshold of 80 cycles yields a good tradeoff. Here we see 0.37% (min. 0.01%, max. 46.54%) of cycles being estimated in the median for the 9.6% (min. 0.1%, max. 187%) overhead. This gives us a good balance between low overhead with only a minimal number of cycles being estimated for most benchmarks.

We also show that our instrumentation-based approach yields more precise, i.e., more reproducible results than sampling profilers. When compared to five Java sampling-based profilers, our approach has the lowest median change of run-time percentage between 30 runs of a benchmark, achieving a median of 0.6% difference in

percentage of run time, whereas the best sampling profiler achieves a median of 1.4%.

This chapter establishes why state-of-the-art JVM instrumentation-based profilers are untrustworthy in the presence of JIT compilation. We show that early bytecode instrumentation can both impose large overhead and change compilation behaviour, which in turn makes the resulting profiles unrepresentative of uninstrumented execution. Using this information, we investigate late-compiler-stage instrumentation as a way to reduce observer effect by moving probes to a point where key optimisations, such as inlining and loop transformations, have already happened, and we explore how to keep such instrumentation practical by selectively estimating regions where probe cost would dominate. This results in an instrumentation approach that reduces overhead and maintains realistic program behaviour when instrumented.

4.2 State of the Art in Instrumentation in JIT Compilation

To build toward motivation for our work with late-compiler-stage-based instrumentation. We begin by analyzing current state-of-the-art instrumentation-based profilers to characterize them in the context of JIT compilation.

4.2.1 Research questions

For evaluating state-of-the-art instrumentation-based profilers in a JIT compilation setting, we define two research questions.

RQ1: What is the run-time overhead of a state-of-the-art instrumentation-based profiler? We will measure the overhead of JVM instrumentation-based profilers. This will give us an idea of the impact of the profilers on the overall behaviour of a program and helps determine whether these tools remain practical for repeated benchmarking.

RQ2: Do state-of-the-art instrumentation-based profilers give unrealistic results? Instrumentation requires placing probes within the code, which can alter its performance behaviour significantly. To assess whether the instrumentation changes the performance properties in a way that makes them unrealistic, we measure whether

disabling inlining causes a difference. Since inlining is essential to enable subsequent optimizations, disabling it is a good proxy to assess whether the instrumentation added to the program interferes with optimizations, and thus makes profiling results less realistic. Additionally, we will assess the impact of disabling inlining on sampling profilers to compare with profilers that do not change the program's code.

Instrumentation-based Profilers The instrumentation-based profilers we evaluate are the following:

- **JProfiler:**¹ A commercial tool, which we use in its instrumentation mode.
- **VisualVM:**² An open-source tool, which we use in its instrumentation mode.
- **YourKit:**³ A commercial tool that we use in its instrumentation-based mode.

4.2.2 Assessing Run-time Overhead

Instrumentation-based profiling typically incurs a higher overhead than sampling-based profiling.

To assess the extent of the overhead as part of RQ1, we measure for each profiler its default settings using full instrumentation, where every method in the program is instrumented, which 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 profiler 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 to run all experiments. Instead, we ran only the DeltaBlue benchmark, whose overhead is roughly in line with the other profilers, and we report results for it where relevant.

Figure 4.1 reports median run time and overhead over the uninstrumented execution of all benchmarks. While it is to be expected that fully instrumenting a program

¹Version 14.0.2. <https://www.ej-technologies.com/products/jprofiler/overview.html>

²Version 2.1.8. <https://visualvm.github.io/>

³<https://www.yourkit.com/>

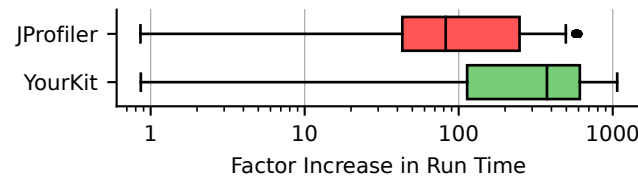


FIGURE 4.1: Overhead of instrumentation-based profilers for the Are We Fast Yet benchmarks. The overhead is expressed as a factor over the uninstrumented run time. The median runtime factor increase is $82\times$ for JProfiler and $374\times$ for YourKit.

will result in high overhead, the extent of this overhead can be substantial enough to render the use of a profiler impractical and uninformative. Any instrumentation brings the risk of changing which optimizations a compiler will perform, which in turn means any measurements made by the instrumentation risk not being representative of normal program execution. For VisualVM, which is not in the figure, profiling DeltaBlue causes a $485\times$ longer run time. This overhead is similar to the median overhead we found for YourKit and JProfiler. To answer RQ1, instrumentation-based profilers impose extremely high run-time overheads, with median run-time increases of $82\times$ for JProfiler and $374\times$ for YourKit. These overheads are likely impractical for many benchmarking use cases, and we will go on to examine its impact on both compilation and the realism of the reported results from these tools.

4.2.3 Assessing Impact on Compilation

To understand why these profilers incur such high overhead, we assess how instrumentation affects the compilation of benchmarks. First we will look at the overall amount of code generated, and then we will investigate how instrumentation affects inlining. The data for the analysis is extracted from Graal's compilation log, which is enabled with the flag `-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 the Are We Fast Yet benchmarks.

The very large overhead we saw in Figure 4.1 is likely due to the instrumentation process itself. The increased size of the native code seen in Figure 4.2 suggests that the probes cause the additional code and that the compiler is unable to optimize the instrumented code effectively.

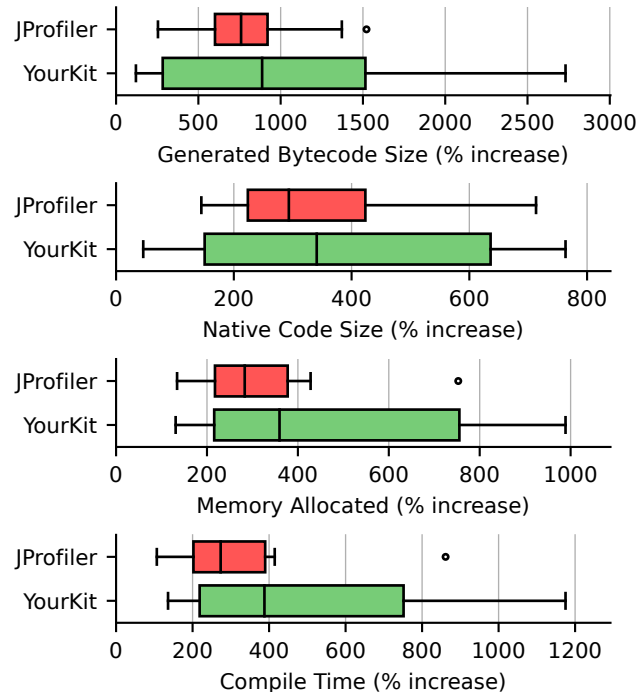


FIGURE 4.2: Percentage increase of the amount of bytecodes, native code size, memory allocation, and compile time for attached instrumentation-based profilers on the Are We Fast Yet benchmarks.

When YourKit is attached, the added instrumentation causes the native code size to increase by a median of 341%, which contributes to the overhead of $374\times$. 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 (Machkasova et al. 2009). The increased code size and the possible changes to optimization decisions suggest that any results obtained from such profilers are likely less realistic than for other types of profilers.

Impact of instrumentation on inlining To answer RQ2, we measure how inlining changes when instrumentation-based profilers are attached. The data for Figure 4.3 was collected from Graal’s inlining log while profiling the Are We Fast Yet benchmarks.

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 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 code. At the same time, the compiler is not able to remove the instrumentation, because it is indistinguishable from normal application code. As a result of the instrumentation, the optimized program behaviour can resemble the behaviour observed with a profiler that simply counts method activations. Arguably, this is a strong indication that the profiled behaviour will likely not resemble the normal program behaviour any more. This makes profiling results “unrealistic”, since normal optimizations do not have the same impact anymore.

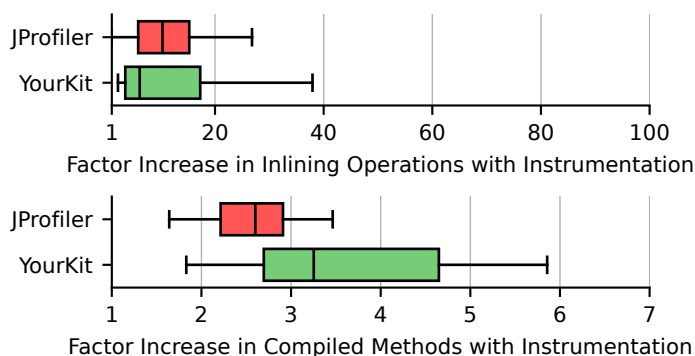


FIGURE 4.3: The increase in compiled methods and inlining caused by attached instrumentation-based profilers for AWFY benchmarks. The increase is expressed as a factor compared to the results of uninstrumented executions.

To understand better how much instrumentation changes the benchmark behaviour, we ran the DeltaBlue benchmark with each instrumentation-based profiler, once normally, and once with inlining disabled.⁴ We would expect a drastic change in program behaviour with inlining disabled. 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 behaviour as much as instrumentation.

Table 4.1 shows the results for Async and JProfiler. The full results are in the appendix in Table A.2. These tables show that the profiles with and without inlining are close to identical for the instrumentation-based profilers. For Async, the sampling

⁴Inlining is disabled with the Graal compiler flag `-Dgraal.Inline=false`

TABLE 4.1: Profiling results for Async and JProfiler, with and without inlining.

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

profiler, this is however not the case. Here enabling inlining drastically changes the profile as one 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 three methods: `deltablue.Plan`, `Vector.forEach`, and `ScaleConstraint.execute`. This suggests that inlining and subsequent optimizations change the observable behaviour 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.

Instrumentation bias caused by high activation counts Table 4.1 suggests that these profilers do not accurately reflect the optimised program that would run when

uninstrumented. One plausible explanation is that the inserted instrumentation biases the measurement towards “how often” code is executed, rather than “how much work” the optimised code performs. In other words, once probes are present everywhere, the profile can become dominated by the cost and frequency of probe execution. This would also explain why disabling inlining has little visible effect for the instrumentation-based profilers: the profiler’s own probes, and their activation, can dominate the profile and effectively drown out the optimisation differences that Async still observes.

This creates an issue in which the resulting profile begins to correlate with activation counts rather than performance behaviour. Frequently executed code, such as small methods, can appear important because it is instrumented and therefore expensive to execute repeatedly, even if, in the uninstrumented program, the compiler would inline or optimise it away to the point where it is not a meaningful optimisation target. As a result, the profile can provide guidance that is at best unhelpful and at worst misleading.

Figure 4.4 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.

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

FIGURE 4.4: Worst-case scenario for instrumentation-based profilers. Highest activation counts may misguide optimization efforts.

Instrumentation-based profilers are unrealistic With this, we conclude for RQ2 that instrumentation is very likely not providing accurate results and that the probes change the application code to such a degree, that the run-time behaviour becomes unrealistic. This means, any optimization decisions made based on these profiling

results may not have the desired effect in normal execution, when the compiler can more freely optimize the program without interference from probes.

4.3 BuboC: Late-Compiler-Stage-based Instrumentation

To try to alleviate the issues identified in [Section 4.2](#) we have implemented our own instrumentation-based profiler BuboC. In this section, we will detail how late compiler-stage instrumentation works, how BuboC is implemented, and how we prevent over-profiling and account for inlining.

4.3.1 Late-compile-stage Instrumentation Overview

Late-compiler-stage instrumentation inserts instrumentation probes into the program late during JIT compilation. By delaying the insertion of probes as much as possible, our approach minimizes the observer effect and run-time overhead: we add probes as late as possible to ensure most compiler optimizations are already applied and thus not disturbed, but as early as needed to achieve correct semantics.

Once the JVM selects a method for JIT compilation, the Graal compiler will compile and optimize it through its many phases, each performing specific optimizations (see [Section 2.5](#)). Our implementation is split into two such phases. 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 all methods that have been inlined, as well as to prepare the lookup of a memory address for our second phase. At this point, however, we do not insert any instrumentation nodes, which avoids interfering with optimizations.

In the low tier, as late as possible, we add our instrumentation nodes. These are nodes to record the CPU cycles at the start and each exit from the root method. It also includes nodes to compute the difference between the two cycles counts and to add it to the sum of cycles taken by this method, or more correctly the compilation unit. The sum is stored at the memory address for which we prepared the lookup in our first phase. This means, the update operation will be to a memory address known at JIT compilation time, which minimizes the overhead, compared to alternatives such as looking up the address at run time. The value is stored in a buffer, with a separate

location for each compilation unit, which is processed to compute and output the overall profile right before the JVM shuts down.

Since we instrument code as part of JIT compilations, only methods that are compiled will be instrumented. This means that we can neither collect profiling information during application start and warmup, nor for code that is not compiled. However, for most applications, the methods relevant to performance are invoked frequently and are therefore JIT compiled, so this is only a minor restriction in most use cases.

4.3.2 Implementation of the Low-tier Instrumentation

BuboC (Bubo Compilations) is our late-compiler-stage-based instrumentation with a focus on instrumenting whole compilation units, hence the “C”. We build in [Chapter 6](#) an alternative late-compiler-stage-based instrumentation called BuboL (Bubo Loops).

As mentioned previously, the most important component of BuboC is the low-tier phase that we added to the Graal compiler. We start by finding the most relevant compiler nodes in the Graal IR graph, which are the `StartNode` and the `ReturnNodes` belonging to a compilation unit.⁵

To record how long a method compilation unit is activated, we insert our custom `ClockNode` immediately after the `StartNode` and right before each `ReturnNode`. In this context, “before” and “after” refers to the node position in the control flow graph maintained by Graal. When Graal emits the native code, our `ClockNode` produces a RDTSC (Read-Time-Stamp-Counter) instruction, which reads the number of CPU cycles from a hardware register. To process the CPU cycles, we also insert nodes to calculate the difference between start and end cycles and add the result to our buffer right before the method’s return nodes. For this, we rely on existing Graal IR nodes.

Additionally, BuboC has the ability to instrument calls within a compilation that invoke non inlined methods. The time spent in these calls can be subtracted from the root compilation, ensuring that we record only the time spent in the instrumented compilation unit itself.

Our overhead data, presented later in [Section 4.5.1](#), is collected using the version of BuboC that instruments calls from a compilation unit into non-inlined methods, so

⁵We also handle `UnwindNodes` where present.

that time spent in such calls can be accounted for separately from the root compilation. All other results in this chapter use a version of BuboC that does not instrument non-inlined calls. We explicitly note whenever results are obtained with non-inlined call instrumentation. The reasoning for why we do not instrument calls in all experiments will be discussed in [Section 4.6](#).

The buffer is implemented as a long array. Our instrumentation writes to this array using the unique compilation unit ID as the index. Each entry stores the sum of CPU cycles during which that compilation unit was active. We additionally maintain a mapping from each unique ID to the methods it represents, allowing us to attribute cycles correctly to the corresponding source-level methods, as detailed in [Section 4.3.4](#). Together, this information enables us to determine which methods consume the most CPU cycles.

4.3.3 Reducing Profiling Overhead for Small Methods

When the cost of a probe is comparable to, or larger than, the work performed by a method, the profile becomes dominated by instrumentation cost rather than program cost. In such cases, highly activated but inexpensive methods can appear artificially important, even though optimising them would not yield meaningful speedups. One reason for the higher overhead of instrumentation-based profilers is therefore that their probes can take more time to execute than the method itself. For example, in a trivial method that simply adds one to an integer argument, the computation itself takes only a handful of CPU cycles.

For a compilation unit with no calls or with all calls inlined, our instrumentation takes approximately 80 or more CPU cycles per activation of a compilation unit: our instrumentation executes one RDTSC instruction to read the current CPU cycles at the beginning, one at the end, a memory read, an addition, and a memory write. Each RDTSC instruction takes about 40 cycles depending on the processor architecture ([Fog 2022](#)). This means, a trivial method with instrumentation would require many times more CPU cycles than its uninstrumented original. If the method is “hot”, i.e., frequently called, this can cause our instrumentation to cause significant overhead.

To minimize this instrumentation overhead for small methods, we implemented a second, more lightweight type of probe. Instead of making two RDTSC calls, we simply count how many times the method is activated. While this means that we do not measure all run-time behaviour accurately, we estimate the cycle cost based on Graal’s approach to determining an approximate cycle cost for a compilation unit.

Graal estimates the cost for each IR node, considering branch probabilities and loop counts based on previously observed run-time behaviour. With the cycle estimate for the compilation unit, and its activation count, we can calculate an estimate for the overall number of cycles taken. However, the estimate does not model memory, so its estimates may be less reliable for memory-intensive code. Nevertheless, it is still in part based on previous profiling at the JVM level, and thus is not merely static, but is also based on the observed dynamic behaviour.

Since we estimate the cost of our normal instrumentation to be about 80 cycles, we use this as the threshold. When a compilation unit is estimated to require more than 80 cycles, we instrument it with our standard probe that measures the activation time. Otherwise, we only count activations and calculate the time based on the profile-based estimate. We evaluate the tradeoffs of this approach in [Section 4.5.2](#).

4.3.4 Attributing Cycles to Inlined Methods

As discussed in [Section 2.6](#), 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 inter-mixed with parts from other methods.

To attribute time to inlined methods after all optimizations, we estimate the cycle cost for the remaining elements of an inlined method. We collect inlining information and 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 originates from a specific method.

This allows us to identify which methods make up the hottest compilation units, while avoiding the overhead of instrumenting each remaining part of an inlined method separately.

4.4 Research Questions and Methodology

This section details our research questions for assessing late-compiler-stage instrumentation, followed by the experimental setup used to evaluate them.

4.4.1 Research Questions

RQ3: What is the run-time overhead of Late-compiler-stage Instrumentation? We will measure the overhead of BuboC and compare it to both instrumentation-based profilers as well as the overhead of sampling-based profilers. This will give us an initial idea of the impact of the late-compiler-stage instrumentation on the overall behaviour of a program.

RQ4: Does Late-compile-stage Instrumentation yield more precise results? Precision, i.e., how close measurements from run to run are to one another, is another important requirement for a profiler. Since sampling profilers can struggle with precision ([Chapter 3](#)), we will assess whether late-compile-stage instrumentation is able to produce results that are similar across multiple runs.

4.4.2 Experimental Setup

All experiments were executed using the Graal compiler on top of the HotSpot JVM in OpenJDK 21.0.2. We are using a version of the Graal compiler from April 2024.⁶ Graal is generally used in its *libgraal* version, which means that the compiler itself is ahead-of-time-compiled. This ensures that Graal gives the best possible compilation times from the start. However, some minor engineering issues prevented us from using *libgraal* for BuboC, and thus, all experiments with BuboC use *jargraal*, i.e., the Java version that itself is subject to JIT compilation. We believe this will not have any major impact on our results, and in the worst case might put BuboC at a disadvantage compared to other profilers.

All benchmarks (see [Section 2.9](#)) were run on a machine with an AMD Ryzen 5 3600 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.

⁶<https://github.com/oracle/graal/commit/249d3e4abd2f357461c5ceb682791e22b2c8a92f>

Profilers

For our evaluation, we use both instrumentation- and sampling-based profilers. We selected these profilers based on our previous work in [Chapter 3](#) and their compatibility with JDK 21. The selected sampling profilers are listed in [Section 2.10](#), and the selected instrumentation-based profilers are the same as those used in [Section 4.2](#). For each tool we used the following version for this chapter. Async-profiler (April 2024 build⁷), Honest Profiler (February 2022 build⁸), Java Flight Recorder as shipped with OpenJDK 21.0.2, perf (Linux 5.14.0), JProfiler 14.0.2, and YourKit 2024.3-b150.

4.5 Evaluation of Late-Compiler-Stage-based Instrumentation

In this section, we compare our approach of late-compiler-stage-based instrumentation, as implemented in our BuboC profiler, with sampling- and instrumentation-based profilers.

4.5.1 Run-time Overhead

We start by comparing the run-time overhead of late-compile-stage instrumentation to sampling- and instrumentation-based profilers to determine which type of profiler BuboC is most similar to in terms of overhead.

BuboC vs Classic Instrumentation

As previously shown in [Section 4.2](#), [Figure 4.1](#), the median overhead for the instrumentation-based profilers is in the range of $82\times$ to $374\times$ over all AWFY benchmarks. The lowest median overhead comes from JProfiler, causing programs to take $82\times$ more time compared to their uninstrumented version. For example, the run time of the DeltaBlue benchmark changed from 108ms per iteration to 36738ms when being instrumented by JProfiler.

⁷<https://github.com/async-profiler/async-profiler/commit/005f7a3dc38f3bb369d3a024dfbe5684551cfff2>

⁸<https://github.com/jvm-profiling-tools/honest-profiler/commit/5783ffab6690807783521a1b6dca6e74e77ba853>

In contrast, BuboC has a median overhead of 23.3% (min. 1.4%, max. 464%) for all 14 benchmarks, resulting in a much lower impact on the execution of a program.

BuboC vs Sampling

Sampling profilers are expected to have a lower overhead than instrumentation-based profilers, since any overhead would be proportional to the sampling frequency, instead of incurring an overhead, for instance for every method activation.

Given that BuboC's overhead is much lower than that of classic instrumentation-based profilers, we will also compare directly to sampling profilers. For each benchmark, we calculated the median iteration time with and without a profiler attached, based on which we determine the run time overhead induced by each profiler. The results are shown in [Figure 4.5](#).

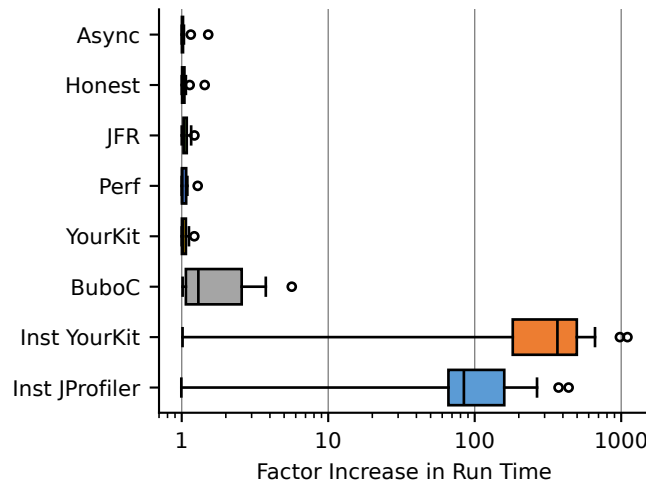


FIGURE 4.5: Percentage of run-time overhead that each profiler adds to the AWFY benchmarks. Async has the lowest median overhead over all benchmarks with 1% (min. 0.1%, max. 52%). BuboC has a median overhead of 23.3% (min. 1.4%, max. 464%).

BuboC has the highest median overhead among the sampling-based profilers, with 23.3% (min. 1.4%, max. 464%). This is expected, because the cost of instrumentation exceeds that of periodic sampling. BuboC's median overhead of 23.3% is also higher than the 75th percentile of the other profiles, as indicated by the right edge of the boxes. Nonetheless, BuboC's overhead is much closer to that of sampling-based profilers than that of instrumentation-based ones.

4.5.2 Tradeoffs of Estimating Small Compilation Units

As discussed in [Section 4.3.3](#), for compilation units that are below a certain estimated cycle cost, BuboC only counts the number of activations and estimates the compilation unit's cost, instead of actively measuring the number of cycles they take. To assess how this impacts profiling overhead and the ratio between measured and estimated cycles, we vary the cycle cost threshold in this experiment.

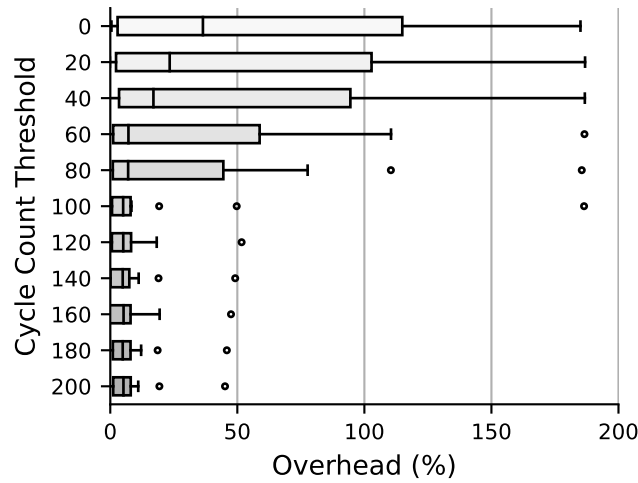


FIGURE 4.6: Profiling overhead depending on full instrumentation threshold shown as a box plot. Compilation units with an estimated cycle cost larger than the threshold are fully instrumented to measure the elapsed CPU cycles. The other compilation units only count their activations.

Overhead for different thresholds [Figure 4.6](#) shows the profiling overhead over all benchmarks for different cycle cost thresholds for full instrumentation. A threshold of 0 means that all compilation units are fully instrumented. For these results, BuboC only instruments compilation units and does not additionally instrument calls into non-inlined methods. We use this configuration to isolate the effect of instrumenting whole compilation units and to study the tradeoff between full instrumentation and estimation under different cycle thresholds. For other thresholds, compilation units less-than the threshold merely count their activations. As outlined in [Section 4.3.3](#), we estimate the overhead of full instrumentation to be about 80 cycles.

[Figure 4.6](#) shows thresholds up to 200 cycles and focuses on the first 200% of overhead. We omit the outlier for a threshold of 0 cycles, where BuboC fully instruments all compilation units and reaches an overhead of up to 435%, which is the worst case we observe.

For the threshold of 80 cycles, we have a median overhead of 9.6% (min. 0.1%, max. 187%), which is significantly lower than the median overhead of 36% for the threshold of 0 cycles.

For the Are We Fast Yet benchmarks, increasing the threshold further to 100 cycles reduces the median overhead to 5% (min. 0.1%, max. 186%). We do not observe any benefit for higher cycle thresholds.

Proportion of measured and estimated cycles. While our approach to cycle estimation is based on initial JVM-level profiling, which is part of the JIT compilation process, we expect that measurements are more accurate than estimates. Thus, for deciding on a threshold for instrumentation we want to know the proportion of measured and estimated cycles in the benchmark executions. Figure 4.7 shows the results for different thresholds.

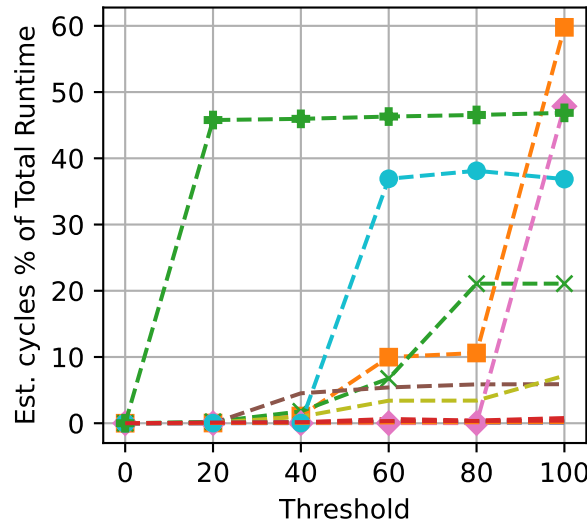


FIGURE 4.7: Per benchmark percentage of run time that is estimated based on the threshold for full instrumentation. The benchmarks of note are Permute (+), Queens (•), Richards (■), DeltaBlue (x), and List (♦), since they see an increasing number of cycles being estimated with the increasing threshold for full instrumentation.

Starting with a cycle threshold of 20, we see that the Permute benchmark (+) has 46% of its recorded cycles estimated. Though, across all benchmarks the median percentage of measured cycles is 99.96%.

With the minimum cycle threshold increased to 80, we see 21% of cycles being estimated for DeltaBlue (x) and 38% for Queens (•). Across all 14 benchmarks the

median timed percentage is 99.62% (min. 53.45%, max. 99.98%), which means 0.37% (min. 0.01%, max. 46.54%) are estimated. At a threshold of 80 cycles, for the majority of our benchmarks, our cycle information is measured rather than estimated.

In regard to RQ3, these results suggest that late-compiler-stage instrumentation can keep its impact on overall program overhead low. By trading direct measurement for estimation, we can reduce the run time overhead while still accounting for majority of execution cycles with measured data and only a small fraction via estimation. For the AWFY benchmarks, the 80-cycle threshold reduces the median overhead to 9.6% (min. 0.1%, max. 187%) (see [Figure 4.6](#)), while in the median only 0.37% of total cycles are estimated. Even in the benchmarks with higher estimated measurements, the estimates are still derived from run-time feedback for example, branch probabilities and loop frequencies. Finally, when measuring all calls, BuboC’s median overhead of 23.3% (min. 1.4%, max. 464%) placing it much closer to sampling profilers than to state-of-the-art instrumentation, and the mitigation strategies above further reduce the extent to which instrumentation effects total run time.

4.5.3 Precision of Late-Compile-Stage Instrumentation

To evaluate the precision of late-compiler-stage-based instrumentation and answer RQ4, we show how similar results from multiple runs are between BuboC and sampling profilers. For this, we ran all 14 benchmarks 30 times per profiler. From the collected profiles, we identified each method that contributed at least 5% of the run time. For each of these methods, we calculated the difference between the smallest and largest percentage of the overall run time as the maximum percentage minus the minimum percentage across the 30 runs. For each profiler, the median difference is the median of these differences across all methods and benchmarks. We chose the 5% cutoff to reduce noise from methods that do not dominate the overall execution.

[Figure 4.8](#) shows results for each profiler. The data points are the differences in run time of the methods for all benchmarks. Thus, a smaller median difference in run time corresponds to fewer differences between multiple profiler runs, which suggests that the profiler is more precise. The highest median difference from all our profilers is from YourKit with 2.6% which suggests relatively high precision for the median

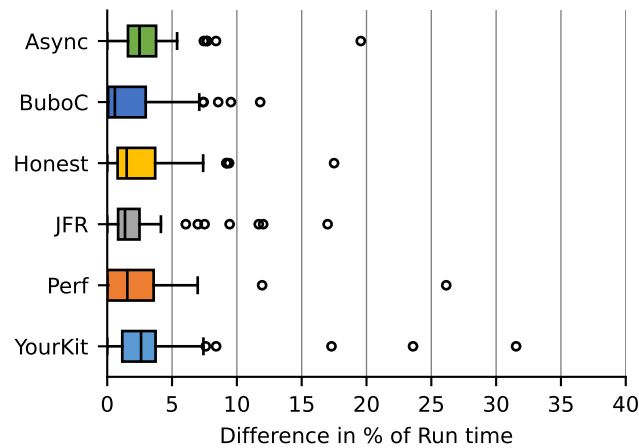


FIGURE 4.8: Difference in % of run time spent in every method, for each profiler. Each data point is the difference between the minimum and maximum percentage of run time for each method.

method. Though, for some methods, the profiler struggles to obtain precise results. The most likely cause is that the tool itself affects the compilation behaviour (see [Chapter 3](#)).

BuboC has a median difference in run time of 0.6% (0.02%, 11.8%), which suggests that our approach of late-compile-stage instrumentation can yield results that are at least as precise as traditional sampling profilers.

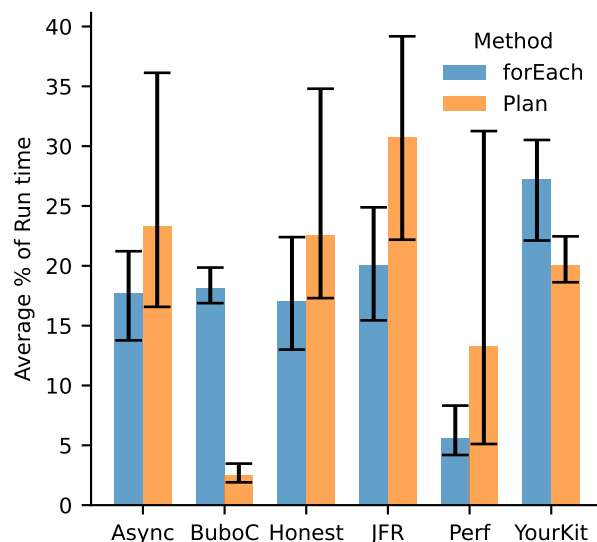


FIGURE 4.9: Bar chart with the ranges for the two methods most commonly identified as taking the most time in the DeltaBlue benchmark. The bar indicates the mean run time percentage attributed to the method. The error bars indicate the minimum and maximum percentage attributed to it.

Since the DeltaBlue benchmark is known to be a challenge for JVM sampling

profilers (as seen in [Chapter 3](#)), we will take a more detailed look at the precision for the two methods that most profilers identify as the methods in which the benchmarks spend most of their time. The results are visualized in [Figure 4.9](#). As expected, the five sampling profilers produce results that can differ significantly between the 30 runs, which is visible in the wide range of smallest and highest run-time percentage as indicated by the error bars. For all sampling profilers, the range is wide enough that the first and second methods can switch places in a profile between runs. For BuboC, the ranges of the `forEach` and `Plan` methods are much more narrow. This means, these methods do not change places in the profile between runs.

4.6 Practicality and Limitations

In this section, we will discuss how practical it is to adopt our late-compile-stage approach, its limitations, and the limitations of our evaluation.

Practicality of late-compile-stage instrumentation The phase-based architecture of the Graal compiler makes it possible to put our approach into practice without major engineering hurdles. Based on our git statistics, BuboC adds 1,023 lines of code to Graal and we changed 168 lines in already existing classes. Furthermore, our profiler is only active if an optional flag is set. This ensures that normal executions incur no overhead when the profiler is not enabled.

Current implementation limitations of BuboC In our prototype implementation, one of the major limitations is that we assume an upper limit on the number of compilation units that can be profiled. The buffer for cycle information is a single array of longs, with a size of 200,000 elements. We believe that in a production implementation, this can be avoided by reserving space, for instance in the metadata describing the compilation unit, and thus, there would be a space to record the cycles that is directly connected with each compilation.

Another limitation is that we currently do not use a thread-safe design to update the cycle count. We did not need it, because the *Are We Fast Yet* benchmarks are single threaded. Making the profiler thread-safe without performance penalty would

require further work, but overhead could be minimized for instance by using thread-local buffers, which only need to be written to global storage periodically.

Instrumenting non-inlined calls As discussed in the implementation, BuboC can also instrument non-inlined calls from within compilation units, but we only enabled this for the overhead results in [Section 4.5.1](#). The remaining experiments in this chapter were collected with a version of BuboC that does not instrument non-inlined calls, because this capability was not yet implemented at the time they were produced.

Enabling non-inlined call instrumentation increases the number of RDTSC-based probes in the compiled program, and therefore increases overhead. However, we expect the overall trends in [Section 4.5.2](#) to remain the same. In particular, selective instrumentation still avoids fully instrumenting small compilation units that contribute few cycles, and many such units contain few or no non-inlined calls. We therefore believe that the findings in [Section 4.5.2](#) remain valid and demonstrate the opportunity for instrumentation optimisations.

For the precision results in [Section 4.5.3](#). Without instrumenting non-inlined calls, BuboC attributes callee time to the caller compilation unit, even though that time is spent executing outside the unit. Because callee behaviour can vary across runs, this adds run-to-run variability to the compilation unit’s measured cycles; instrumenting calls and subtracting their time would give a more accurate measure of the unit’s self time. Therefore if we had instrumented non-inlined calls BuboC precision should only improve.

Finally, for [Figure 4.9](#), both `forEach` and `Plan` do not contain non-inlined calls, and therefore those results would remain unchanged even if non-inlined call instrumentation were enabled.

Measuring time Our current implementation measures cycles⁹ with the Read Time-Stamp Counter (RDTSC) x86 instruction.¹⁰ To our knowledge, it is one of the fastest available options. On the Zen 2 architecture we use, it takes 34-40 cycles (Fog [2022](#)). This instruction is documented to not be serialized, i.e., it can overlap with other

⁹RDTSC cycle values can be converted to time

¹⁰The x86 specification for RDTSC is given in the AMD64 Architecture Programmer’s Manual [p. 443] (AMD Inc. [2024](#)).

instructions in the out-of-order pipeline. The Read Time-Stamp Counter and Processor ID (RDTSCP) instruction would give slightly stronger properties, but comes with a higher overhead. The overhead for Java's `System.currentTimeMillis()` and `nanoTime()` methods is even higher, though, they would also guarantee stronger semantics. In an early experiment, we saw 210% overhead on DeltaBlue compared to the uninstrumented version when using `currentTimeMillis()`. At that point, switching to RDTSC reduced the overhead to only 80%.

Late versus final stage instrumentation Our approach to late-compiler-stage instrumentation only introduces nodes into the latest practical compilation phase on the graph-based GraalIR intermediate representation. This means our instrumentation is not yet present when inlining, loop transformations, and dead code elimination occur. However, our instrumentation is added before the graph is turned into the lower intermediate representation, and the compiler will still perform address translation, register allocation, and instruction selection at this level. Thus, our instrumentation has an influence on these optimizations. However, we believe that the impact here is in the general case minimal and moving the instrumentation to a point after these optimizations would only be of limited benefit while requiring much higher engineering effort for this use case. Nevertheless, we explore later-stage instrumentation in [Chapter 6](#), where BuboL places instrumentation even closer to final machine code to reduce its influence on compiler optimisations.

Verifying accuracy of profiling results An ideal profiler would yield profiles, which with 100% accuracy and precision reflect the behaviour of the uninstrumented program. Unfortunately, to obtain a perfectly accurate profile, one would need to run the program in question either on a simulator or on hardware that can perform the profiling without interfering with program execution (Gottschall et al. 2021). To the best of our knowledge, practical approaches rely on approximating correct results, which is a challenge with the complexity of today's systems, JIT compilation, out-of-order instruction execution, and complex cache hierarchies. This very issue is something we seek to solve in a later chapter. In [Chapter 5](#), we implement and demonstrate

a practical approach to approximate the ground truth, and use it to evaluate the accuracy of profilers.

4.7 Related Work

For related work, we will focus on instrumentation at the compiler level, binary rewriting, and work trying to assess the accuracy of profilers.

Instrumentation in compilers The work of Zheng et al. (2015) and Basso et al. (2023) inspired our approach to perform late-compile-stage instrumentation. Zheng et al. (2015) 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 Basso et al. (2023) and Zheng et al. (2015) focused on specific compiler optimizations and general events around them, as well as better understanding the compiler and possible performance issues with it. We on the other hand focus on more classic application-level profiling to achieve precise profiling results on the method level.

Frenot et al. (2024) reduce the overhead of exact control-flow profiling by using existing program variables, such as loop counters, to avoid inserting additional profiling counters. Similar to BuboC, their approach uses compiler information to reduce instrumentation overhead. The two works do differ, as their work reconstructs exact edge frequencies, whereas BuboC measures cycles of compilation units. A future tool could combine these approaches, using their technique to reduce the number of edges that require instrumentation and BuboC's late compiler-stage placement to reduce interference with compiler optimisations, thereby reducing both profiling overhead and the observer effect.

Adding instrumentation at the compiler level is not new. One early example is gprof (Graham et al. 1982), which added instrumentation as part of C, Fortran, and Pascal compilers into the relevant methods. Compilers such as GCC and LLVM also use profiling to enable profile-guided optimizations, and thus, support the placing of instrumentation into the compiled code (Preuss 2010).

Binary rewriting In some cases, modifying the compiler itself is not the best option and instrumenting binaries instead can be more beneficial.

Dynamic binary instrumentation can for instance be used to make instrumentation very targeted for use in production. For instance, Mahlke et al. (2013) samples the program using instrumentation at configurable frequencies set by the user. Other work on binary instrumentation focused on reducing the cost of instrumentation. Kumar et al. (2005) for instance combine multiple probes into one to reduce the overhead without losing information. To further optimize the instrumentation, some approaches use self-modifying instrumentation (Roy et al. 2011).

Exact Profiling Frenot et al. (2024) reduce the overhead of exact control-flow profiling by using existing program variables, such as loop counters, to determine how often code executes, avoiding the need to insert some additional profiling counters. Similar to BuboC, their approach uses information available within the compiler to reduce instrumentation overhead. However, their work reconstructs exact control-flow edge frequencies in statically compiled programs, whereas BuboC measures the cycles spent in dynamically compiled Java compilation units and places instrumentation late in compilation to reduce its interference with JIT compiler optimisations. Furthermore, exactness in their work refers to the completeness of edge-frequency counts during the instrumented execution, rather than whether that execution remains representative of an uninstrumented execution.

4.8 Conclusion

In this chapter, we showed that state-of-the-art instrumentation-based profilers for Java have high overhead and report unrealistic results.

To overcome these issues with instrumentation-based profiling, we proposed late-compiler-stage-based instrumentation. Our approach minimizes interference with compiler optimizations and as a result delivers more realistic profiles than other instrumentation-based Java profilers. At the same time, we achieve better precision than sampling profilers and reduce the median profiler overhead on the

Are We Fast Yet benchmarks to 23.3% (min. 1.4%, max. 464%), which is more similar to that of sampling-based profilers.

To achieve this low overhead, we instrument small compilation units with probes that only count the number of activations. For these compilation units, we then estimate the CPU cycles based on run-time compiler profiling information about their cycle costs. Thus, these estimates are not purely static, but more accurately reflect the program behaviour observed before compilation. Furthermore, we use the compiler's understanding of inlining to attribute the cycles recorded for a compilation unit to the method fragments that remain in the compilation unit after optimization.

With the work demonstrated in this chapter, we have highlighted issues with instrumentation-based profilers and outlined an approach to overcome these issues. However, the broader challenge of determining, with a high degree of confidence, whether a profiler is truly accurate still remains. We address this in the next chapter by using deliberate machine code slowdown, which allows us to approximate the ground truth of performance behaviour.

Chapter 5

Approximating the Ground Truth with Machine Code Slowdown

5.1 Introduction

Sampling profilers for systems with JIT compilation can suffer from safepoint bias, report wildly different results across multiple runs, and different profilers may disagree on results for the same program, as seen in [Chapter 3](#).

To improve these profilers, we need to be able to determine whether they are accurate. This, in turn, means we need to be able to compare the profiles they record with a ground truth, i.e., a perfect representation of where a program spends its time.

Unfortunately, the complexity of today's hardware and software systems makes it virtually impossible to derive a correct ground truth from a theoretical performance model, and using hardware simulation is generally impractical (Gottschall et al. [2021](#)). This, however, means the only practical approach is to approximate a ground truth based on measurements of a concrete system. This is problematic because measuring the system changes the system, (see [Chapter 4](#)) i.e., it induces an observer effect (Machkasova et al. [2009](#); Zheng et al. [2015](#)). Measuring at the CPU hardware level comes with overheads, profiling JIT compiling VMs such as the JVM can interfere with the JIT compilation, possibly impact optimization decisions, and suffers from the aforementioned safepoint bias (Mytkowicz et al. [2010](#)).

This chapter proposes a methodology to assess profiler accuracy in systems with JIT compilation. Although the methodology is applicable to both sampling-based and instrumentation-based profilers, we restrict our evaluation here to sampling-based

profilers. The main technique is to slow down a program accurately at the level of native code basic blocks using feedback from CPU instruction profiling support. By comparing profiles across different levels of slowdown, we approximate the ground truth, which allows us to assess the accuracy of sampling profilers. We implement our approach in the Graal compiler, ensuring that compilation decisions are not affected and that the program's performance behaviour remains unchanged.

To enable this methodology, we first need a way to slow down programs in a controlled and predictable way. We therefore investigate which instructions produce the most accurate and consistent slowdown. Across six candidate instruction sequences, we find that the instructions NOP and MOV provide the most predictable behaviour and the finest slowdown granularity. When targeting a 100% slowdown for the Towers of Hanoi benchmark, they achieve program slowdowns of 109% and 106% respectively. Other sequences proved far less suitable, including PUSH+POP (202% overhead), SFENCE (405%), a mixed instruction sequence (Long PUSH+POP, 410%), and PAUSE (11,505%). Based on these results, we use MOV for slowdown in the remainder of this work.

Given the complexity of modern CPUs with cache hierarchies, out-of-order execution, pipelining, and microcode (Colwell 2021), we demonstrate that we can slow down the Are We Fast Yet benchmarks (Marr et al. 2016) by adding an overhead of 50%, 100%, or 150%. We find that the median run-time overhead across all tested benchmarks is 50.0%, 100.0%, and 146.8%. The overhead of individual basic blocks is accurate for the majority of blocks that dominate the run time. This demonstrates the ability to accurately change the run time of these benchmarks with respect to total run time measured with wall-clock time as well as at the basic-block level, as observed by CPU instruction profilers.

As the next step, we confirm that the benchmarks maintain original performance behaviour, in the form of the percentage of time per method remaining the same with the different slowdowns, as one would expect. We investigate this using four sampling profilers for JVMs: Async-profiler, JFR, JProfiler, and YourKit.

We found that in 13 out of 14 benchmarks, the run time percentage for methods reported by each profiler was within 5%pt of the expected percentage. Only for one benchmark did they misattribute the run time. This happened because the debugging

information provided by the JIT compiler does not allow for a correct machine-code to source mapping.

Finally, we tested the accuracy of the profilers by approximating the ground truth by slowing down individual basic blocks. By adding a specific slowdown, we can estimate the change in run time an accurate profiler needs to report. This way, we can assess whether profilers are sensitive enough to detect and attribute the change accurately. We found three of the four profilers detected the changes correctly and accurately most of the time. However, JProfiler misattributed most of the changes and was significantly more inaccurate than the other profilers.

This chapter presents a methodology for assessing profiler accuracy in JIT-compiled systems and shows that, while sampling-based profilers can be accurate, their results remain limited by attribution challenges and sampling bias.

5.2 Existing Approaches to Assessing the Accuracy of Java Profilers

Since the accuracy of a profiler corresponds to how reliably developers can identify optimization targets, evaluating their accuracy has been explored previously.

Approaches and challenges to evaluating Java profiler accuracy Mytkowicz et al. (2010) proposed to assess whether profilers give *actionable* results by injecting a Fibonacci sequence computation into methods at bytecode loading time. With this approach, they tested whether a profiler detected the run time caused by this injected computation, and whether a profiler correctly attributed the additional run time to the modified method, and if not, how far they are off as a ratio between expected and actual run time.

With compiler optimizations such as inlining and dead code elimination, their approach becomes unfortunately unreliable. On JVMs, compiler optimizations such as inlining are applied on the modified bytecode, which means the inserted computation can change optimization decisions. Additionally, modern CPU design features, such as out-of-order execution and pipelining, introduce further complexities. Thus, any slowdown injected into the source or bytecode may have unintended side effects.

Inlining and code motion may also impact where the slowdown is applied, which means that there is no guarantee of a consistent slowdown effects for a single target method to approximate accuracy more precisely. We discuss this further in the context of Graal in [Section 5.7.3](#).

Challenges in evaluating profiler accuracy: the need for a ground truth Evaluating profiler accuracy requires a ground truth: a known reference of program performance behaviour against which a profiler can be compared. However, such a reference is difficult to obtain in JIT-compiled systems. Using source code to determine execution behaviour is unrealistic, because compilation, optimisation, and hardware can substantially change the machine code that is ultimately executed. Even sampling profilers, which do not directly modify the program, can still perturb execution and compilation enough to introduce an observer effect.

Ground truth in non-deterministic systems In addition to the above issues, in most modern systems, each program run establishes essentially its own ground truth. Because of the nondeterminism, caused by for instance caching, garbage collection, and concurrent execution, every run can execute slightly differently. This means, theoretically, there are multiple ground truths for a program. To make the problem tractable, one needs to reduce non-determinism, to achieve more stable program behaviours that can be compared.

To approximate the ground truth, we implemented a dynamic system that introduces accurate and targeted slowdowns throughout a program. Furthermore, we reduce the non-determinism in the system by using compilation replays, and more deterministic compiler settings, to achieve comparable program runs.

5.3 Divining Profiles: Approximating a Ground Truth with Basic-Block-level Slowdowns

In the following section, we outline our approach to slowing down a program, without interfering with compiler optimizations and minimizing impact on microarchitectural performance behaviour. We first describe the idea behind our approach,

give an overview of it, and then discuss its implementation in Graal.

5.3.1 Approximating the Ground Truth with Proportional Slowdowns

Our approach relies on approximating the ground truth by proportionally slowing down a program, and identifying deviations from the expected performance. We start by defining the key terms.

Ground truth and approximated ground truth The **Ground Truth** is the true run time behaviour a program exhibits, for a specific execution. In our experience, machine-code basic blocks are a suitable abstraction in this context. We thus abstract program behaviour by time t_i for each basic block bb_i . Unfortunately, there exists no pure function $f(bb_i) = t_i$ to measure the time without interference. The complexity of modern systems means that executions may differ even under stable conditions. Measuring an execution will introduce further differences by affecting compiler decisions, memory access behaviour, and the timing of pipelined and out-of-order execution. Therefore, attempting to measure t_i will itself change t_i , which is the observer effect.

Instead, we approximate f with a function $\hat{g}$ such that the difference between actual and approximate time is negligible,¹ i.e.:

$$|\hat{g}(bb_i) - f(bb_i)| \leq \varepsilon.$$

In our case, $\hat{g}$ is the time obtained from hardware counters, i.e., the **Approximated Ground Truth**, and we check for deviations ε by slowing down every block by a constant factor. We assume that there is a linear relation between a basic block's execution time and the time its slowed-down version takes:

$$\hat{g}(bb_i \times 1.5) = 1.5 \hat{g}(bb_i), \quad \hat{g}(bb_i \times 2) = 2 \hat{g}(bb_i), \quad \hat{g}(bb_i \times 2.5) = 2.5 \hat{g}(bb_i).$$

These constant factors 1.5, 2, and 2.5 are our target slowdowns of 50%, 100%, and 150% that we will use in our experiments. Any departure from this linear relation

¹Here, negligible means that the difference is too small to materially affect our evaluation of whether a profiler correctly reports the change in the proportion of time spent in the slowed-down method.

will show up in experiments as a deviation in the time spent inside the basic block and consequently the corresponding method. If our Approximated Ground Truth is inaccurate, the slowdown will exaggerate a change in program behaviour. We will test our assumption of a linear relation by looking for deviations and the effect slowdown has on program behaviour in [Section 5.6.2](#).

Using the approximated ground truth to evaluate profilers To evaluate profilers, we use a known approximate ground truth and apply the slowdown to a single code region. An accurate profiler should report an increase in run time for the slowed-down region only, leaving all other regions unchanged. Because we precisely control how much slowdown is added, we can measure the extent to which a profiler reflects actual execution behaviour, e.g. the exact amount of additional run time in the target code region should be detectable. This allows us to detect profiler inaccuracies at the method level. Therefore, when evaluating a profiler, our approximated ground truth is the expected difference between the original and slowed-down profiles, which we use to determine whether the profiler correctly identifies the program location and magnitude of the change.

5.3.2 Slowing Down a Program Proportionally and Accurately

To slow down a given program, we need to understand how it compiles to native code and how much time it spends in each basic block of the native code. Then we can slow down each basic block by the desired factor and consequently increase the run time proportionally and accurately. However, because of the complexity of today's systems, we cannot predict the needed slowdown (see [Section 5.7.2](#) for a discussion). Instead, we need to measure how much time each basic block takes and then insert a suitable number of instructions into the basic block, so that its run time increases appropriately. To measure the time a basic block takes, we use VTune's hardware-counter-derived timing information (see [Section 2.8](#)). Because of the complexity of today's systems, this is a search process for the appropriate slowdown. We refer to this search as *divining*. The end result is that we know for each basic block how many instructions we have to insert to achieve the desired run-time increase. [Figure 5.1](#) illustrates the steps of this process.

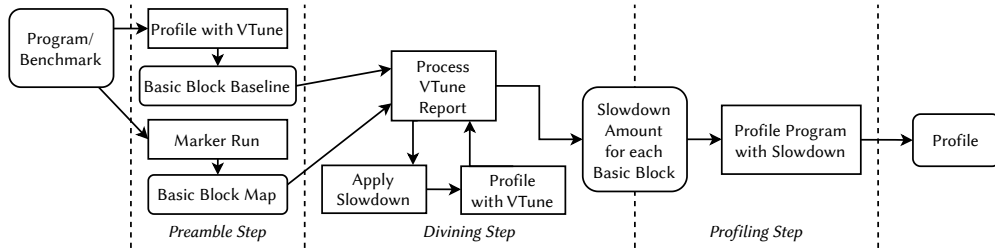


FIGURE 5.1: Illustration of the steps to slow down a program. First, in the preamble step, we identify how the program compiles to native code and how much time basic blocks take without slowdown. In the divining step, we find the right amount of slowdown for each basic block. This slowdown information can then be used to profile the program with a Java profiler, for instance, to assess the profiler’s accuracy.

In the remainder of this section, we detail how we implemented this process in the Graal compiler. First we discuss the GroundTruth Scheduler, which controls the overall process, and then detail the implementation of our custom compiler phase in Graal, that inserts instructions at the LIR (Lower Intermediate Representation) level.

5.3.3 Implementation of the Divining Process with our GroundTruth Scheduler

The overall process is automated with our *GroundTruth (GT) Scheduler*. It runs a target program with VTune and our modified version of the Graal JIT compiler to identify the appropriate amount of slowdown for each relevant basic block following the process from [Figure 5.1](#). The GT Scheduler implements the two main phases, the *Preamble Phase* and the *Divining Phase*.

Preamble phase The preamble phase performs two runs using the VTune profiler. It first runs the target program without any modifications, but with VTune attached to determine the baseline run time for all relevant basic blocks. Afterwards, it does a *Marker Run*, which is used to establish a reliable mapping between basic blocks at the Graal LIR level and basic blocks in the final native code. For this, we use a Graal LIR phase that introduces marker instructions that encode Graal LIR block IDs into the native code, as detailed in [Section 5.3.4](#). The marker run is also profiled with VTune so that we can process the hardware instruction profiles to establish the mapping.

The GT scheduler processes the VTune output of these two runs, comparing the generated assembly for each method with significant run time.² We parse the assembly text output VTune produces to detect all basic block markers, which provides us with a reliable mapping between Graal LIR and native basic blocks. Additionally, we collect the time each native basic block took in the normal run as baseline, and to know how much time blocks contribute to the total run time. Blocks that contribute fewer than 0.001 seconds are excluded from further processing.³

Divining phase The divining phase determines the amount of slowdown needed for each basic block to achieve the target slowdown. The process begins with initial slowdown of 1 for every block, i.e., the smallest amount of slowdown we can add to a block, which corresponds to adding one extra instruction. The program is then executed with these slowdowns and profiled with VTune. The generated hardware instruction profile is then analyzed to determine whether more slowdown is needed based on the time VTune reports for all instructions in a basic block.⁴

For blocks that do not exceed the target slowdown, an additional slowdown is added. If a block exceeds the target slowdown, we compare the achieved slowdown from the last iteration where the block was below the target and the one where it was over and pick the slowdown that comes closest to the target. The process terminates once we approximated the target slowdown for each basic block. The final slowdown value for each block is thus the one closest to the target slowdown, usually slightly under or over.

When blocks require significantly larger amounts of slowdown, ranging from hundreds to thousands of slowdown instructions we adjust the amount of slowdown applied per iteration. Instead of incrementally adding slowdown by 1 at a time, we start by increasing it in steps of 10. Once we surpass the target slowdown, we perform

²We process every method that contributes at least 2% of to the total run time, which is a good tradeoff between precision and the overall time divining takes.

³Similar to methods, this is a good tradeoff between precision and divining time. However, it also avoids including basic blocks that VTune sometimes omits because of the sampling of hardware counters, which would lead to instability in the divining process caused by such insignificant blocks appearing and disappearing non-deterministically.

⁴The underlying hardware counters and sampling may misattribute execution time to neighboring instructions. We assume that the accuracy on the level of basic blocks is sufficient for our purposes, and discuss the issues and implications more fully in [Section 5.7.2](#).

a binary search, using the last known slowdown amount that was below the target speed and the first amount that exceeded it, to refine the slowdown more accurately.

Note that we handle each compilation unit separately to optimize the overall time the divining process takes. This avoids slowing down the whole program and handling each compilation unit independently takes less time overall. Though, all blocks within a compilation unit are processed simultaneously during each iteration, to ensure that performance behaviours between basic blocks of the same compilation unit are not changed.⁵

The end result of the divining process is a JSON file that encodes the needed slowdown for all relevant basic blocks in all relevant methods (compilation units), which can then be used to assess the accuracy of profilers.

5.3.4 Implementation of Compiler Phases in Graal

An implementation of our approach needs to overcome two challenges. It needs to minimize its impact on compiler optimizations to ensure the program's original performance behaviour remains unchanged, and it needs to reliably map basic blocks of the compiler's lower intermediate representation (LIR) to the basic blocks in the native code it generates.

We minimize any impact on compiler optimizations by implementing our approach as LIR phases. We place these phases as late as possible, right before the final compiler phase, which decides the order in which machine code is emitted.

The second challenge, the reliable mapping from LIR basic blocks to native code basic blocks is solved by explicitly marking native blocks with their corresponding LIR block IDs in the previously sketched preamble phase (see [Section 5.3.3](#)). By marking blocks, we overcome several issues: Graal LIR instructions can be complex and a single LIR instruction may result in multiple as well as different machine instructions based on context. This is because LIR instructions can have complex mappings to native code, and Graal performs instruction selection and peephole optimizations when it emits the native code. Thus, mapping back from the native code to the original LIR instruction is nontrivial. For instance, Graal has a LIR instruction

⁵While there can be performance effects between compilation units, this is a good tradeoff between precision and overall divining time.

that is compiled to a dispatch table, and the number of emitted native instructions depends on the number of entries in the dispatch table. This LIR instruction can also emit two branches, which means a single LIR instruction corresponds to multiple basic blocks in the native code.

To establish the mapping, we add a *Marker Phase* to Graal's LIR. The results of this phase are then utilized by our *Slowdown Phase*, also added at the LIR level, to insert slowdown instructions to the right basic blocks in separate runs.

```

1 Block 34:
2 mov r11, 0x1914
3 mov r10d, 0xffffffffcd
4 jmp <Block 36>

```

(A) Original Machine Code Block

```

1 Block 34:
2 vpbldd xmm0, xmm0, xmm0, 0x2e
3 vpbldd xmm0, xmm0, xmm0, 0x0
4 mov r11, 0x1914
5 mov r10d, 0xffffffffcd
6 jmp <Block 36>

```

(B) Marked Machine Code Block, the `vpblendd` immediate value here of `0x2e` signifies that it corresponds to the Graal LIR block 46

```

1 Block 34:
2 mov r11, 0x1914
3 mov rax, rax
4 mov r10d, 0xffffffffcd
5 mov rdx, rdx
6 jmp <Block 36>

```

(C) Slowdown Machine Code Block, the additional `mov` instructions at lines 3 and 5, were inserted to increase the CPU run time of the basic block

FIGURE 5.2: Three versions of a Machine Code Block: (a) the original unmodified block, (b) the version with marker instructions, (c) the slowed-down version.

Marker Phase

The marker phase is only used in the preamble step (see [Section 5.3.3](#)) to establish a reliable mapping between Graal LIR blocks and native basic blocks. It inserts side-effect-free operations into the generated machine code to encode IDs for the LIR-level block. These operations do not alter the program's computations, but are easily recognized as unique markers in the emitted machine code. Specifically, we use `vpblendd` instructions (Intel 2016 p. 5-321), which are part of the AVX2 extension. When used with a single register and an immediate value, e.g., as `vpblendd reg, reg, reg, imm8`, the register remains unchanged. By using two `vpblendd` instructions one after another we can use the immediate values to encode unique IDs as 16-bit numbers, which is sufficient to identify blocks within a single compilation unit. This

use of `vpblendd` with a single register is not generated by Graal otherwise, and thus, can safely be used for our purpose.

Figure 5.2b illustrates a basic block that was modified by a marking step. Compared to the original block in Figure 5.2a, Figure 5.2b has two additional `vpblendd` instructions. These `vpblendd` instructions contain the hexadecimal value `0x2e`, which is the decimal value 46. This allows us to determine that native basic block 34, as reported by VTune, corresponds to the Graal LIR block 46.

Slowdown Phase

The slowdown phase is used by the divining and profiling steps to insert a specified number of instructions that slow down a basic block, without otherwise changing the block's behaviour. This is implemented using a JSON file that includes for each compilation unit how many slowdown instructions need to be inserted for each specific basic block. The slowdown phase reads the JSON file named by a command-line option.

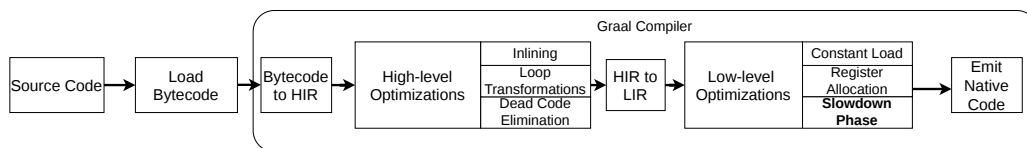


FIGURE 5.3: Sketch of the Graal compiler pipeline, going from source code to emitting native code. Our slowdown phase is one of the last low-level optimizations, after all high-level optimizations, such as inlining and loop transformations happened, and thus, does not impact them.

This phase is inserted late in the lower-level optimizations; after register allocations and constants loads, as illustrated in Figure 5.3. Therefore, slowdown instructions are added only after all major code transformations, such as inlining, loop transformations, and dead code elimination, have taken place. This avoids changing any high-level optimization decisions avoiding unintended behavioural changes. The slowdown phase loads the JSON file and reads the per-block information for the current compilation unit. For each basic block for which it has information, it inserts the specified number of `mov reg, reg` instructions. A `mov` to the same register consumes CPU cycles on the Intel Core i5-10600 we use, but has no other effect. We

investigated other instructions, and found 2-byte NOPs to be the only other viable option. The full investigation will be shown in [Section 5.4](#).

[Figure 5.2c](#) shows a slowed-down version of a machine code basic block. The slowdown phase inserted `mov` instructions on lines 3 and 5. Each `mov` operates on a different register and they are evenly distributed throughout the basic block as much as possible.

Register rotation to prevent pipeline stalling When multiple instructions use the same registers, they have to be carefully serialized by the CPU, which can cause *pipeline stalling*. This can induce much higher overhead than intended, since it limits for instance the amount of out-of-order execution that can happen. To prevent such pipeline stalling, we rotate the registers used in the `mov` instructions. We tested this approach on a Core i5-10600 with Skylake architecture. However, microarchitectures vary sufficiently so that the exact register selection strategy may need to be adjusted for specific pipelining, register renaming, and other microarchitecture optimizations based on the used CPU.

Distributing slowdown instructions within blocks In addition to using multiple registers, we also aim to distribute slowdown instructions evenly throughout a block whenever possible. For example, if a block contains ten instructions and requires ten slowdown instructions, we intersperse the slowdown instructions with the existing ones rather than clustering them at the start or end.

This is important for blocks containing code from different methods, or where instructions have no debugging information. As alluded to in [Section 2.2.2](#), profilers use debugging information to determine at an instruction level, which method an instruction belongs to. Thus, our slowdown instructions must not alter existing debugging information, and make sure that slowdown instructions are correctly attributed. If our slowdown altered existing debugging information, it could change the profiler's results e.g., causing run time to be attributed to wrong methods. To minimize this risk, we do not set any debugging information for our slowdown instructions and intersperse them as evenly as possible with existing instructions.

The effect of not doing this distribution will be evaluated in [Section 5.4.2](#). However, this approach is not without its limitations, as discussed in [Section 5.7.1](#).

5.4 Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level

In this section, we discuss the requirements for a machine-code-level slowdown instruction, as well as the six machine-code candidates we evaluate. We assess these instructions to determine which is most suitable for use in our methodology to approximate the ground truth.

5.4.1 Requirements for Machine-code-level Slowdown

Accurate amount of slowdown To accurately predict speedups and assess profiler measurements, a specific basic block needs to be slowed down proportionally to its original run time. For instance, the block's run time may need to be doubled.

This means, we need to be able to insert precise amounts of slowdown into a basic block. It also means the slowdown must apply to only the target basic block, without affecting other blocks. If a slowdown intended for one block, e.g. in `methodA`, inadvertently affects another part of the program, e.g., a block belonging to `methodB`, the performance behaviour becomes unreliable. Thus, if for any reason the slowdown is only approximate, it would limit the precision available to our approach of approximating the ground truth.

Side-effect-free slowdown When introducing slowdown into a program, we must maintain its logical correctness. Therefore, the inserted slowdown must avoid affecting e.g., memory, registers, and CPU flags, which subsequently could alter the program's logical behaviour. A program's execution must remain exactly the same, regardless of the applied slowdown, to maintain the validity of the measurement and ensure that any observed changes are solely the result of the intended slowdown.

Distributed Slowdown

While not a requirement for reliable program slowdown, for determining profiler accuracy, we expect that slowdown needs to be evenly distributed across a basic block. If it is concentrated in only a few spots, we may reach the target slowdown accuracy, but it may still bias observable performance behaviour. With inlining, a basic block can have instructions from different source methods (see [Section 2.6](#)). Thus, for tools that work at source level, slowdown may need to be attributed to specific instructions within a basic block. However, this depends on whether concentrated slowdown measurably changes the observed performance behaviour, and therefore needs to be evaluated. With the limited precision of hardware counter sampling, we therefore investigate whether evenly distributing slowdown within a basic block to limit biasing the performance behaviour from the source-level perspective too much.

5.4.2 Evaluating Candidates

This section evaluates our six candidates by determining how many of them are needed, which overhead they achieve, how the overhead is distributed over basic blocks, and how this changes the performance behaviour visible to a Java profiler.

Methodology

For this evaluation, we use the Towers of Hanoi benchmark from the Are We Fast Yet suite (Marr et al. [2016](#)). The Java version JIT-compile to 53 basic blocks within a single compilation unit, and has a suitable mix of larger and smaller basic blocks. Since it is fully deterministic, it simplifies experiments. We run Tower for 500 iterations and it is fully JIT-compiled after only a few. We rerun the experiment 10 times to determine the achieved overhead.

The main purpose of this evaluation is to determine the most suitable candidate instruction for slowdown. Estimating slowdown across all 14 benchmarks is time consuming: for a single instruction, such as MOV, the process takes approximately 3.75 days. We therefore restrict this evaluation to one benchmark. Once the most suitable instruction has been identified, it is used in the full evaluation, where it is applied across all 14 benchmarks and at three different slowdown levels (see [Section 5.6.1](#)).

Candidate Instructions

As discussed in [Section 5.4.1](#), a slowdown instruction or instruction sequence cannot affect the program’s underlying logic, must allow for accurate and deterministic slowdown, and we must be able to interleave it with other instructions. Based on these requirements, we selected the six candidates shown in [Table 5.1](#). They consume CPU cycles without altering the surrounding program logic or control flow, and thus, satisfy our requirements.

TABLE 5.1: Candidate slowdown instructions used in our study

Name	Opcode Template	Definition / Purpose	Motivation for Inclusion
NOP (2-byte)	66 90	No operation; e.g., used for code alignment.	Has zero side effects and intended for this use.
MOV (reg→reg)	89 /r /r	Dummy move of a register to itself, without side effects.	A fast and highly optimized instruction.
PUSH+POP	50 /r 58 /r	Push a register onto the stack and then pop it back into the same register.	A small sequence that may consume more cycles.
SFENCE	0F AE	Store-fence; ensures all preceding stores are globally visible before later memory operations.	Serializing delay that leaves registers untouched.
Long PUSH+POP	SUB SP, IMM VMOVDQU [SP], VEC VMOVDQU VEC, [SP] ADD SP, IMM	Allocates temporary stack space, spills and refills a vector register, then releases the space—consumes cycles without affecting logic.	Uses the vector unit of the CPU, to compare the performance behaviour with a normal PUSH+POP.
PAUSE	F3 90	Used in spin-wait loops, as a hint for the CPU to improve their performance.	A purpose-built delay instruction.

We use a 2-byte instead of a 1-byte NOP because the latter seemed to be removed by Graal. Since some Intel CPUs optimize NOPs early in their pipelines (Intel [2024](#) Section 2.1.2.1), we also included the MOV, and the PUSH+POP instruction sequence for variety. SFENCE and PAUSE were included because they consume CPU cycles without modifying registers or control flow. Finally, we included the Long PUSH+POP sequence to see whether using a different functional unit, e.g., the vector unit, makes a difference.

TABLE 5.2: Total Number of instructions needed to at least double each basic block's run time in the *Towers* benchmark.

Candidate Sequence	Total Inserted Instruction
NOP	303
MOV	304
PUSH+POP	72
SFENCE	55
Long PUSH+POP	63
PAUSE	55

Required Number of Slowdown Instructions

For each candidate, we determine how often it needs to be inserted into a basic block to double the block's run time, i.e., reach the 100% target slowdown. Table 5.2 shows the results. In total, over all 53 basic blocks, we needed to insert 303 NOP or 304 MOV instructions to achieve the slowdown. In contrast, we only needed 55 PAUSE instructions. This already hints at NOP and MOV having the finest granularity, while the other candidates are much coarser, and thus, give less flexibility.

Run-time Overhead

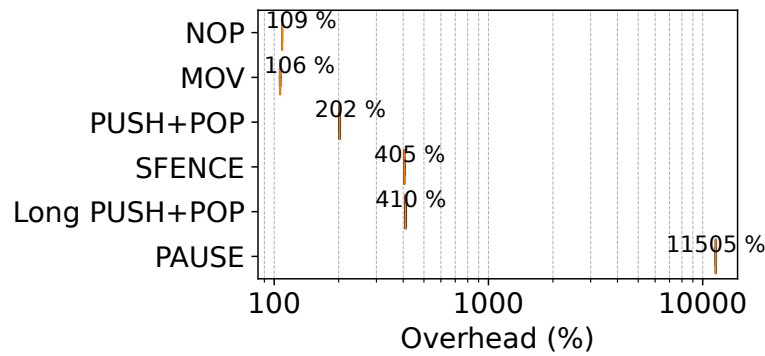


FIGURE 5.4: Boxplot with the overall program's run-time overhead achieved with each candidate for a target overhead of 100%. The x-axis uses log-scale. As baseline, we use a benchmark run without adding any slowdown.

Figure 5.4 shows a boxplot of the run-time overhead observed when running the *Towers* benchmark for each candidate. The boxes appear as lines because their interquartile ranges are small relative to the scale of the logarithmic x-axis. This variation indicates that all six candidates produce consistent run-time overhead across repeated executions. Only the NOP and MOV instructions achieve a run-time overhead close to the desired target of 100%. PUSH+POP causes an overhead of 202%.

SFENCE and long PUSH+POP overshoot the target even further with an overhead of about 400%. The highest overhead is caused by PAUSE with 11,504%.

These results show that inserting even a single instance of some instructions can slow down the program more than desired. In our experience, it is likely that a program has a few large basic blocks and many small ones. The candidates that cause high degree of slowdown may be suitable for slowing down particularly large basic blocks, though, it would concentrate the slowdown in one part of the block, instead of distributing it throughout.

Distribution of Slowdown Among Blocks

To determine whether the slowdown is accurately applied to each basic block, we plot their VTune-reported run time in [Figure 5.5](#) as a histogram. This shows how close each block's slowdown comes to the target of 100%. For each percentage-point difference, the plot shows how much percent of run time the blocks contribute to the overall run time. Ideally, all blocks would fall into the 0-bucket in the middle, which would mean 100% of the run time is covered by basic blocks that reached the target slowdown. Though, since a block may take virtually no time to execute, adding even a single instruction may slow it down by more than the 100% target.

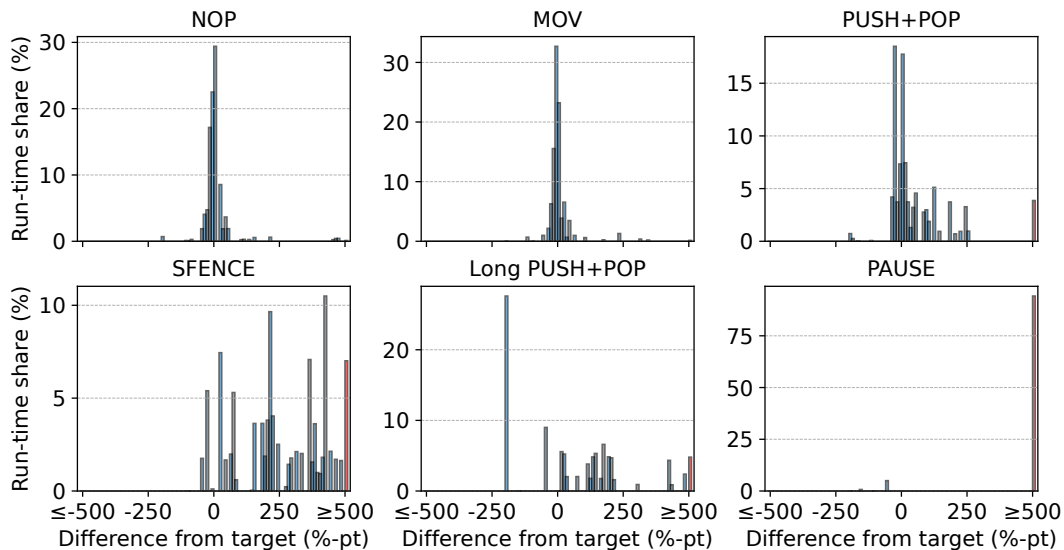


FIGURE 5.5: The histogram shows the difference in percentage points from the 100% target slowdown for all blocks on the x-axis. The % of benchmark run time is on the y-axis.

For the NOP and MOV instructions, the majority of the run time is clustered around the 0-bucket. The distribution is also close to symmetric, which means that while some blocks are below the target, others are above the target, which nearly balances out the difference. The PUSH+POP sequences see a larger run-time percentage being spent in blocks that take up to 250% points more time than the target.

For the SFENCE, the Long PUSH+POP, and the PAUSE instructions, this pattern is further exacerbated. For PAUSE, the blocks that cover most of Tower's run time are not even on the plot, and appear around the 5000%-point-difference mark. This means, that these candidates are not suitable to accurately slow down a majority of the basic blocks that one can expect in a program.

Performance behaviour with Slowdown

To evaluate the effect of slowdown on the performance behaviour of a program, i.e., how time is distributed across program parts, we compare the profile of normal with the slowed down runs, specifically, we use Async-profiler ([Section 2.10](#)).

If the relative percentage of total run time spent in each method remains the same, the slowdown is well distributed. Thus, basic blocks are slowed down accurately and when they have instructions from multiple methods, the run time is attributed in the same proportions as without slowdown. Thus, it indicates that the program retains a similar ratio of time spent in each program part, showing that our slowdown technique does not distort where time is spent by a program.

[Figure 5.6](#) shows normal and slowed down runs on a scatter plot where the top six methods in Towers are shown. The x-axis shows the median run-time percentage of 10 normal runs and the y-axis shows the median of 10 slowed down runs. Ideally, methods are on the $x=y$ line, which means their percentage of run time remains unchanged. Based on the plot, NOP and MOV give the best results, and the methods' run-time percentages are close to the $x=y$ line, staying roughly the same despite being slowed down. In contrast, Long PUSH+POP and the other candidates show major difference in the profile. For instance, for PUSH+POP, the `moveDisk` method goes from about 20% to around 40% of the program's run time.

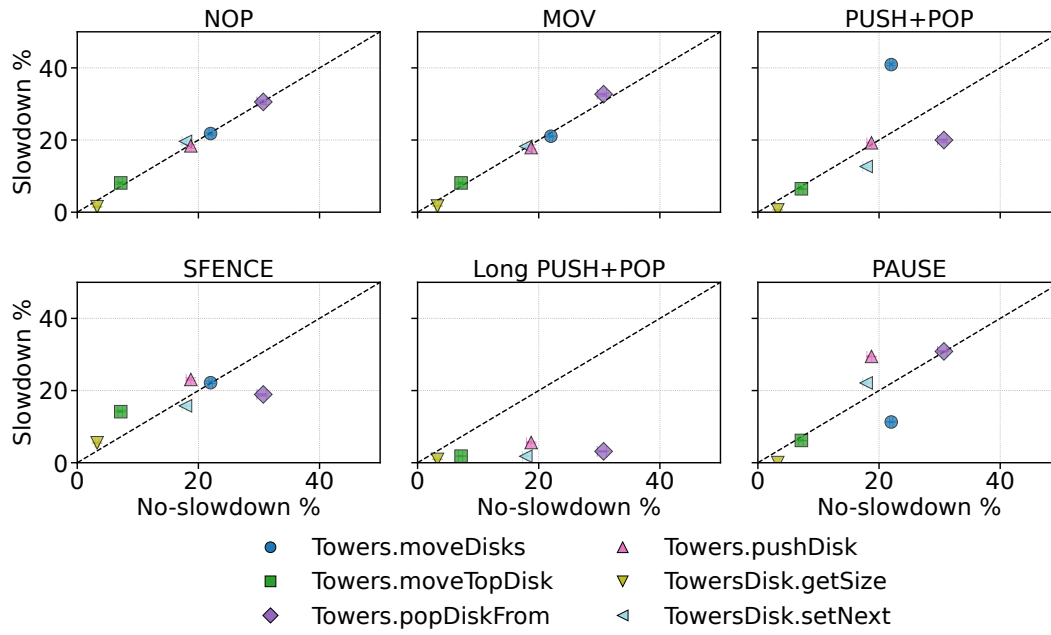


FIGURE 5.6: A scatter plot per slowdown instruction with the median run-time percentage for the top six Java methods. The $X = Y$ diagonal indicates that a method's run-time percentage remains the same with and without slowdown.

To quantify the difference more accurately, we calculated the mean squared error (MSE). NOP and MOV have an MSE of 1.2 and 1.5 respectively, which we consider negligible. The next best candidate is SFENCE with an MSE of 35.6. The long PUSH+POP sequence has the worst result with an MSE of 893.

Importance of Distributing Slowdown

As discussed in [Section 5.4.1](#), the instructions within a block can come from multiple source methods. Therefore, adding instructions only at the start or at the end of the block, we could bias the performance behaviour.

To test what effect the placement of slowdown has, we ran experiments that placed the slowdown at the start of the block, mixed throughout the block (our default approach), and with all the slowdown at the end. Similar to [Section 5.4.2](#), we then used async-profiler to record profiles.

[Figure 5.7](#) shows that different placements cause async-profiler to see noticeable differences in run time percentage for methods. Specifically, for NOP and MOV we see that placement affects how accurately the performance behaviour is maintained. For example, for NOP, placing the slowdown at the end of a block causes a larger change than when the slowdown is mixed in. To quantify the difference, we use the

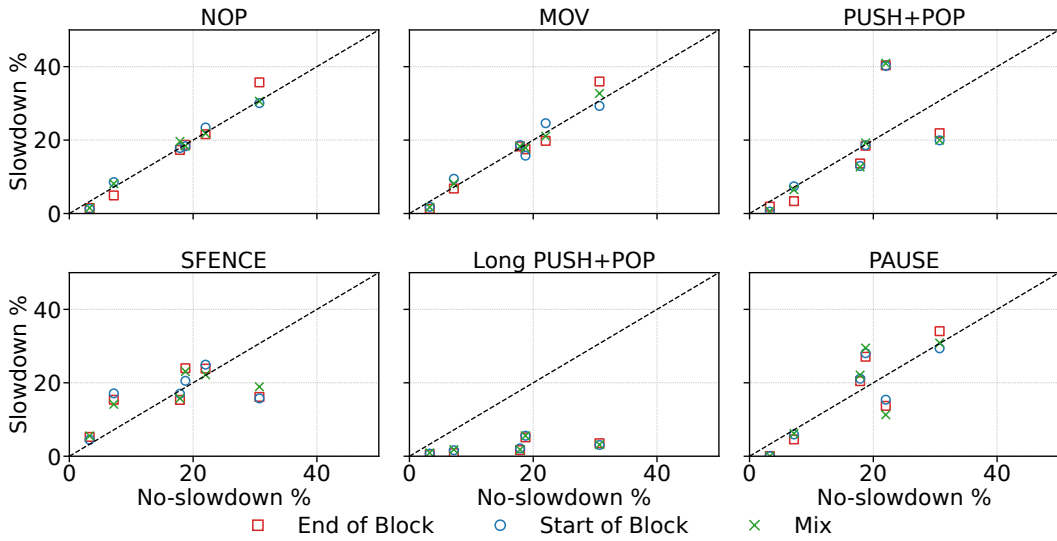


FIGURE 5.7: A scatter plot showing the impact of placing slowdown at the start, mixed throughout, or at the end of a basic block. We show the median percentage of run time for the top six Java methods of Towers. For a method on the $X = Y$ diagonal, the run-time percentage remained unchanged.

mean squared error (MSE). For NOP with the slowdown mixed throughout, the mean squared error remains 1.2, as in [Section 5.4.2](#). However, when the slowdown is at the block start, it goes up to 5.7, but when at the end, the MSE is only 1.4. For MOV, the MSE is 1.5 when mixing in the slowdown, and it goes to 4.3 for slowdown at the start and 6.4 for slowdown at the end. This not only confirms that placement can be important for specific use cases, but also suggests that indeed finer-grained instructions should be used that can be placed more freely and uniformly throughout basic blocks.

Best Candidate for Accurate, Side-Effect-Free, and Distributed Slowdown

We evaluated NOP, MOV, PUSH+POP, SFENCE, long PUSH+POP, and PAUSE as candidates for slowing down the Tower benchmark. NOP and MOV required the most instructions to be added to the Tower benchmark to double its run time, thus being the most fine-grained. For the overall as well as per-block run time, they were closest to the 100% target slowdown. Finally, with them the async-profiler saw the smallest divergence from the original performance behaviour. All other candidates are too coarse-grained to enable accurate slowdown of small basic blocks, and because of their high overhead, they can not be distributed as evenly throughout a basic block.

However, NOP and MOV may be optimized by CPUs (Intel 2024 Section 2.1.2.1), making them ineffective on other micro-architectures, requiring a re-evaluation of the candidates.

For MOV, we had to implement register rotation to prevent pipeline stalls. When many MOV instructions operate on the same register, the microarchitecture in use would at a certain number of MOVs cause much larger slowdowns than expected. We were able to verify these pipeline stalls using VTune in microarchitecture exploration analysis mode,⁶ and were able to compare the level of pipeline usage when using the same register. By rotating registers for the inserted MOVs, we avoided this effect and achieved a more uniform slowdown across all basic blocks.

We select MOV for the remainder of the evaluation because NOP and MOV showed almost identical slowdown behaviour, within the limits of measurement noise and run-to-run variation. If we evaluated both across the full benchmark suite, we would substantially increase the experimental cost without adding meaningful insight. We also found that MOV required slightly more inserted instructions than NOP on average, indicating finer slowdown granularity, and achieved run-time overhead closer to the target. Additionally, in our setup, NOP was more prone to being removed or treated specially, requiring a 2-byte encoding to avoid elimination, which made MOV the more stable practical choice.

In the next section, with MOV as the chosen slowdown instruction, we extend the evaluation beyond a single benchmark. We evaluate its effect on performance behaviour, its granularity in achieving three slowdown levels, and its distribution across all 14 Are We Fast Yet benchmarks.

5.5 Evaluating Machine-Code Slowdown for Approximating Ground Truth

We now evaluate our approach to approximating a ground truth profile with three research questions. The goal is to assess its suitability and to investigate the accuracy

⁶<https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/general-exploration-view.html>

of four JVM sampling profilers. First, we outline the research questions and then the experimental setup.

5.5.1 Research Questions

The first two questions assess whether our slowdown-based approach works as intended for all 14 of the Are We Fast Yet benchmarks. The third question aims to evaluate our approach to approximating the ground truth and assess accuracy.

RQ1: Does machine-level slowdown at the block level result in an accurately slowed-down program? With RQ1, we test whether slowing down programs using hardware-counter-based feedback leads to a proportional slowdown in all desired blocks. Additionally, we evaluate whether this approach is effective across all 14 of our benchmarks, ensuring a desired slowdown rate.

RQ2: Do sampling-based profilers report the same percentage of run time for identified methods regardless of program slowdown? Having confirmed a proportional slowdown, we test whether our approach maintains the original performance behaviour of the program. For this, we expect that our approach does not influence major compiler optimizations and minimizes impact on the performance behaviour at the microarchitecture level. Thus, sampling-based profilers should report the same percentage of time being spent in each method for a program with and without slowdown. We expect this, because the percentage of run time is relative to the total execution time, and all relevant basic blocks of the program are slowed down proportionally, if RQ1 holds.

RQ3: Do sampling-based profilers accurately detect changes to a slowed-down method? We test the ability of sampling-based profilers to detect substantial changes to the performance of a program part. When slowing down a specific block in a method, we expect profilers to detect the increased run time of that method and report an accurate time increase.

Slowdown granularity Our approach to slowing down basic blocks is limited by suitable machine instructions. Thus, while we would ideally want to slow down

basic blocks with arbitrarily small amounts to approximate the ground truth precisely, `mov` instructions were the most reliable smallest instruction on the microarchitecture in use.

This means, our approach is in practice not able to approach a close-to-zero overhead to achieve the greatest degree of precision. Furthermore, our experiments are limited by the hardware we have available and the time it takes to divine the needed slowdowns. As a practical tradeoff, we thus chose to use three slowdowns, 50%, 100%, and 150% based on our argument in [Section 5.3.1](#). When experimenting with slowdowns, these gave good results, avoiding too-small slowdowns that cannot be created reliably with a `mov` instruction. These choices are also spaced-out enough to make the impact clearly visible.

5.5.2 Experimental Setup

Given the complexity of modern systems, our experimental setup aims to minimize nondeterministic interference from the system, and the software components under our control. Since our implementation is relying on microarchitecture-specific behaviour, we run all experiments on a single machine to ensure consistent results. Furthermore, since we are using a system with JIT compilation, we run all benchmarks for 500 iterations to account for warmup ([Barrett et al. 2017](#)). We also verified the benchmarks are fully compiled very early, after one or two iterations. Furthermore, each experiment is run 10 times. The benchmarks were executed using ReBench ([Marr 2023](#)), which adapts the system settings to minimize interference and noise (see [Section 2.9](#)).

Selected benchmarks As benchmarks we chose the Are We Fast Yet benchmarks ([Marr et al. 2016](#)) (see [Section 2.9](#)) because they are fully deterministic Java programs, representing what we believe to be the best case for Java profilers. Owing to their size, it is feasible for us to investigate any compilation issues and ensure that slowdown is applied accurately.

Selected sampling-based profilers We selected four actively maintained JVM sampling-based profilers for our experiments. We configure 10ms as sampling rate,

i.e., the rate at which they collect call stacks. 10ms is the default rate for most of them, and thus, is what developers would often use.

We evaluate four JVM profilers as seen in [Section 2.10](#). They are as follows: Async-profiler v3.0,⁷ Java Flight Recorder (JFR) as included in OpenJDK 21.0.2,⁸ JProfiler 14.0.5,⁹ and YourKit 2024.9.¹⁰

Hardware and software configuration All experiments were performed on a machine with an Intel(R) Core(TM) i5-10600 CPU with 6 cores and 3.30 GHz, 16 GB RAM, on Rocky Linux 9.4 with Kernel Version 5.14.0. All experiments ran on top of OpenJDK 21.0.2 with the HotSpot JVM. We used our modified version of the Graal compiler, which is build on a version from August 2024.¹¹

Compiler configuration We use Graal in its *libgraal* configuration, which means the compiler itself is ahead-of-time-compiled to ensure optimal compilation times.

To reduce nondeterminism, we use Graal’s support for compilation replay, only JIT compile Java code with Graal’s highest tier setting, and disable background compilation. Compiler replay creates a log file of compilation decisions during execution, which is then used in subsequent runs, to enable the compiler to replicate previous decisions for the same benchmark. This significantly improves the consistency in the resulting machine code across runs, simplifying our experiments. We use this replay feature for all three steps of the process, i.e., preamble, divining, and profiling.

By disabling tiered compilation with the `-XX:-TieredCompilation` flag, only the highest compiler tier is used. While this has performance tradeoffs for production settings, in our case and for the selected benchmarks it does not. It merely avoids us having to support other compiler tiers.

We also disable background compilation, which means that the application thread waits for the compilation to be available when it is triggered. This reduces nondeterminism between application and compiler threads, since application threads do not continue to collect profiling information.

⁷<https://github.com/async-profiler/async-profiler/releases/tag/v3.0>

⁸<https://openjdk.org/jeps/328>

⁹<https://www.ej-technologies.com/jprofiler/agent?version=14.0.5>

¹⁰<https://www.yourkit.com/changes/2024.9/>

¹¹<https://github.com/oracle/graal/commit/049d6d3ab565c74549a590a4b744077a45f7527e>

By reducing nondeterminism, we also ensure that different runs of a benchmark result in the same LIR and native code basic blocks, so that markers are valid and our slowdown can be applied consistently. To ensure that the Graal compiler generates machine code as similarly as possible across runs, we set a few more flags. Minor nondeterminism can result in code being generated at different addresses, which results in Graal adding a different number of NOP instructions for alignment. We disabled the alignment by setting `-Djdk.graal.IsolatedLoopHeaderAlignment=0` and `-Djdk.graal.LoopHeaderAlignment=0`. This simplified mapping LIR blocks to machine code blocks. The performance differs by 1% on our benchmarks, and thus is minimal. For certain language features such as stack overflow checks, the compiler would also add NOP instructions for alignment. Since this leads to a feedback loop with our insertion of slowdown instructions, we disabled it by adding a new `-Djdk.graal.DisableCodeEntryAlignment=true` flag.

Finally, to give profilers the best chance to map native code to source code and correctly attribute run time to methods in a profile, we set the `-XX:+DebugNonSafepoints` flag for all profilers. This ensures that debug information is not only generated for safepoints. Profilers like Async-profiler explicitly enable this functionality themselves to ensure that they can report the best possible profiles. [Section 5.7.1](#) briefly discusses the flag's tradeoffs.

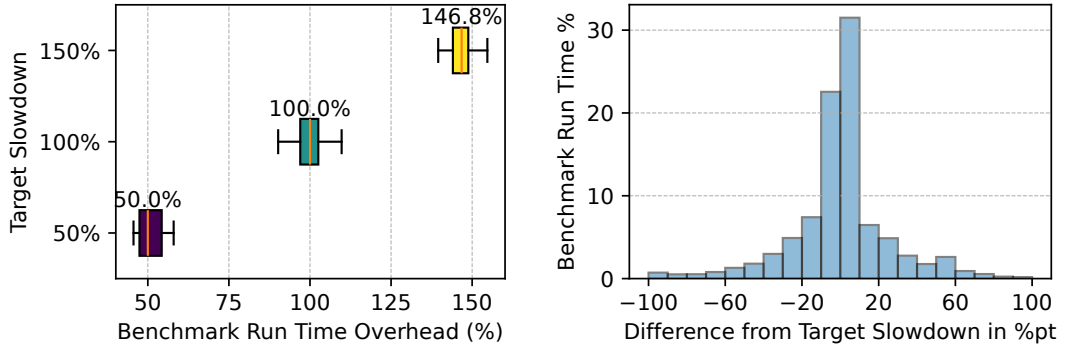
We believe all these precautions together enable a statistically meaningful comparison between profiles, thus allowing us to establish an approximated ground truth that reflects a consistent program behaviour and enables comparisons with a consistent baseline.

5.6 Results

This section presents our experiments and results to answer each of the research questions in [Section 5.5.1](#).

5.6.1 Experiment 1: The Effect of Slowdown on Program and Basic Block Run Time

To answer RQ1, i.e., whether our approach to slowing down programs gives accurate results, we examine its impact on overall benchmark run time and the slowdown at the basic-block level.



(A) Box plot of the median overheads for the 14 benchmarks at 50%, 100%, and 150% slowdown. For each benchmark, we took the median of 10 runs at the target slowdown, comparing it to the median of 10 runs without slowdown. Overall, the benchmarks are successfully slowed down.

(B) The histogram shows the distribution of overhead differences from the target slowdown of 100% across all benchmarks. The x-axis represents the difference in percentage points. The y-axis shows the percentage of benchmark run time. The results show that the majority of run time is spent in blocks with overheads close to the target slowdown.

FIGURE 5.8: The results of experiment 1 show the median overheads at the benchmark level as well as the basic-block level are on target, which means that our approach to slowing down programs preserves program behaviour effectively.

As Figure 5.8a shows, we can successfully slow down the Are We Fast Yet benchmarks and the overall run time increase for these benchmarks is close to the target slowdown. The figure shows median overhead of 10 runs for each benchmark at a target overhead of 50%, 100%, and 150%. The benchmarks were run without using any profiling tools, and report the results as measured by the Are We Fast Yet benchmark harness. The median run-time overhead across all benchmarks is 50.0%, 100.0%, and 146.8%, respectively. Table A.3 in the appendix shows the achieved overheads for each benchmark. The benchmarks with the largest difference from the target are DeltaBlue, with a 139.5% slowdown for a 150% target, and Havlak, with a 115.0% slowdown for the 100% target.

To assess the slowdowns of basic blocks, Figure 5.8b shows a weighted histogram

over all benchmarks for the target slowdown of 100%. The y-axis shows the percentage of benchmark run time the basic blocks contribute. The x-axis shows how close they are to the target slowdown in percentage points. The basic block time was measured with VTune’s hardware-counter sampling support.

We can see that the basic blocks that dominate a benchmark’s run time are indeed close to the target slowdown. However, other blocks can be much slower or faster than they should be. This is often the case for small basic blocks, where adding a single instruction changes the run time significantly. However, we also see that the basic blocks that have too much slowdown are roughly balanced out by the blocks that have too little slowdown. Note, we limited the histogram to the range of -100 to 100% pt for readability, there are a few more outliers, but they have insignificant contributions to a benchmark’s run time. [Figures A.1](#) and [A.2](#) in the appendix show the histograms for the 50% and 150% target slowdowns.

Overall, we can successfully and proportionally slow programs down as intended and answer RQ1 positively. In particular, the basic blocks that make significant contributions to a benchmarks run time are slowed down accurately, and thus, profilers should see them as such. As discussed in [Section 5.2](#), previous work inserted slowdown at the bytecode level. For completeness, [Section 5.7.3](#) reports on our experiments with this approach, and why it would not be suitable in the context of a state-of-the-art JIT compiler.

5.6.2 Experiment 2: The Effect of Block Slowdown on Profilers Reported Methods

To answer RQ2 on whether sampling-based profilers report the same run-time percentage for slowed-down programs, we run the benchmarks with the slowdown configuration files and the sampling profilers. If the profilers are accurate, the percentage of time spent in each method should remain unchanged, given that the most relevant basic blocks achieve the target slowdown.

[Figure 5.9](#) shows the median percentage of run time for each of the five hottest methods of each of the 14 benchmarks on a scatter plot. We compare the method’s

normal percentage (without slowdown) with its slowed-down percentage. Methods where the percentages match as expected appear along the $X = Y$ diagonal line.

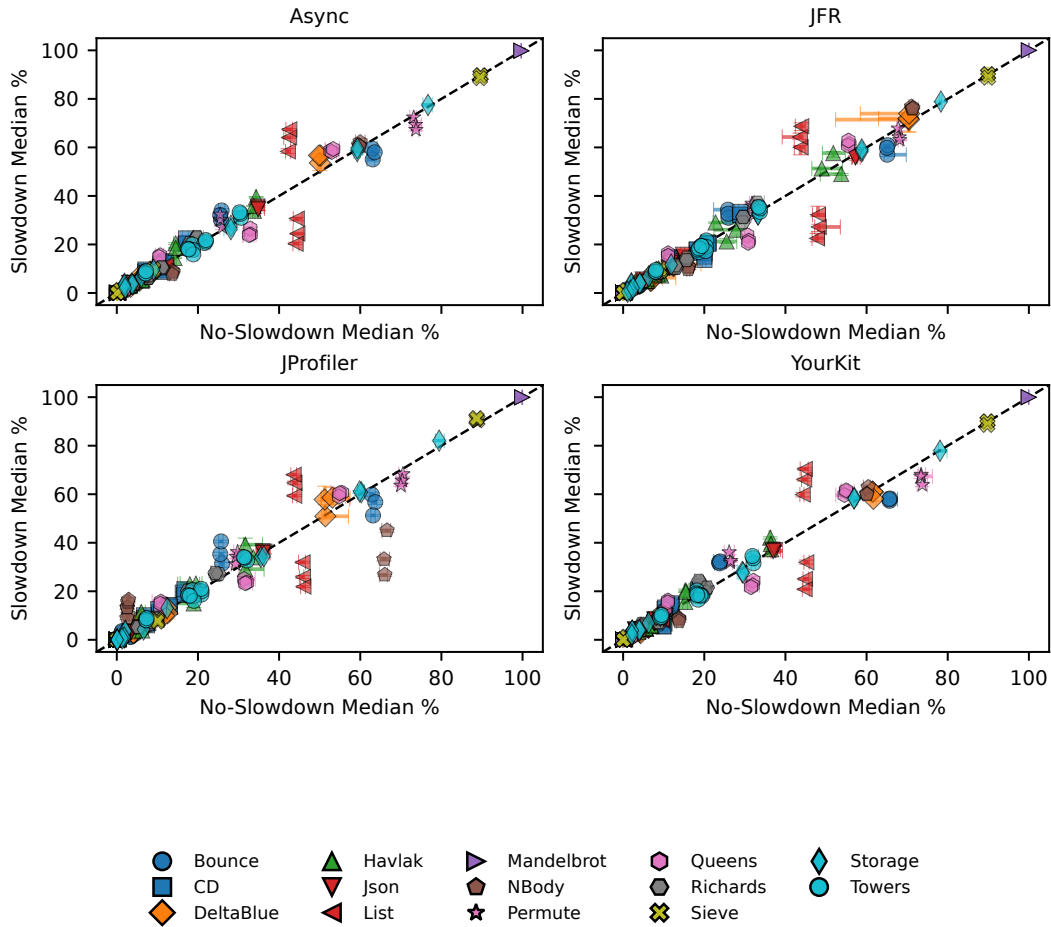


FIGURE 5.9: A scatter plot for each profiler with the median percentage of run time for each of the top five methods in each of the 14 benchmarks. The diagonal $X = Y$ line indicates that a method’s run-time percentage remains the same under slowdown and no-slowdown. The plot shows results for 50%, 100%, and 150% slowdown.

The results indicate that the majority of methods are close to the $X = Y$ line, i.e., the run time percentage profilers attribute to these methods remains unchanged. This suggests that our approach to slowing down benchmarks works regardless of whether a method contributes a smaller or larger proportion of the total run time, and independent of the target slowdown. However, the List benchmark is a clear outlier, and we discuss it further below.

Summary statistics To summarize Figure 5.9 and answer RQ2, we look at the correlation coefficient between the report run times for a method, and analyze how

TABLE 5.3: Correlation and deviation statistics for each profiler. A deviation indicates a change in the percentage of run time attributed to a method between normal and slowed-down runs. If there is no change, the correlation coefficient would be 1.

Profiler	Correlation Coefficient	Deviated >5%pt #Methods (%)	Benchmark of Worst Case	Worst Case Deviation (%)
Async	0.9828	25 (11.85%)	List	24.93
JFR	0.9848	28 (12.02%)	List	25.30
JProfiler	0.9652	31 (14.03%)	NBody	39.25
YourKit	0.9821	29 (12.89%)	List	25.62

strong the deviation is from the expected run time in Table 5.3. By looking at the correlation between slowed-down and normal run time, we can determine how well slowdown preserves the overall run-time proportions. A correlation coefficient of 1 would indicate no change of proportions. JFR has the highest correlation of 0.9848. JProfiler had the lowest correlation coefficient of 0.9652, indicating some deviations but still a strong correlation.

We see that our approach of machine code slowdown maintains the original performance behaviour. It does not significantly alter the underlying program execution and therefore preserves the program’s ground truth. From these results, we conclude for RQ2 that our approach maintains run-time proportions well.

To look closer at the deviations, we count all methods where the run time percentage differs by more than 5%pt. Given that our benchmarking has always some noise, we chose 5%pt is a good middle ground. JProfiler reports for 31 methods (14.03% of all) a run time that deviates by more than 5% from the expected run time, with NBody having the worst case method with about 39% deviation. For the other profilers, the affected number of methods is slightly smaller, but the List benchmark is consistently affected by a deviation of about 25%.

As discussed in Section 2.2.2, profilers can have sampling bias. In Section 5.7.2, we check whether this is an issue for our experiments and find that it is unlikely to have any significant impact.

Inaccurate run time attribution for List benchmark Based on our investigations, the deviation in List is most likely caused by us inserting slowdown instructions without debugging information. As outlined in Section 5.3.4, we mix the slowdown

into the blocks and do not assign debugging information. Instead, we rely on profilers using their heuristics to determine debugging information.

While Graal produces debugging information for most instructions, some end up without. Table 5.4 shows the number of LIR instructions for each benchmark without slowdown. We only include data relevant for the performance of the benchmark, thus, only data from basic blocks that contribute at least 0.001 seconds are included. Note, this data reflects LIR, not machine code instructions. As explained in Section 5.3.4, a single LIR instruction may emit multiple machine code instructions.

TABLE 5.4: LIR instruction debug information ratio for the Are We Fast Yet benchmarks. The table includes only executed compilation units that have at least one block contributing 0.001 seconds of run time.

Benchmark	Comp. Unit	LIR Blocks	LIR Instructions	Instructions With Debug Info	Ratio (%)
Bounce	1	14	256	223	87.11
List	1	27	273	193	70.70
Mandelbrot	1	19	111	84	75.68
NBody	1	6	1115	995	89.24
Permute	1	10	96	69	71.88
Queens	1	65	560	464	82.86
Sieve	1	8	160	139	86.88
Storage	2	37	875	606	69.26
Towers	1	19	320	273	85.31
CD	9	263	4159	3290	79.11
DeltaBlue	5	117	4851	3480	71.74
Havlak	9	319	8327	6208	74.55
Json	10	744	8155	6676	81.86
Richards	5	118	723	615	85.06

As shown in Table 5.4, the List benchmark has the second-lowest ratio of debugging information to total LIR instructions. The hottest basic blocks, which account for most execution time, often lack sufficient debug information, with one of the top five containing none. As these blocks are already prone to misattribution during sampling, introducing slowdown further amplifies this effect, leading to greater misattribution and potentially explaining the observed change. We further investigate this in experiment 3 by assigning debugging information to slowdowns.

5.6.3 Experiment 3: Can Profilers Detect Individual Blocks Being Slowed Down?

Finally, to address RQ3 of whether sampling profilers accurately detect slowdown, we test whether they can detect individual basic blocks being slowed down. We select blocks that contain instructions of only one method and insert slowdown to increase

the total benchmark run time by 5 seconds to assess whether the change is detected, the expected additional run time is measured, and the execution is attributed to the correct method.

As seen in [Section 5.6.2](#), debugging information is required to correctly attribute a program's run time. While some instructions in a block may have debugging information, others may lack it. For such instructions, profilers need to infer the debugging information. As discussed in [Section 2.2.2](#), profilers could simply assume such an instruction belongs to the root method of a compilation unit. However, inlining and code motion can mean the code originates from a different method and such a simple heuristic would misattribute run time.

Experimental Setup

Compared to the previous experiments, this one needs additional setup.

Debugging information To see how profilers handle ideal situations, we assign the same debugging information to our slowdown instructions that the other instructions have. Furthermore, we place slowdown instructions right before the first instruction in the block that has valid debugging information. From the profiler's and compiler's perspectives, debugging information remains unchanged and only the number of instructions increases.

Block selection For this experiment, we identified the basic blocks that have instructions from a single method, based on the LIR instructions source information. Additionally, we considered blocks that consumed at least 0.2 seconds, but preferred blocks that took more time. The goal was to select blocks that are important for the benchmark's execution, and only require a small slowdown to add 5 seconds to the overall run time. This avoids making the experiment unrealistic, and changing the overall performance behaviour, e.g., by introducing pipeline stalling. At the same time, we keep the introduced slowdown lower, which allows us to approximate the ground truth more precisely as argued in [Section 5.5.1](#).

We selected suitable blocks for the experiment manually, which was time-consuming. Consequently, we focused on four benchmarks: two larger ones with multiple computation units, Havlak and Json, and two smaller ones, Richards and List. We included List to further investigate the incorrect attribution of slowdowns. For each of our benchmarks, we selected three blocks as target blocks. The full list is in the appendix in [Table A.4](#).

Target slowdown To make the target slowdown independent of profilers, we aim to add 5 seconds to the time our 500 iterations of a benchmark take. This is a run time increase of 5%–15% on the selected benchmarks and should change the time reported for a corresponding method by 3–12 percentage points. Furthermore, this type of performance change is similar to slowdowns that may be introduced accidentally and developers may want to detect in practice.

Experimental execution For each target block, we ran 10 invocations of the standard benchmark setup with the applied slowdown and 10 without slowdown for comparison. We then used the median run time and the percentage of total run time for each method in our later analysis.

Results

Overall results [Table 5.5](#) summarizes the results. Overall, Async-profiler, JFR, and YourKit detect the changes comparably accurately, where Async-profiler and JFR have a detection rate of 83.33% each. JProfiler however detects only 16.67% of the changes. In the majority of tests, JProfiler either did not detect the target method as contributing any run time or incorrectly attributed the additional run time to a different method. To assess the accuracy of a detected change, we determine a *prediction error*. For the methods JProfiler identified correctly, it reported on average a percentage of run time that is off by 17.91%pt.

Example with mixed results To illustrate the experiment, [Figure 5.10](#) shows the results for one block in the `Vector.hasSome` method being slowed down in the Havlak benchmark. We show the change in the percentage of run time as reported by the

TABLE 5.5: Summary of profiler performance in detecting slowed-down methods.

Profiler	Target Methods	Detected Target Methods	Positive Change Accuracy (%)	Prediction Error (%)
Async	12	11	83.33	1.38
JFR	12	11	83.33	2.18
JProfiler	12	5	16.67	17.91
YourKit	12	11	75.00	1.72

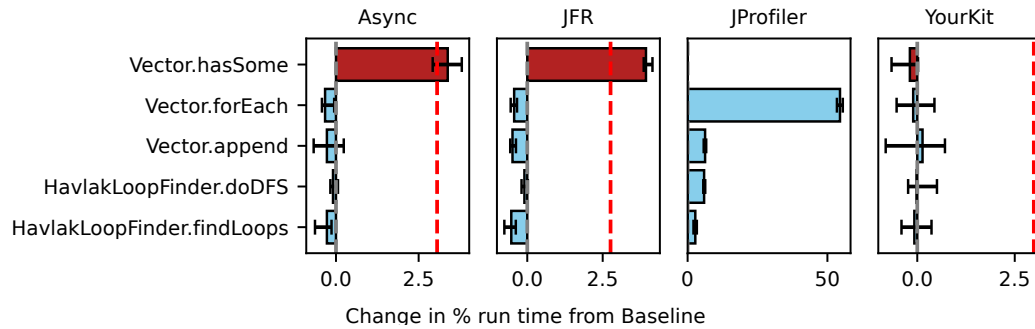


FIGURE 5.10: Change in the percentage of run time with 5 seconds of slowdown added to `Vector.hasSome`. The baseline without slowdown is at 0. The red dashed line indicates the expected change in % of run time. The target method is shown in red.

four profilers. We also indicate the expected change in run time percentage with a red dashed line. Async-profiler and JFR indeed attribute the change relatively accurately to `Vector.hasSome`. JProfiler however attributes the increased run time completely to `Vector.forEach`. YourKit did not manage to attribute the additional run time to any method in particular.

Noteworthy misattributions One of the basic blocks we slowed down is in a Java lambda function, and none of the profilers correctly identified the lambda as target method. All profilers assigned the additional time to other methods. We suspect that incomplete debugging information or incorrect parsing of it might cause the issue.

In other cases, profilers misattributed the additional run time of an inlined method to its caller. This suggests that the profiler did not process all debugging information and merely used the metadata of the overall compilation unit. Thus, one problem might be that profilers are not taking into account all available details.

For the List benchmark, we found that Async-profiler, YourKit, and JFR failed to identify an increase in run time for block 2, within the `List.tail` method and attributed it to another method. They all either reported a reduction in run time

or only a small change, which is at the level of the measurement noise. However, since we set debugging information explicitly, this misattribution must be caused by something else. An alternative cause could be sampling skid. Sampling-based profilers may incorrectly identify the executing method because of a delay between triggering and recording a sample. If this skid introduces bias, sampling results may misrepresent the distribution of execution time and produce inaccurate profiles.¹²

Conclusion To answer RQ3 on whether sampling-based profilers accurately detect slowed-down methods, we find that, depending on the profiler, they can be fairly accurate. However, we also find examples where these profilers have major issues in detecting and attributing the slowdowns accurately. At the same time, when a profiler correctly identifies the target method, our predictions for the expected change in run time are accurate for three of the four profilers. This suggests that our approach to deliberately adding a specific amount of execution time is reliable and produces consistent results. Arguably, this shows that our proposed methodology is valuable and can be used to guide the future development and improvement of profilers.

5.7 Discussion

This section discusses issues that can affect our results or deserve attention in future work beyond this chapter.

5.7.1 Compiler-Generated Debugging Information

Handling debugging information for our experiments turned out to be nontrivial and the following section gives an overview of the concerns.

Tradeoffs of DebugNonSafepoints

In our experiments, we found that setting the DebugNonSafepoints flag (see [Section 5.5.2](#)) to ensure more debugging information is generated is important for JFR's and JProfiler's accuracy. For example, without extra debugging information, JFR found only 7 out of the 12 target methods in experiment 3 (see [Section 5.6.3](#)) and

¹²<https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html>

detected the added run time for only 4 of them. Unfortunately, the additional debugging information uses memory and might slightly increase compilation time, which in turn could result in a slower warmup. Because of the additional memory use, JFR does not automatically enable the flag.¹³ The JFR team aims to provide a tool that can be used in production with only minimal overhead even on pathological cases, and thus, choose other tradeoffs than (for instance) Async-profiler, which enables the flag.

Instructions Without Debugging Information

Despite using the `DebugNonSafepoints` flag, the sampling profilers reported large differences in profiles for the List benchmark, when comparing different levels of slowdown, because of incomplete debugging information (see [Section 5.6.2](#)), which is used to identify the source code a specific machine instruction corresponds to. Unfortunately, inlining in combination with other optimizations can obscure method boundaries, reducing the accuracy of range-based method attribution. For instance, loop fusion can combine loops from different methods possibly resulting in a single instruction implementing multiple source lines from different files. This violates the assumption that there is a 1-to-1 mapping from machine to source code, possibly leading the compiler to drop some debugging information. In other cases, e.g. for garbage collection write barriers or register spill and move operations generated by a register allocator, it may not have debugging information to begin with.

Without debugging information, profilers fall back to the information for neighboring instructions, which can lead to misattribution of run time and thus, poor accuracy. In our experiments, we also saw different profilers attributing run time to different methods in such cases, which suggests that they use different techniques to access debugging information or heuristics to determine the corresponding method.

Challenges of Tracking Accurate Debugging Information

One of the main problems for debugging information is that there does not seem to be a single suitable solution. As mentioned above, compiler optimizations lead to situations where there is no 1-to-1 mapping between machine and source code. A

¹³8307057 JFR: Enable `-XX:+DebugNonSafepoints` when JFR starts <https://github.com/openjdk/jdk/pull/14147>

Java stepping debugger can rely on deoptimization to obtain the needed information, a profiler, however, would want to use the available information without introducing run-time overhead. Similarly, for implementation-level concepts such as garbage collection write barriers and operations generated by a register allocator, different use cases may benefit from different options of how to attribute them to a source program.

To improve the situation for profilers, we would suggest to maintain multiple source locations when sources get merged, and generate debugging information also for instructions produced purely for implementation-level concepts. When an optimization merges source locations, one could preserve several source origins, instead of choosing one arbitrarily or none at all. Profilers could then attribute the observed cost across all origins, yielding a fairer picture of run-time behaviour. Similarly, depending on the use case, even a coarse mapping, such as attributing a register-shuffle to the enclosing basic block would be more useful than leaving the source unidentified.

Debugging Information for Slowdown Instructions

As mentioned above, because of the various compiler optimisations, basic blocks can end up with instructions from multiple methods and possibly without debugging information. To avoid biasing profiles, we need to ensure that slowdown instructions are attributed to the correct method. For experiments 1 and 2, we added slowdown instructions to all relevant blocks evenly distributed between instructions, but did not assign debugging information. This approach gave sufficiently accurate results to slow down the benchmarks. In contrast, for experiment 3 we assigned debugging information from the target method to slowdown instructions and inserted them contiguously, i.e., without interleaving them with the other instructions. This is a best case for profilers and allowed us to test whether a profiler could detect a change in a single method.

For experiment 1 and 2, we considered inserting slowdown instructions and debugging information proportionally based on existing debugging information and the per-instruction run time information we get via VTune from the hardware

counters. Unfortunately, the instruction-level information does not seem precise and reliable enough as discussed further in [Section 5.7.2](#). Thus, we decided to rely only on basic-block-level data, which gave good results. We also considered assigning debugging information to slowdown instructions, but disregarded the idea because it could unintentionally alter the attribution of instructions without debugging information, which would bias the results based on our slowdown insertion.

5.7.2 Hardware- and JVM-Related Factors Affecting Accuracy

The hardware and JVM can also have an impact on accuracy, as we discuss next.

Biased Sampling Skid

Sampling profilers can also misattribute run time because of biased sampling skid. Sampling skid is the delay between signaling the JVM to halt for a sample and collecting the instruction pointer. During this delay, execution continues for a few instructions, advancing the instruction pointer. If the skid is consistent, we still obtain samples that represent how the program spends its time. However, the skid is likely variable and dependent on microarchitectural factors (Xu et al. [2019](#)). Therefore, it can bias samples, which then do not have the uniform probability needed to correctly reflect a program's profile.¹⁴ Some profilers try to mitigate biased skid with more precise timing mechanisms or inducing *jitter* into the delay to increase the randomness of samples (Gregg [2013](#)).

Microarchitecture-Specific Slowdown

As outlined in [Section 5.3](#), the divining process creates a slowdown file specific to the program itself and the machine on which it was generated. If you attempt to use a slowdown file created on one machine for the same program on a different machine, unless both machines share the exact same CPU specification, the slowdown is unlikely to be as accurate and may introduce bias in how slowdown is distributed across blocks.

¹⁴The Pros and Cons of AsyncGetCallTrace Profilers: Error Margin: Skidding + inlining: <https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html>

This is because the divining process relies on timing information from sampling hardware counters to determine how long it takes to execute a given basic block. These timing measurements capture factors such as pipelining, out-of-order execution, memory access latency, and other microarchitect mechanisms. One example of this limitation is our use of register rotations. This technique is unlikely to be transferable between different CPUs, as different architectures have varying amounts and types of registers, along with different policies for register renaming.

As a result, slowdown files would unlikely be accurate when transfers between machines with different microarchitectures. Instead, the divining process must be repeated for each target machine to ensure that the slowdown remains accurately distributed across all basic blocks.

Accuracy of VTune and Sampling Hardware Counters

Our basic-block-level slowdowns rely on the CPU's sampling hardware counters. Although profilers like VTune report timing information at the instruction level, while implementing our slowdown approach, we saw various issues that suggested that for instance *instruction skid* and sampling the pipelined out-of-order execution can lead to misattribution at the instruction level (Xu et al. 2019). Consequently, we rely on timing information at the basic-block level. In our experience, the timing information is sufficiently accurate. Furthermore, our experiments indicate that for most profilers, methods, and benchmarks, the proportion of time spent in a method remains consistent regardless of the slowdown (RQ2). If block-level timing were too inaccurate, we would end up slowing down blocks too unreliably to achieve the observed consistency of slowdowns. Since 13 of 14 benchmarks show no major deviations in the run time percentage of methods, we are confident that we have correctly identified the relevant blocks and proportionally increased their run time. Additionally, our collected block-level timings correlate well with the total run time of each benchmark. If the block-level information were incorrect, we would expect to see significant discrepancies between the actual run time and the targeted slowdown.

Inaccuracy for Small Blocks

Experiment 1 identified the distribution of slowdown overhead across basic blocks. For the blocks that contribute most to total run time, the overhead was close to the target. However, for small blocks, the slowdown can be smaller or larger than required. While in our experiments, this was restricted to blocks that contribute only a minor part of the overall run time, it remains a conceptual issue. These inaccuracies arise from the smallest slowdown we can insert and from the sampling granularity of the hardware counters. For example, adding a single `mov` to a small block may increase its run time by 200% instead of the intended 100%.

Raising the target slowdown, e.g., to 500 %, would bring these blocks closer to the target, but this would also lengthen both the divining and benchmarking phases. In future work, one could experiment with alternative slowdown instructions and with placing them at different positions inside the basic block. Our measurements show that placement matters, likely due to out-of-order execution and instruction retirement. For the very smallest blocks, inserting an instruction, which the micro-architecture can optimize, might let us apply a smaller slowdown. A systematic approach could use throughput prediction (Abel et al. 2022) to select instruction sequences that increase block latency while preserving semantics.

Run-Time Overhead of VTune

We also examined the overhead introduced by VTune during the divining process. There may be concerns that if VTune's overhead is significant, it could impact the accuracy of the timing information for block. To evaluate this, we ran each benchmark 10 times, once with no tools attached and once with VTune in hardware event sampling mode with stack collection, using the exact same setup as in the divining process. Our results show that the median overhead across all runs and benchmarks is 2.35%, with a minimum of 0.29% and a maximum of 5.88%. The results give us confidence that VTune's impact on the benchmark run time is unlikely to significantly affect the accuracy of our results. While some inaccuracies might arise from this overhead, it remains well below the slowdown we introduce. A full breakdown of this overhead per benchmark can be found in [Figure A.3](#) in the appendix.

Using a Static Performance Model Instead of an Instruction Profiler

Instead of relying on sampling hardware counters, we could statistically estimate the cost of instructions based on individual instruction costs (Fog 2022) to determine the needed slowdown. If these estimates were accurate, the slowdown divining process could complete quicker. Unfortunately, these estimates are not reliable enough for our purposes, because they cannot account for changing costs of memory accesses, such as cache misses, out-of-order (Tomasulo 1967) or pipelined execution (Shen et al. 2005), which have a significant impact on performance. Consequently, we rely on sampling of hardware counters, because they provide a more accurate reflection of execution time of each block.

Sampling and Safepoint Bias

One of the key observations of Mytkowicz et al. (2010) was that JVM profilers can suffer from safepoint bias. While safepoints come with debugging information that profilers need, they are not uniformly distributed throughout the program, which makes profiles relying on them less representative of the actual performance behaviour of the program.

Since this may influence our results, we assessed the extent to which the profilers that we used are susceptible to safepoint bias and how it may influence our results. For this experiment, we disabled safepoint insertion in the Graal compiler, essentially removing all safepoints and their debugging information. We then compared the results of our four sampling profilers on the 14 Are We Fast Yet benchmarks with the results from the normal Graal version with safepoints. Each benchmark was run 10 times with and 10 times without safepoints.

The results are shown in Figure A.4 in the appendix. We plotted the absolute change in the percentage of run time attributed to each method. This means that, if methodA originally accounts for 100% of execution time but drops to 0% after disabling safepoints and that time shifts entirely to methodB, we report an absolute change of 200%.

Our findings show that most benchmarks see only a small change of 5–10% on all profilers. Thus, safepoint bias has limited impact. However, JFR and JProfiler

seems to have stronger a safepoint bias on some benchmarks. For Mandelbrot, we see changes of nearly 200%. With safepoints enabled, the profiles attribute most execution time to the `mandelbrot()` method. Without safepoints, the profilers attribute most time to the `innerBenchmarkLoop()` in the benchmark harness. We assume that without the debugging information of the safepoint in `mandelbrot()`, JFR and JProfiler fell back to the source information of the entry point into the compilation unit, which attributes the run time to the wrong method. In JFR's default configuration, safepoint bias is noticeable in Mandelbrot, Storage, Towers, CD, Richards. However, enabling `DebugNonSafepoints`, significantly reduces the bias, leaving only Storage showing substantial change after safepoint removal.

In conclusion, while samplers are not as dependent on safepoints as in the past, benchmarks with inlining and tight inner loops, can still suffer from misattribution. Though, since our approach uses hardware counters instead of JVM sampling, safepoint bias does not invalidate the approach, and only shows up as profiler inaccuracy.

5.7.3 State-of-the-Art JIT Compilers and Bytecode-level Slowdown Insertion

As discussed in [Section 5.2](#), previous work experimented with inserting slowdown at the bytecode level. In a state-of-the-art JIT compiler such as Graal, we find that this approach does not work reliably any more. We experimented with it for completeness and inserted loops computing Fibonacci sequences as described by Mytkowicz et al. (2010), into the hottest methods of our benchmarks, to test whether profilers could detect the slowed-down methods. Because Graal does many optimizations, including loop unrolling, whether and where slowdown was observable depended on specifics of the inserted code, including the number of loop iterations and whether values were constants, and whether the computed sequence is merely stored or used subsequently. For example, for iterations below 1000, Graal combined optimizations such as loop unrolling and constant folding, to compute the result at compilation time. However, with the number of iterations being higher, slowdown was more reliable, but rarely attributed to the correct method. Instead of merely attributing it to the target method, other methods also received additional time, because of inlining

and loop transformations that moved the computation. Furthermore, because it required the use of high iteration counts, the granularity of the slowdown is too coarse and imprecise for our use case. Thus, inserting slowdown after all high-level optimizations, and as one of the final steps in the compilation pipeline (see [Figure 5.3](#)) is more reliable.

5.7.4 Generalizing to Other Systems and Languages

Our methodology is applicable to a wide range of systems and languages. While we performed our experiments with Java, Graal, and VTune, the ideas apply directly to other just-in-time compilers and languages using them, e.g., JavaScript with V8 or Python with PyPy, because we do not rely on any Graal-specific mechanism. Furthermore, because the slowdown injection and profiler-evaluation operate solely at the machine-code and hardware-counter level, one can apply them to ahead-of-time (AOT) compiling systems such as GCC or Clang. VTune can also be replaced, e.g., by Linux's `perf`, to obtain the relevant data. In short, any environment with a compiler malleable enough to insert instruction-level slowdown based on hardware-counter information can adopt our profiler evaluation methodology.

5.7.5 Overall Divining Time and Future Optimization Opportunities

The overall divining process, as detailed in [Section 5.3.2](#), is currently fairly time-consuming. Determining how much slowdown to add to slow down all 14 Are We Fast Yet benchmarks by 100% takes 3.75 days. The divining for all three slowdown speeds required a total of 10.88 days. [Table A.5](#) in the appendix provides a detailed breakdown with the number of compilation units and basic blocks for each benchmark. Per slowdown speed, it shows the number of runs required to fully divine all blocks, the total time it took, the overall number of slowdown instructions inserted (column Total), as well as the median number of instructions inserted into a basic block.

The time required varies and is proportional to the number of blocks and the number of compilation units. The time is proportional to the number of compilation units because our implementation divines each compilation unit separately. This was originally done for debugging purposes. However, divining all compilation units

simultaneously slows down every unit and block at once making each iteration of divining slower. As one approaches the final slowdown amount, each iteration runs very close to the target speed, meaning the fine-tuning stage would likely be more time-consuming.

While slowdown is inserted incrementally, and ramps up as thresholds are hit, future work could explore coarser-grained strategies. For instance, a more aggressive binary search could be implemented, which could reduce the number of runs needed during the divining process.

Faster divining would be useful for future applications of our approach, for instance, for a variation of *virtual speedup* (Curtsinger et al. 2015) or race detection (Endo et al. 2025) based on inserting slowdown instructions.

5.7.6 Experimental Time and Effort

In addition to the divining itself, running all benchmarks for our research questions took about 139h, i.e., 5.79 days. Table A.6 in the appendix shows our estimate and the number of unique runs per research question. Note, the data is based partially on time stamps, and partially on estimation, since not all experiments were fully automated. The *Unique Runs* column represents the number of distinct benchmark executions, regardless of whether a profiler was attached. This data was collected by processing each ReBench file. For example, to answer RQ2 and create Figure 5.9, we ran all 14 benchmarks 20 times each (10 times with and 10 times without slowdown), across 3 slowdown levels and 4 different profilers, which means we needed 3,360 unique runs. Together with the other research questions, this added up to 5,100 runs in total.

5.8 Related Work

Related work on the accuracy of JVM profilers was discussed in Section 5.2. Here, we discuss the wider related work.

The Coz profiler (Curtsinger et al. 2015) predicts the optimization potential of specific code by slowing down specific threads to simulate a *virtual speedup* of the thread that was not slowed down. Instead of stopping threads, our approach inserts

instructions at the basic-block level and aims to assess profiler accuracy instead of optimization potential. Though, our approach could be adapted to realize Coz’s virtual speedup and help developers to estimate optimization potential and prioritize optimizations, too. This might have the benefit of having a finer granularity than what is possible with stopping threads, but is likely to come with other tradeoffs, e.g., the time it takes to *divine* the virtual speedup, which would need to be optimized.

Xu et al. (2019) found that profilers relying on hardware performance counters can attribute execution time to incorrect instructions and functions. The root cause of this issue, as they identified it, is the *skid effect*, a delay between the request for an interrupt and the actual halting of execution. This delay, which can be tens of cycles, allows the instruction pointer to advance beyond the originally intended instruction. They observed that in very hot loops, this effect can cause samples to *skid*, leading to execution time being attributed to different instructions, belonging to other functions. To assess the accuracy of hardware performance counters, they built the *Ground Truth Profiler*, an instrumentation-based tool designed to measure the exact number of retired instructions for a given function without relying on hardware performance counters.

Instrumentation-based profiling with Graal Basso et al. (2023) implemented an event-level profiler inside Graal to record compiler-level events that avoid perturbing compiler optimizations. We built an instrumentation-based profiler within Graal in Chapter 4. Both approaches leverage the information provided by the Graal compiler at compile time to make better decisions on when and where to instrument code, while also reducing interference with compiler optimizations. They place probes as late as possible in the compilation phase to avoid disrupting compiler decisions.

While our approach involves injecting code to introduce slowdowns, we perform this at the LIR stage, which is even later in the compilation pipeline. This provides a strong guarantee that our modifications have minimal impact on the compiler’s optimization decisions.

Utilization of slowdown While various projects utilized slowdown (Curtsinger et al. 2015; Endo et al. 2025; Musuvathi et al. 2008; Mytkowicz et al. 2010; Stoica et al.

2023), we are not aware of any study on the most suitable slowdown instruction. However, there is work around simulators that seems relevant.

Abel et al. (2022) present a basic-block cycle-cost simulator capable of predicting, in a simulated environment, the steady-state cycles per iteration of a given basic block. If such a tool were adapted for our use case, it could be used to slow down blocks on a more fine-grained basis, for instance by exploiting instruction dependencies and reduce the time it takes to determine how many slowdown instructions to insert.

Carlson et al. (2011) introduce Sniper, a full-system simulator that predicts run time by grouping instructions between cache-miss and branch-misprediction events into coarse intervals. This abstraction runs at millions of simulated instructions per second while keeping cycle-count error within roughly 25% of measured hardware. Thus, Sniper could possibly improve the performance of our approach to determining how much slowdown to add.

5.9 Conclusion

This chapter introduces a methodology to approximate the ground truth profile to assess the accuracy of sampling-based profilers for just-in-time compiled systems. We do this by accurately slowing down the execution of a program at the machine code level. Using Graal, a just-in-time compiler for Java, we insert slowdown in the lower intermediate representation, which prevents interference with compiler optimizations and minimizes impact on the performance behaviour at the microarchitectural level.

We explored six x86 candidates for adding accurate, side-effect-free, and distributed slowdown at the basic block level. Of the six candidates, we found NOP and MOV to be the most suitable. They achieved the target slowdown at a program and basic block level, while maintaining the observable performance behaviour of the used benchmark.

Our experiments demonstrate that the approach allows us to slow down the Are We Fast Yet benchmarks to a specific target speed and reach the desired overall run time. We also show at the basic-block level of the native code that the basic blocks that dominate the run time achieve the desired slowdown. We further evaluate the approach using the four Java profilers Async-profiler, JFR, JProfiler, and YourKit to

see whether the slowed-down execution results in consistent profiles. Thus, profiles that maintain the same percentage of program time per method, independent of the applied slowdown.

Finally, we approximate the ground truth profile by slowing down individual basic blocks. The assumption here is that different levels of slowdown, approaching a close-to-zero slowdown would give us the ground truth profile. While the practical limitations of the minimal slowdown being a single added machine instruction prevent us from obtaining a true ground truth profile, we show that we can successfully approximate the ground truth by predicting the changed profile when slowing down a single basic block. We use this to assess the accuracy of profilers by seeing whether the slowed-down basic block is not only detected, but the degree of slowdown is determined accurately. With this approach, we were able to identify discrepancies in profiler accuracy, with profilers like Async-profiler more consistently detecting and attributing slowdowns correctly, while JProfiler in the majority of our tests misattributed execution time and had a higher difference between predicted and measured change in time spent in a slowed-down method.

Our results highlight inaccuracies in modern sampling-based profilers, which is likely caused by incomplete debugging information leading to attributing samples to incorrect methods. This issue is particularly prevalent in highly optimized programs, where compiler transformations lead to missing debugging information.¹⁵

In the next chapter, we implement an instrumentation-based profiler with a focus on loop-level accuracy to measure the CPU time spent in loops. We utilise this chapter's methodology of machine-code slowdown to evaluate its accuracy against that of sampling-based profilers.

¹⁵The evaluated artefact for this work contains the raw data, slowdown files, modified Graal source code, benchmark sources, and analysis scripts used in this chapter (Burchell et al. [2025b](#)).

Chapter 6

Overcoming Profiler Inaccuracy with Late-Compiler-Stage Instrumentation-Based Profiling

The following chapter outlines our enhanced version of the instrumentation-based profiler named BuboL, evaluating its accuracy in detecting the CPU cycles spent in a loop, comparing it to sampling-bases profilers.

6.1 Introduction

With established issues such as bias and inaccuracy in sampling profilers ([Chapter 3](#)), as well as the unrealistic results and impractical overhead of bytecode instrumentation-based profilers ([Chapter 4](#)), there is still a clear need for improved profiling accuracy in highly optimised programs. Deliberate machine-code-level slowdown ([Chapter 5](#)) provides a way to approximate ground truth and evaluate profiler accuracy. This chapter applies that methodology at the loop level to evaluate whether late-compiler-stage instrumentation can provide accurate profiling results with practical overhead. We evaluate this using BuboL, a late-compiler-stage instrumentation-based profiler based on BuboC from [Chapter 4](#).

BuboL can instrument loops and gather accurate cycle counts for execution, as well as execution details such as calling context, debug information, and activation counts. Loops are a core program construct, and in many programs they can constitute the majority of execution time; for example, for our benchmark set, JFR and Async report

that 87.12% of total execution time is spent in loops. This importance is also reflected in work on array access behaviour. Åkerblom et al. (2024) traces 3.8 billion array accesses across the Renaissance benchmark suite and finds that, on average, 69.8% of array access patterns are simple sequential traversals or constant accesses, which are typically implemented as loops over arrays. Therefore, a tool that can accurately determine the amount of time spent in loops would be valuable for performance analysis.

BuboL is a proof-of-concept tool that demonstrates that late-compiler-stage instrumentation can produce accurate results while maintaining program behaviour for loops. However, it is not a complete profiler: it only reports costs for loops that the compiler identifies, so it can miss execution outside loops, for example, straight-line code or call-heavy code that may have long waits. As a result, it cannot provide detailed attribution for the non-loop portion of execution, which can be substantial for some workloads (see [Section 6.2.2](#) for a further discussion).

BuboL achieves a high degree of accuracy when measuring the time spent in loops. Using our approach of deliberate slowdown ([Chapter 5](#)), we are able to determine its level of accuracy by detecting specific amounts of slowdown in specified loops. We benchmark BuboL against sampling profilers such as Async and JFR. To achieve this, we assign each loop its own unique set of debugging information, allowing CPU samplers to report loop-level information. Using that in combination with machine-code-level slowdown, we are able to test BuboL’s accuracy against sampling profilers in terms of how accurately each reports the increased time spent in a given loop. Across our benchmarks, the median deviation is 1.7% for BuboL, compared to 12.1% for Async and 18% for JFR. Concretely, this deviation measures how far a profiler’s reported slowdown for a loop is from the known baseline. BuboL is typically within a few percent of the ground truth, whereas the sampling profilers are often off by a much larger margin. We also observe a clear bias, as the samplers almost always overestimate the time attributed to the slowed loop, while BuboL’s estimates are usually only slightly below or above the expected value. At the level of individual benchmarks, BuboL is more accurate than both samplers in every case, giving consistently closer agreement with the baseline.

BuboL also has a practical execution overhead, with an average overhead of

18.63%. We consider this practical because the slowdown remains below our $2\times$ upper bound. For example, if execution with a profiling tool turns a one-hour benchmark into a two-hour benchmark, we consider this the upper limit of what can reasonably be accepted for routine experimentation.¹ For longer-running applications, we would also prefer that overhead does not scale proportionally with execution time, and ideally that its relative impact decreases. We evaluate this aspect in [Section 6.4.4](#).

This chapter contributes to the thesis by proposing and evaluating a loop-level profiling approach based on late-compiler-stage instrumentation. To evaluate this approach, we use BuboL, to associate cycle counts with loops while also collecting compilation and execution details. We then analyse its overhead and observer effect, and evaluate its accuracy in detecting deliberate slowdown compared against Async and JFR, using VTune as a baseline.

6.2 Motivation for Loop-Level Profiling

[Chapter 4](#) shows that instrumentation-based profilers can suffer from substantial observer effect and overhead. We also demonstrated with late-compiler-stage instrumentation that, if probes can be placed late enough in the compiler to avoid influencing major optimisation decisions, then instrumentation can measure execution while minimising observer effect and maintaining practical levels of overhead. However, one limitation was that we did not determine the accuracy of late-compiler-stage instrumentation.²

6.2.1 Goal of This Chapter

The goal of this chapter is to move further towards improving instrumentation-based profiling on the JVM, building on the late compiler-stage instrumentation introduced in [Chapter 4](#). Using the methodology developed in [Chapter 5](#), this chapter investigates the question of accuracy directly. By using deliberate machine-code slowdown

¹There is no single universal threshold for acceptable overhead. We use $2\times$ slowdown as a practical upper bound because, for a user repeatedly running benchmarks, this is equivalent to running the benchmark a second time. While still costly, this remains practical for many repeated-benchmarking use cases.

²Our methodology of deliberate slowdown had not yet been developed when we evaluated BuboC.

to approximate ground truth, we evaluate whether late-compiler-stage instrumentation can attribute CPU time more accurately than sampling-based profilers. We compare against sampling-based profilers because they provide the most realistic state-of-the-art baseline for CPU profiling in our setting.³ We evaluate whether late-compiler-stage instrumentation can provide more accurate loop attribution than sampling-based profilers, while maintaining practical overhead on the JVM.

6.2.2 Instrumentation of Loops

Profilers typically use methods as the highest-level abstraction for attributing where time is spent during program execution. However, it is unclear whether methods are always the best targets. Compiler optimisations such as inlining, code movement, and loop transformations can mix code from different methods, leaving no clear contiguous section of code belonging to a single method (Section 2.6). Additionally, compilers may not maintain a complete record of source information: it may be unavailable for some executed instructions, and incorrect for others (Section 2.2.2). A more in-depth discussion of why methods are especially difficult to profile will be given in Chapter 7, Section 7.1.

These issues make method-level attribution a difficult target for evaluating profiler accuracy. If code from inlined methods becomes highly interspersed and source information may be incomplete, then an evaluation based only on methods risks measuring the quality of source attribution rather than the accuracy of the profiling technique itself.⁴ We therefore need a program construct that remains identifiable in the generated code after optimisation and is still meaningful for performance analysis.

Loops are an alternative to methods, as they remain meaningful sections of execution while still being identifiable after compilation. Loops often dominate execution time, and therefore are often target of manual optimization by developers. Focusing on loops also lets us test accuracy in a controlled way, because we can apply

³The bytecode instrumentation-based profilers evaluated in Chapter 4 introduced substantial observer effect and overhead, making them unsuitable as accuracy baselines for this experiment.

⁴In Chapter 5, this limitation was controlled by selecting basic blocks whose instructions could only be attributed to a single method.

deliberate machine-code slowdown to specific loops and check whether a profiler correctly attributes the resulting increase in program run time to the intended loop.

Why loops are a meaningful target As discussed above, loops provide a more concrete target than methods for evaluation. While method boundaries can become mixed through inlining and code movement, loops that survive optimisation often retain identifiable low-level structure. For loops that remain after optimisation, back edges and exit conditions still exist in some form, providing concrete locations in the low-level IR where instrumentation can be applied.

Some compiler transformations can change or remove the loop structure of the source program. If multiple source-level loops are merged, we can measure the resulting aggregate loop that remains in the late compiler IR. This means that the reported measurement corresponds to the optimised loop structure that actually executes, rather than to each original source-level loop.

A different case occurs when a loop is fully unrolled or otherwise eliminated. In this situation, there is no remaining loop structure for us to instrument, so we don't report a separate measurement for that loop. We do not expect this to be a major limitation for the workloads evaluated here, because loops that dominate execution usually represent substantial repeated computation and are therefore less likely to be completely removed. Complete unrolling or elimination is more commonly applied to small, simple loops whose cost is unlikely to account for a significant proportion of total execution time.

Loops are also attractive from an instrumentation perspective. A probe placed inside a frequently executed loop body can add overhead on every iteration, whereas probes placed around the loop are only active when entering and exiting the loop, while still measuring the loop as a whole. Nested loops remain an important caveat, since probe cost can still become proportional to the relevant iteration count.

Finally, loop profiling has been explored in related work. Moseley et al. 2007 propose loop-centric profiling and argue that loops can be more informative than methods when identifying opportunities for parallelisation. Jain et al. 2017 present an LLVM-based loop profiler that instruments loops to collect run-time information such as clock ticks and iteration counts.

6.3 BuboL

This section provides an overview of BuboL, detailing its placement as a late compiler-stage instrumentation phase and the instructions that make up the BuboL probe.

6.3.1 Overview of BuboL

BuboL⁵ builds on BuboC,⁶ the late-compiler-stage instrumentation-based profiler introduced in [Chapter 4](#). BuboC instruments whole compilation units by placing probes at compilation-unit entries, exits, and calls, allowing it to record the CPU time spent within each compiled unit while minimising interference with earlier compiler optimisations.

BuboL applies the same late-instrumentation principle to loops. Rather than measuring whole compilation units, BuboL instruments *all loops* that remain within a compilation after optimisation. This allows us to evaluate whether late-compiler-stage instrumentation can measure loop-level CPU time more accurately than sampling-based profilers, while still preserving the program's original performance behaviour as closely as possible and keeping overhead practical.

Probes in the form of RDTSC (Read Time Stamp Counter) instructions are placed at the loop header and at the final loop exits, outside the loop body. So that all time captured between the two probe points covers the full execution of the loop, from the logic that decides to iterate, through the main body of the loop, and to the exits of the loop.

In addition to cycle counts, BuboL records each loop's activation count and captures the relevant debug information and calling context needed to map the recorded loop back to its source level origin. Although our evaluation focuses on the accuracy of loop cycle counts, these additional data demonstrate that BuboL, and any future instrumentation platform built on the same approach, can collect richer information beyond elapsed cycles. Because instrumentation is inserted late in the compiler, it can access post-optimisation information such as inlining decisions, code transformations, loop structure. Being closer to the final emitted code also allows the

⁵*Bubo Loops*

⁶*Bubo Compilations*

instrumentation to make decisions based on the structure of the compiled program that will actually execute. This increases its adaptability and potential usefulness.

6.3.2 Late-Late-Compiler-phase Instrumentation

BuboC’s idea of being placed late in the compiler phase was inspired in part by Basso et al. 2023 and their event profiler, which is also implemented late in the compiler optimisation pipeline. For BuboL, we take it one step further: the logic and placement of our probe takes place in the LIR (Lower IR), whereas for BuboC it is done in the low tier of the HIR (Higher IR).

Placement at the LIR level requires rebuilding our logic using LIR optimisation phases and `LIRInstruction` instead of nodes. While there can still be some sense of a compilation graph, the program logic is presented as a control-flow graph of basic blocks, with each block containing a list of `LIRInstructions`, and each operation emitting machine-code instructions.

Working at the LIR level also required additional engineering effort beyond rebuilding the BuboC logic. In particular, we had to handle details that are normally managed by later compiler stages, including memory allocation, stack-slot assignment, and correctly writing values to memory. At the HIR level, much of this complexity is handled automatically during lowering to LIR. Placing the instrumentation at LIR also provides the benefit of precise control over what machine code is emitted. Because BuboL specifies exactly what is emitted for each probe at the LIR level, we can ensure that no additional compiler-generated logic is inserted to support the probe, which could otherwise occur when instrumenting at HIR level.

Probe placement is now later in compilation compared to BuboC, further reducing the opportunity for our instrumentation to interfere with subsequent optimisations. BuboC’s approach already avoided many major optimisation decisions, but the new placement of BuboL occurs after additional optimisation, including code movement, null-check optimisations, and control-flow optimisations.

Working at the LIR level makes loop instrumentation more reliable because loops are closer to their final machine-code form than they are at the HIR level. At LIR, the basic blocks that contain the loop logic exist more concretely, and the order and paths

of execution are largely fixed. BuboL can therefore place probes on a loop structure that cannot be reshaped by later optimisation phases.

6.3.3 BuboL Probe

BuboL consists of one compiler phase that targets every loop identified at the LIR level. To identify loops, we defer to the Graal compiler's `CFGLoop` class,⁷ which during lowering contains all loops within the control-flow graph of Graal LIR blocks. Using this class gives us the starting header block for every loop and all associated loop blocks and exits, making identification of where to instrument straightforward. This also means that we rely on the Graal compiler to identify what is and what is not a loop.

For each loop, this phase emits two new `LIRInstructions`: `BuboRDTSCToSlot`, which records the start time of the loop, and `BuboRDTSCWriteDelta`, which records the end time and writes the elapsed cycles to memory. Together, these two instructions form the BuboL probe for a loop.

Start probe: `BuboRDTSCToSlot`. As shown in [Figure 6.1](#), `BuboRDTSCToSlot` emits an `LFENCE`, `RDTSC`, and second `LFENCE` sequence on lines 2–4. The `LFENCE` instructions serialise execution, ensuring that the `RDTSC` is not reordered and therefore does not record the timestamp too early or too late relative to the start of the loop logic. The `RDTSC` instruction reads the current cycle count from the timestamp counter, returning its result in the 32-bit `EDX` and `EAX` registers. The probe then combines these two 32-bit values into a single 64-bit value on lines 6–7, and stores the result in a stack slot on line 8. This stack value is later read by the end probe to compute the elapsed cycles for the loop.

End probe: `BuboRDTSCWriteDelta`. `BuboRDTSCWriteDelta` acts as the end part of the probe. It emits its own set of `RDTSC` and `LFENCE` instructions to capture the end time of the loop. As shown in [Figure 6.2](#), this sequence appears on lines 5–7. It then moves the previous start timestamp out of the stack and into a register, shown on line 10. The end timestamp is formed on lines 12–14, where the low 32 bits from `EAX` and the

⁷<https://github.com/oracle/graal/blob/master/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/core/common/cfg/CFGLoop.java>

```

1 Basic_Block_40_Start:
2   lfence ; serialize prior instructions
3   rdtsc ; read TSC -> EDX,EAX
4   lfence ; prevent reordering after rdtsc
5
6   shl rdx, 32 ; shift high 32 bits
7   or rax, rdx ; combine into 64-bit value
8   mov [rsp+32], rax ; store cycle count to stack slot
9
10  ; ... original loop body follows ...

```

FIGURE 6.1: Loop entry probe emitted by BuboRDTSCToSlot. The probe serialises execution, reads the timestamp counter, and stores the start cycle value on the stack.

high 32 bits from EDX are combined into a single 64-bit value. The start time is then subtracted from the end time on line 16, giving the cycles passed since start to finish. This result is then written to memory in our own buffer (allocated when the Graal compiler is initialised) by loading the loop-specific counter address on line 18 and atomically accumulating the delta on line 19.

```

1 Basic_Block_49_Exit:
2
3   ; ... original loop body before ...
4
5   lfence ; serialize prior instructions
6   rdtsc ; read TSC -> EDX,EAX
7   lfence ; prevent reordering after rdtsc
8
9   ; load start timestamp from stack slot
10  mov r9, [rsp+StartSlot] ; r9 = start64
11
12  mov r10d, eax ; copy low 32 bits of end timestamp
13  shl rdx, 32 ; move high 32 bits into upper half
14  or r10, rdx ; combine into full 64-bit end timestamp
15
16  sub r10, r9 ; delta cycles, end - start
17
18  mov r11, [CounterAddress] ; load loop cycle counter address
19  lock add qword ptr [r11], r10 ; atomically accumulate delta

```

FIGURE 6.2: Loop exit probe emitted by BuboRDTSCWriteDelta. The probe reads the end timestamp, computes the delta cycles, and atomically accumulates the result into the loop specific counter.

Both LIRInstruction instances explicitly spill and restore all registers they use into temporary stack slots before and after the probe logic. This makes the emitted probe sequence self contained and avoids clobbering any live register values at the instrumentation point. We do this to minimise unintended perturbation, particularly

avoiding cases where the probe forces Graal to emit additional register moves that could alter the generated code and its performance behaviour. This comes at the cost of increased overhead from the additional spill, restore, and register pressure. The effect this approach has on register allocation is discussed later in [Section 6.4.3](#).

6.4 Evaluation of BuboL

We evaluate BuboL against sampling-based profilers with respect to overhead, observer effect, and accuracy in attributing CPU cycles spent in loops.

6.4.1 Research Questions

We outline here our research questions, which guide our evaluation and assess whether instrumentation-based profilers can accurately record the CPU time spent in loops, compared to CPU samplers, while maintaining the original program behaviour and practical levels of overhead.

RQ1: To what degree does profiling loops with late-instrumentation-based profilers change compiler optimisations? This question tests whether instrumentation, placing additional instructions at the start and end of loops to gather information, has any significant influence on compilation. Such influence could change performance behaviour to the point that the execution with instrumentation no longer resembles the execution without it. So we must first evaluate to what degree we change the compilation.

RQ2: What level of overhead does BuboL incur when instrumenting loops, and to what extent does it change performance behaviour? A key aspect identified in [Chapter 4](#) is that instrumentation-based profilers can cause high overhead. We want to evaluate what level of overhead is incurred when instrumenting loops late into compilation. We then evaluate if this level of overhead changes performance behaviour.

RQ3: Do instrumentation-based profilers and sampling-based profilers record the proportion of CPU time spent in loops accurately? We evaluate which profiler

most accurately attributes changes in loop execution time. To test this, we will use our deliberate slowdown technique of [Chapter 5, Section 5.6.3](#). We will selectively slow down loops in our benchmark suites and test which of Bubo, Async, and JFR, is most accurate in detecting the degree of increase in time spent in those loops.

6.4.2 Experimental Setup

The following subsection details the benchmarks, sampling-based profilers, hardware, and compiler setup used during experimentation for these results.

Benchmarks Our benchmark set is a selection from the Are We Fast Yet (AWFY) benchmark suite (Marr et al. 2016). We specifically use a subset of the benchmarks that contain a variety of loops, since not all AWFY benchmarks contain loops. The benchmarks evaluated are: Sieve, NBody, Mandelbrot, Json, and Bounce.

Across these five benchmarks, we identify 17 source-level loops, but we evaluate a total of 11 loops. This difference arises because not every source-level loop is suitable for our accuracy evaluation. We include only loops that contribute a meaningful proportion of total run time, and only loops that do not contain non-inlined calls.

Significance and compiler-created loops We include in the overall accuracy measurement only loops that account for at least 2% of run time.⁸ This avoids measuring smaller and faster loops, whose results are more unstable and for which noise or non-determinism can be misconstrued as meaningful change. Additionally, smaller and faster loops can be created as a result of compiler optimisations. Loop splitting can produce many small loops that contribute very little to overall execution time. Although these loops are instrumented, their measured time is often below the 2% threshold. We also observe that many of these loops contribute so little to execution time that sampling-based profilers miss them entirely. Excluding such loops therefore makes the comparison more favourable to the sampling profilers, since it reduces the chance of evaluating loops that are missed simply because of sampling bias or the probabilistic nature of sampling.

⁸This threshold applies only to the accuracy evaluation: for the observer-effect analysis, we also evaluate outer loops that may contribute less than 2% of total run time, especially when those loops contain nested loops and are assessed using their exclusive time.

Loops that contain no non-inlined calls We only evaluate loops that do not contain non-inlined calls to other parts of the program. Non-inlined calls add an extra level of complexity because, once execution enters a callee, the instructions being executed are no longer part of the caller’s compilation unit, and the loop-specific debug information we assign to the caller’s loop body no longer applies. At that point, attributing callee time back to the calling loop is only possible if the profiler can reliably recover the full call stack and maintain the correct loop context across the call and return. For sampling profilers in particular, their accuracy becomes heavily dependent on the availability and quality of debugging information and stack unwinding, and it is not something we can enforce purely by rewriting loop debug locations in the caller. To avoid making debug-information quality and stack recovery a confounding factor in our evaluation, we exclude loops containing non-inlined calls. This is another factor that should give the sampling profilers a better chance of being accurate when evaluating the amount of execution time spent in loops.

To add to the set of loops and benchmarks we evaluate, we also implemented LoopsBench,⁹ a benchmark suite we created that contains 14 loop-focused workloads. Each benchmark is designed around loops as the primary source of execution time, for example Bubble Sort, Monte Carlo, and Matrix Multiply. LoopsBench also avoids calls that may not be inlined. It therefore provides a larger set of loops that can be evaluated, increasing the total number of evaluated loops from 11 (from AWFY) to 25 overall.

Sampling profilers We compare BuboL against Java Flight Recorder (JFR) and Async-profiler (Async). JFR is particularly relevant because it is integrated into HotSpot with access to dedicated instrumentation and sampling framework. Async-profiler provides a low-overhead alternative that focuses on accuracy and reducing profiling bias. Async and JFR also provide slightly different forms of sampling; Async-profiler uses OS signals and timers to trigger samples, whereas JFR coordinates its sampling through the JVM and its managed threads. In our earlier evaluations in [Chapter 3](#) and [Chapter 5](#), both of these tools also showed the highest consistency

⁹A full list of the benchmarks is given in [Table A.7](#).

and accuracy among the sampling profilers we tested, making them suitable for comparison because they provide the best sampling-based accuracy in our evaluation.

Hardware All experiments were performed on a machine with an Intel(R) Core(TM) i5-10600 CPU with 6 cores and 3.30 GHz, 16 GB RAM, on Rocky Linux 9.4 with kernel version 5.14.0. All experiments ran on top of OpenJDK 25.1 with the HotSpot JVM. We used our modified version of the Graal compiler, which is based on a December 2025 source revision.¹⁰

Compiler setup We make use of compiler replay, which is the mechanism the Graal compiler provides to log compilation decisions made during a compilation. This log can then be used for future runs, in order to provide a more consistent and deterministic compilation.

For the evaluation of sampling profilers, additional debug information is required so that we could identify which regions of execution corresponded to loops. To achieve this, we modified the OpenJDK used to build Graal to include a set of compiler markers. These are additional methods that are available during compilation. They allow us to assign loop debug information to an identifying method that denotes a loop. This enables loops to be identified by sampling-based profilers. For samplers themselves, we also included `-XX:+DebugNonSafepoints`, as we found in [Chapters 3 and 5](#) that it resulted in better JFR results. In addition, we assign this loop-specific debug information to all instructions, regardless of whether they have debug information in the first place, therefore overwriting any existing debug information. We do this to give sampling-based profilers the best opportunity to identify the loop from debug information, rather than requiring them to approximate the method being sampled, as they must do when debug information is missing.

When running VTune, or evaluating VTune results, we use the same setup described in [Chapter 5](#). We configure Graal in `libgraal` mode and again use compilation replay to ensure that machine code generation is as consistent as possible across runs. To reduce nondeterminism between application and compiler threads, we disable background and tiered compilation, ensuring that the highest optimisation tier is

¹⁰<https://github.com/oracle/graal/commit/0b3e44cfdce6db641a665d6c4f425dad24be8303>

used as early as possible. We also disable various alignment mechanisms that would otherwise introduce minor code layout differences, such as additional NOP instructions, since these can interfere with our inserted slowdown instructions and block mapping. To summarise their purpose, these compiler options aim to produce more stable machine code across runs. This gives us a more reliable and consistent compilation result, ensuring that the machine-code-level slowdown is applied to the same generated basic blocks each time.

6.4.3 Does BuboL change compiler optimisations?

To address RQ1, namely whether profiling loops with late-instrumentation-based profilers changes compiler optimisations, this section discusses and demonstrates the effect that BuboL has on compiler optimisation in the Graal compiler.

Placement of BuboL in the Graal pipeline

As described in [Section 2.5](#), Graal lowers Java bytecode through multiple IR levels before emitting machine code. BuboL inserts its instrumentation late in this pipeline, at the LIR level, immediately before register allocation. This placement is intended to minimise interference with earlier optimisation decisions. This gives the instrumentation access to the final loop structure used by the compiler, since no later phase will reorder, reshape, or remove the loops that BuboL instruments. The remaining phases after BuboL are discussed later in this section.

Optimisation phases before BuboL If the decisions the compiler makes when performing its optimisations happen without instrumentation probes ever existing, or having any influence, then those decisions remain unchanged. From the point of view of major optimisations such as inlining, loop transformations, and constant folding, BuboL has no effect, as it does not exist at that point in compilation. [Figure 6.3](#) shows how the major optimisations take place before BuboL phase, placed late into the Low-level optimisations

The only way BuboL can have any influence is via the extra time spent in our compiler phase, which introduces a small delay within the compilation pipeline. This could, in

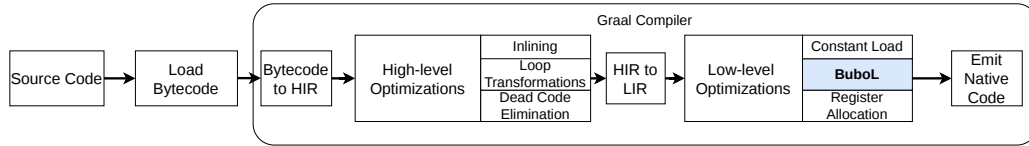


FIGURE 6.3: Placement of BuboL in Graal’s LIR pipeline: the BuboL phase runs after the low level LIR optimisations and immediately before the register allocation and register fixing stages.

Phase	Description
EdgeMoveOptimizer ¹¹	Optimises move operations that occur on control flow edges, reducing unnecessary data movement between basic blocks.
RedundantMoveElimination ¹²	Eliminates redundant move instructions that do not change program state.
NullCheckOptimizer ¹³	Optimises null checks by removing explicit checks when possible.
ControlFlowOptimizer ¹⁴	Simplifies control flow by reducing unnecessary jumps, removing unreachable blocks.
ComputeCodeEmissionOrder ¹⁵	Computes the final block emission order for machine code generation.
LocationMarkerPhase ¹⁶	Inserts location markers for debugging supporting debug information and metadata.

TABLE 6.1: Graal LIR phases executed after BuboL instrumentation.

principle, shift when subsequent compilations begin, meaning the next whole compilation might be started very slightly later. This delay is on the order of microseconds, so no measurable behavioural change is expected. Regardless, this delay is a known (albeit negligible) scheduling effect. To negate this effect, background compilation is disabled in our own evaluation, ensuring that compilation is completed before execution continues.

Optimisation phases after BuboL The optimisation phases that occur after BuboL may possibly be affected. Register allocation immediately follows the BuboL phase. This phase uses a linear scan approach, which greedily assigns registers. Since BuboL emits RDTSC instructions which occupy the RDX and RAX registers, some effect on register allocation is expected, which we will discuss shortly in [Section 6.4.3](#).

After the register allocation stage, the compiler enters the post-allocation and final code stage. The phases are included in these two stages, and are shown in [Table 6.1](#). Out of these six compiler phases, none can be influenced by the presence of a BuboL probe in the control flow.

EdgeMoveOptimizer optimises moves on control-flow edges, whereas BuboL emits its probe logic inside existing basic blocks and does not introduce new jumps or edges. This also means that ControlFlowOptimizer should have no influence over the BuboL probe, since there is no new control flow to simplify or remove.

RedundantMoveElimination may see the move instructions emitted by BuboL, but these moves are required to combine timestamp values and write results to memory. They are therefore not redundant program moves that can be removed.

For NullCheckOptimizer, ComputeCodeEmissionOrder, and LocationMarkerPhase, the interaction with BuboL is limited. BuboL does not emit null checks, so there is nothing for NullCheckOptimizer to optimise. ComputeCodeEmissionOrder and LocationMarkerPhase operate on code layout and debug metadata. While BuboL probes emit no debug information¹⁷ and should not introduce additional debugging metadata, no change in the resulting program behaviour is expected from these phases.

Overall, these six compiler phases give us high confidence that, after BuboL has inserted its probes, later compilation stages do not reshape the instrumented program. The main remaining interaction is register allocation.

BuboL register allocation BuboL requires registers to execute its probe logic. This is required for the RDTSC, since the instruction returns its timestamp value through RDX and RAX.

BuboL requests two temporary registers from the Graal compiler. These registers may be different registers at different points in execution, but are free to use at the point where the BuboL probe is executed.

¹¹<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/EdgeMoveOptimizer.java>

¹²<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/RedundantMoveElimination.java>

¹³<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/NullCheckOptimizer.java>

¹⁴<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/ControlFlowOptimizer.java>

¹⁵<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/ComputeCodeEmissionOrder.java>

¹⁶<https://github.com/oracle/graal/blob/9744151499221d3e36bc88f736d0b1f32a43ab96/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/lir/dfa/LocationMarkerPhase.java>

¹⁷By default, BuboL emits no debugging information. For RQ2, where we measure BuboL's observer effect, we do emit debugging information for the BuboL probe. However, this is done after the LocationMarkerPhase, so it does not introduce any side effects on those optimisation decisions.

However, there is no guarantee that at least two registers are always free at any given point. In some cases, register allocation will spill one or more registers in order to provide temporary registers for the probe. The spilled values are stored to the stack and later restored. This means that BuboL can introduce additional register moves around the probe, potentially several instructions depending on how many registers are free, which would not be present if the probe were not emitted. These moves are emitted as spill and restore code before the probe, rather than being part of the probe itself. These additional moves, if emitted, can cause a change in performance because the CPU spends additional time executing them. In that sense, BuboL does change the generated program.

However, these spill and restore instructions are emitted immediately around the probe logic and do not alter control flow or change which basic blocks are executed. Their cost is therefore part of the probe overhead itself, rather than new work introduced into the underlying loop body. Consequently, while they may increase the cost of instrumentation, the time spent in each part of the program should not shift much due to these spills and restores.

In regard to the six phases discussed in [Table 6.1](#), no effect on these register allocation related moves is expected. These moves introduced are necessary, meaning they should not be optimised away, removed, or moved.

Overall, in answer to RQ1, BuboL has a minimal effect on compiler decisions. The only identifiable change is to register allocation, caused by BuboL requesting temporary registers for its instrumentation so that the allocator can select available registers and reduce spills and restores. Placing the instrumentation as late as possible in the pipeline limits its effect on earlier core optimisations, including inlining, loop transformations, and dead code elimination.

6.4.4 BuboL's Overhead

Overhead is an important aspect of any profiling tool. If the overhead of using a tool is too large, it can become impractical in use cases where the program being profiled runs for a long time. Instrumentation also competes with sampling in terms of both accuracy and overhead. While sampling will likely always have lower overhead,

reducing the overhead gap between sampling and instrumentation increases the range of use cases where instrumentation is viable.

BuboL achieves reductions in overhead partly because it does not perturb compilation. As shown in [Chapter 4](#), a key factor behind the high overhead observed in instrumentation-based profilers such as JProfiler and YourKit is interference with optimisations such as inlining, which can incur major performance losses. Since BuboL uses late compiler-stage instrumentation, it avoids interfering with major optimisations such as inlining, thereby addressing one of the main sources of overhead identified in [Chapter 4](#).

However, BuboL probes still incur direct execution cost. The probes consist of emitting an RDTSC instruction, which is an expensive machine code instruction (Fog [2022](#)). On our benchmark machine using the Cannon Lake microarchitecture, the cycle cost is approximately 18 cycles. This is expensive in terms of single instruction cost. In addition, LFENCE is emitted to cause serialisation, along with a small number of move instructions to place results onto the stack and into memory. The cost of the probe is therefore non-negligible, and BuboL necessarily incurs overhead.

For this overhead experiment, we include the full set of 14 benchmarks from AWFY as well as LoopsBench. The non-inlined call restriction for the accuracy evaluation does not prevent us from measuring overhead, and using the full set allows us to observe BuboL's cost across a wider range of program behaviours, including loop-heavy, non-loop-heavy, and recursive benchmarks. The overhead of BuboL for the 14 Are We Fast Yet benchmarks, plus LoopsBench, is shown in [Figure 6.4](#). Across all 15 benchmarks, the overall median overhead is 20.2%. The worst case is Permute with 279.5%, and the best case is NBody with an overhead of 0.08%. The List, Queens, and Towers benchmarks are excluded since their overhead was zero or negative. This is due to them not containing any loops, and therefore our instrumentation had no effect, and a combination of JVM and CPU non-determinism caused them to run faster.

For the benchmark set we use to evaluate accuracy and observer effect, highlighted in red, the median overhead is 18.6%, with the worst case being Sieve with an overhead of 38.2%. The reason Permute and Storage have a high overhead compared to the other benchmarks is due to recursion. Both benchmarks contain loops inside

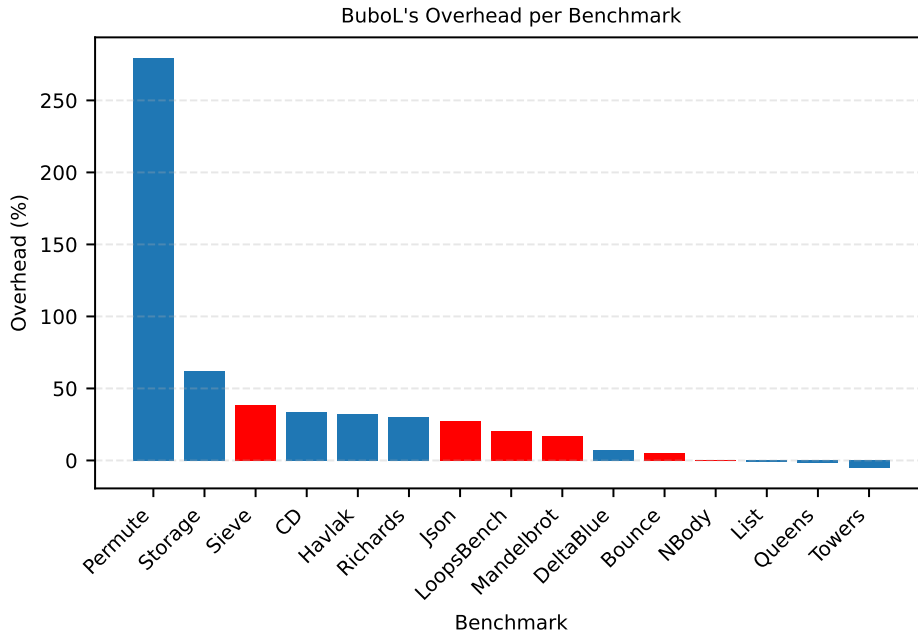


FIGURE 6.4: BuboL’s overhead for the 14 Are We Fast Yet benchmarks plus Loops-Bench. Benchmarks highlighted in red are used later in this chapter. Across the 15 benchmarks shown, the median overhead is 20.2%, ranging from 0.08% (NBody) to 279.5% (Permute). For the subset used later in this chapter (red), the median overhead is 18.6%, with a worst case of 38.2% on Sieve

recursive methods, and the recursive nature means the probe points for loops inside the recursion are activated very frequently.

For example, in Permute the main loop inside the recursion had an activation count of 6.3 billion in approximately 19 seconds of execution. With each activation performing an RDTSC as well as the other probe logic, this explains the large overhead. In contrast, for Sieve, the loop with the highest activation count is activated 3.5 million times in 12 seconds of execution, which is far smaller, and this is reflected in its overhead.

As one would expect, the cost is strongly linked to probe activation count, rather than being an unavoidable fixed cost of BuboL. Since BuboL currently instruments every detected loop without mitigation, the measured overhead should be treated as an upper bound. Future versions could reduce overhead through selective instrumentation, for example by avoiding loops inside deep recursion, applying activation thresholds, or using lightweight estimation strategies. We demonstrated the benefit

of these techniques with BuboC in [Chapter 4](#), where instrumenting only compilation units that meaningfully contribute to execution time reduced overhead while preserving useful accuracy.

To address the first part of RQ2, we evaluate the overhead introduced by BuboL when instrumenting all loops. For the subset of benchmarks used to evaluate accuracy and observer effect, BuboL has a median overhead of 18.6%. Across the full set of 15 benchmarks, which includes worst-case scenarios such as recursion in *Permute*, the median overhead is 20.2%. Although overhead can become high when probes are activated extremely frequently, as seen in recursive benchmarks such as *Permute*, the benchmarks in the evaluated subset remain far below our practical-overhead threshold of $2\times$. This is with BuboL instrumenting all loops, without mitigation strategies to reduce probe activation frequency. This suggests that even an all-loop instrumentation strategy can remain practical in terms of overhead.

6.4.5 BuboL’s effect on performance behaviour

We now consider the second part of RQ2: BuboL’s effect on performance behaviour. This evaluation tests whether the overhead introduced by BuboL changes where execution time is spent in the program, and to what degree this could affect our accuracy measurements.

In [Section 6.4.3](#) we established that BuboL causes minimal changes to resulting compilation decisions, with the only change being additional moves incurred as overhead. Then in [Section 6.4.4](#), we measured the overhead of BuboL on our benchmark set.

The next step is to evaluate whether this overhead itself introduces a significant change in performance behaviour. To do this, we evaluate the overall performance behaviour of a selection of benchmarks, before and after applying BuboL. Additionally, a concern is that the presence of nested loops, meaning a loop inside a loop, could change performance behaviour if both loops are instrumented. In such cases, the BuboL probe could disproportionately affect the inner loop, for example making it appear more significant than it should be due to the added overhead of BuboL itself.

Therefore we evaluated the extent to which the BuboL probes measure themselves, and to what degree there is an observer effect in such scenarios.

Measuring BuboL’s change to overall performance behaviour To measure overall performance behaviour, we use the Async profiler, sampling the entire program without BuboL attached, and then with BuboL attached. We then measure how the distribution of samples across methods changes. If BuboL incurs additional performance change in other areas of the program, beyond its own overhead, then an increase in samples would be expected in those areas.

To avoid sampling BuboL itself for this experiment, the logic emitted by BuboL is given its own separate debugging information for this experiment only. Therefore, in an Async profile, we can identify when a sample has hit BuboL’s probes. By removing these samples from the profile, the direct overhead of BuboL is removed.

When considering only the samples attributed to BuboL’s probes, the proportion of time reported by the sampler aligns with the overhead we expect for each benchmark, as previously shown in [Section 6.4.4](#). The remaining change in the number of samples attributed to each method should therefore reflect changes in performance behaviour caused by BuboL, excluding its own direct overhead.

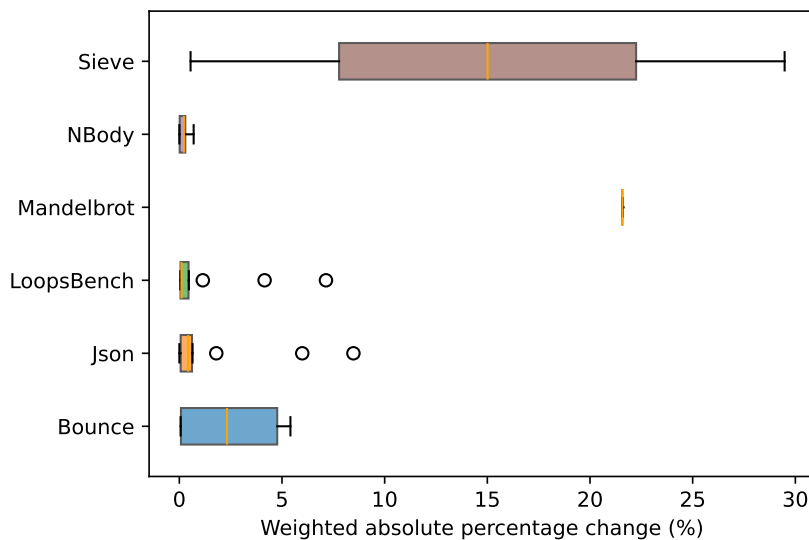


FIGURE 6.5: A measure of BuboL’s effect on our benchmark set, weighted absolute percentage change in execution sample distribution per benchmark when BuboL is enabled, measured with Async.

Figure 6.5 shows, for each benchmark, how enabling BuboL alters the distribution of Async-Profiler samples across methods. Each benchmark was executed three times with BuboL off, followed by three executions with BuboL on, with Async attached in all runs. For every method reported by Async, we first compute the median number of samples across the three “BuboL off” runs and the median across the three “BuboL on” runs. The difference between these two medians gives the method-level percentage change in samples.

Because change in methods that contribute more to overall execution time are more significant than changes in methods that contribute little to overall execution time, each method’s percentage change is weighted by its share of total samples. For example, suppose Method A accounts for 70% of the total execution and its sample count increases by 5% after instrumentation. Under weighting, its contribution becomes 3.5% overall change. Suppose Method B accounts for only 5% of total execution, but its samples increase by 60%. Although 60% is a large relative change, weighting reduces its overall impact to 3%, because the method accounts for only a small fraction of execution. Weighting therefore reflects how much each method actually influences the behaviour of the whole program, rather than treating all methods as equally important.

For each benchmark, Figure 6.5 summarises the resulting per-method weighted absolute percentage changes. Each box therefore represents the distribution of changes across methods within a benchmark. Larger values indicate that enabling BuboL causes a greater change in performance behaviour, whereas smaller values indicate that the overall distribution of execution time across methods remains more stable.

Using Figure 6.5 to evaluate BuboL’s overall program performance behaviour change, a measurable observer effect from BuboL is present. As expected, the cost of executing RDTSC and the serialising LFENCE introduces delay, which can affect the distribution of samples across the program. The effect is most apparent in regions of execution immediately before and after probes. This penalty incurs additional cost, meaning the sampler collects more samples in areas around the probe.

The overall median weighted absolute percentage change across all benchmarks is 1.9%. The lowest is NBody, with a median absolute percentage change of 0.2%, and the highest is Mandelbrot, with a median change of 21.5%, followed by Sieve with a

median percentage change of 15%.

A possible explanation for why Sieve and Mandelbrot show such large increases is that each benchmark consists primarily of a single method containing multiple loops. Mandelbrot consists of only one method, while Sieve has three, with one method contributing 97% of the total run time for Sieve. As a result, the additional delay from serialisation and any knock-on effects introduced by the probes, are concentrated within the same method for all probes. This means the additional cost incurred by the probes is not distributed across multiple methods, leading to a larger observed change.

Besides Sieve and Mandelbrot, the remaining benchmarks exhibit substantially smaller median weighted percentage changes, below 5%. Thus, the larger observer effects are concentrated in two benchmarks rather than being representative of the whole set. LoopsBench (which consists of 14 sub-benchmarks covering a range of loop structures) its median percentage change is only 1.03%, suggesting that in a wider variety of loop-specific workloads the observer effect remains small. Taken together, this indicates that the effects observed in Sieve and Mandelbrot are specific to their program structure.

Overall, enabling BuboL does affect whole-program behaviour, primarily due to the inherent cost of executing the probes. For most benchmarks this effect is small once the direct probe overhead is excluded, but Sieve and Mandelbrot show more substantial changes. These results show that BuboL's effect on whole-program behaviour is usually limited, but can become significant for some program structures. We discuss this point, together with the loop-level observer-effect results, in [Section 6.4.6](#).

6.4.6 BuboL's observer effect on loops

A factor that is particularly important for a profiler whose goal is to accurately measure time spent in loops is the influence that the profiler exerts on loops themselves. In particular, a nested loop scenario raises an additional question. When an outer loop is probed and an inner loop is also probed, the measured time for the outer loop may include overhead introduced by the inner probe. The key question is the degree

to which the outer loop records time that would not normally be present if the inner probe were not instrumented. Therefore, we conducted an experiment to measure the effect on the outer loop’s reported time when inner probes are removed.

We ran all benchmarks three times with BuboL instrumenting all loops, as it does by default. We then ran each benchmark a further three times, but this time only instrumenting loops that were not nested inside other loops. Comparing the difference in the outer loops’ results will show what effect BuboL’s probes themselves have on our measurements. We hope to determine to what degree we measure our own overhead, and how this may influence our results when we come to evaluate BuboL’s accuracy in RQ3 (Section 6.4.7).

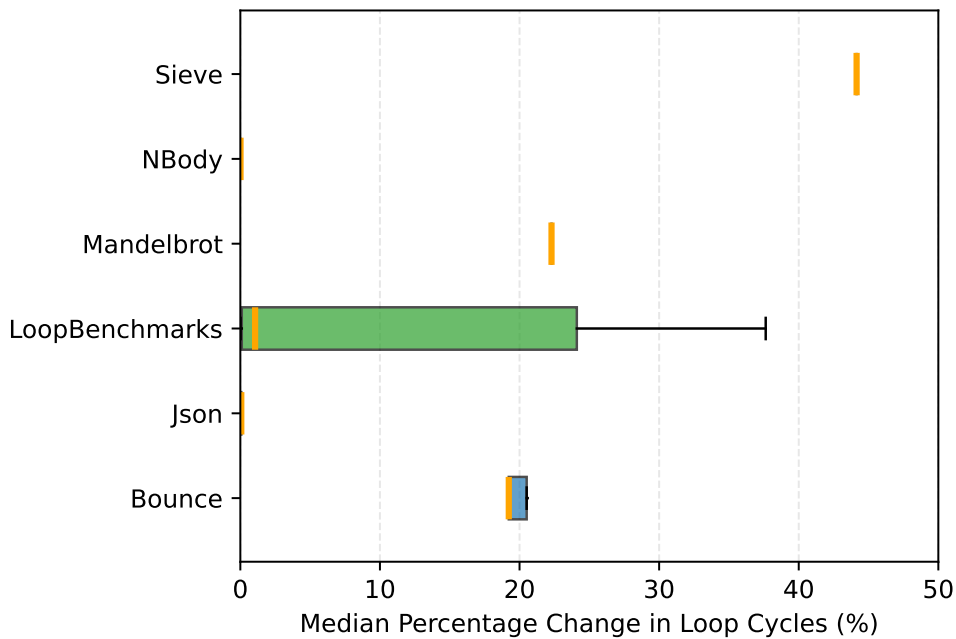


FIGURE 6.6: A measure of BuboL’s effect on the outer loops in our benchmark set: the absolute percentage change in cycles recorded for each benchmark’s outermost loops when we also instrument inner loops. An increase indicates how many additional cycles are now recorded in the outer loops because the inner-loop probes are present and recording. Note that, in the majority of benchmarks, the outer loops account for only a small proportion of total run time (after subtracting the inner loops’ run time), so a large percentage change may still correspond to a small change in the total number of recorded cycles.

For Figure 6.6, we show, for each benchmark, a box plot of the absolute percentage change in loop cycles the outermost loop is measured without and with nested-loop instrumentation. Each data point in a benchmark’s distribution is one loop that

exists in both configurations, using the median cycles across the three runs for each configuration.

We find that our overall median percentage change is 0.14%, with the two highest being Mandelbrot at 22.2% and Sieve at 44.1%.

Figure 6.6 shows the extent to which the outermost probes end up measuring the overhead of our own probes, and how much bias we can incur in small, tight loops where the probe cost accounts for a significant portion of the run time.

Sieve and Mandelbrot show how this bias can occur. Both benchmarks contain multiple loops nested within each other, and they show that, at least for their outermost loops, a portion of the time being measured is time spent in the inner loop probes, or due to the additional performance effects these probes can introduce. But an important factor that should be noted is that, for Mandelbrot and Sieve, the outermost loop is not the most important loop in terms of time spent. For Sieve, the outermost loop accounts for 3.2% of the total exclusive run time, and for Mandelbrot it accounts for 0.01% of the exclusive runtime. So, for these two loops, even if they have changes of 22.2% or 44.1%, the effect on the overall measurement will not be to a great degree. But nonetheless, if one's goal is to measure these outer loops regardless of their significance, then their observer effect is still important.

Therefore, there is a concern that the results we measure from the outermost probes can be misleading, because the time in the outer loop, is not purely time spent doing the outer loop's workload. This can make measurements for the outermost loops inaccurate, and it can bias our results in a way that makes some loops look more expensive than they really are. This issue is most severe for tight, smaller loops where the cost of the main body of the loop is less than, or comparable to, the probe overhead. In that case, the activation cost of the probe becomes a significant part of what is being measured, so the loop's measured time is inflated simply because it was instrumented. That means we are partly measuring BuboL itself rather than the benchmark itself, which is a problem if we later use those measurements to judge accuracy in RQ3.

A positive here is that LoopsBench, which is a single benchmark but which contains 14 loop workloads, shows a low median change overall. While some workloads still exhibit larger effects, the median and 75th percentile remain small, suggesting that

this bias does not dominate across all the loops evaluated in LoopsBench. We use LoopsBench here primarily to cover a broader range of loop structures than the AWFY benchmarks contain. The larger outliers occur in workloads with multiple levels of nested loops, where the outermost loop shows the greatest change because its measured time accumulates the overhead of multiple inner probes when all nested loops are instrumented.

There are strategies to mitigate this. First, if the cost of the probe is roughly known, and because BuboL records activation counts, then we could subtract an estimated probe overhead from the measured totals. This could help separate “real loop work” from “probe cost”, and produce a more accurate number for the outer loops.

Additionally, the overhead introduced by the probes makes it desirable to decide dynamically what to instrument. Instead of measuring every loop, we can choose to instrument only loops that are worthwhile in terms of information gained versus cost. For example, if a loop is very small, we may avoid instrumenting it and instead estimate its cost, because the probe overhead would otherwise dominate the measurement and introduce avoidable bias.

Relationship between BuboL probe and loop frequency

The previous results show that BuboL’s observer effect is most visible in smaller, tighter benchmarks and in nested-loop cases where an outer loop can include the cost of inner loop probes. This raises a more specific question: how does the size¹⁸ of the inner loop affect the amount of probe overhead recorded by the outer loop? To answer this in a controlled way, we focus on the relationship between inner loop iteration count and the additional cycles measured by the outer-loop probe.

To measure the rate at which BuboL’s probe cost relates to the number of inner-loop iterations a benchmark incurs, we implemented an additional benchmark to test this exact situation. This benchmark, LoopStep, is a simple benchmark that contains two loops, an outer loop and an inner loop. The point of this benchmark is to represent a scenario where there are two loops, one nested in the other, with both

¹⁸When we refer to a loop’s “size”, we mean its computational cost; a loop that performs more work is therefore considered larger.

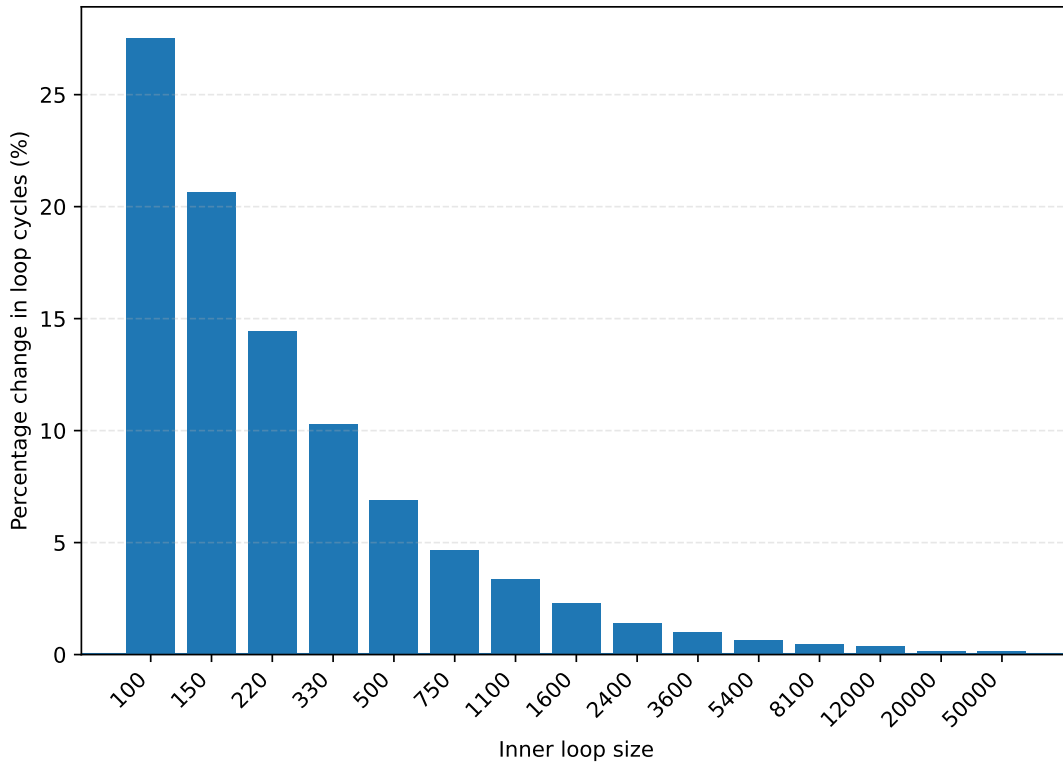


FIGURE 6.7: A measure of BuboL’s effect on the outer loops in our LoopStep benchmark. The percentage change indicates how many additional cycles are recorded in the outer loop when the inner-loop probes are present and recording. The gradual decline as the inner loop size increases indicates that the additional overhead BuboL incurs on the inner loops becomes less significant as the amount of looping increases.

using some CPU time, but with the inner loop accounting for the majority of the time spent.

The benchmark performs a fixed set of floating-point computations as well as simple arithmetic. We store the result of these calculations in a variable that acts as a sink, to stop the JIT compiler from optimising away the computations and the whole loop. We use a smaller workload for the outer loop and a larger workload for the inner loop.

We set the outer loop to run 1,000 times, and we step the inner loop gradually from 100 to 50,000 iterations. We then perform the same experiment as before (Section 6.4.6), recording the cycles spent in the outer loop, first when only the outer loop is instrumented by BuboL, and then again when both the inner and outer loops are instrumented by BuboL. Using the percentage increase in cycles between these two instrumentation modes allows us to determine how much of the time recorded in the outer loop is in fact CPU time spent in the inner loop’s BuboL probe. The results of

this can be seen in [Figure 6.7](#).

The higher levels of percentage increase in outer-loop cycles recorded for smaller inner-loop sizes shown in [Figure 6.7](#) indicate that, for small loops, BuboL’s cost makes up a substantial portion of the cycles recorded in the outer loop. For this experiment, when the inner loop size is 100, we observe a 27.54% increase in cycles recorded when we also instrument the inner loop.

What is clear from [Figure 6.7](#) is that, as the inner loop size increases, the percentage change in outer-loop cycles decreases. This indicates that as the inner loop runs for longer, the probe cost becomes smaller as a fraction of total CPU time, so the inner-loop BuboL probe contributes proportionally less to what the outer probe records. We find that, for this benchmark, once we reach 3,600 inner loop iterations, the increase is only 1.02% cycles.

This relationship between BuboL and the number of inner iterations in the Loop-Step benchmark is a good indication that BuboL’s cost can be effectively hidden by a larger workload. For loops with larger bodies of execution and larger repetition counts, the self-observer effect becomes less dominant. In other words, with larger loop sizes we measure less and less of BuboL’s own incurred overhead, and therefore, we can be more confident that the majority of the CPU cycles recorded are those of the program rather than those introduced by BuboL’s probes. This reduces BuboL’s own observer effect.

To what extent does BuboL change performance behaviour?

To address RQ2, we considered how much BuboL changes performance behaviour, both across the whole program and within nested loops. The results show that BuboL’s effect is benchmark dependent. For most benchmarks, the change in performance behaviour is small. However, for benchmarks with tight or heavily nested loops, the probes can introduce a more noticeable observer effect, either by changing the whole-program distribution of execution time or by causing outer loops to include the cost of inner-loop probes.

For the AWFY benchmarks that are more micro-benchmark-like, such as Sieve and Mandelbrot, our outermost loop probe does measure a significant portion of

CPU time that is caused by BuboL itself, with outer-loop changes of 44.1% and 22.2% respectively (Figure 6.6). However it must be noted that these outer loops contribute only a small percentage of the total run time (3.2% for Sieve and 0.01% for Mandelbrot). For larger benchmarks like NBody and Json, we see a far smaller proportion of time being measured that is due to BuboL's own probes, consistent with their much smaller whole-program distribution changes (for example, NBody has a median change of 0.2%).

We also test, using the nested LoopStep benchmark, how increasing the amount of inner-loop work affects the additional cycles recorded when instrumenting the inner loop. There is a relationship between a loop's cost, its activation count, and the cost of BuboL. In cases where there are loops with a smaller workload and a small activation count, BuboL will incur a penalty, for example we observe a 27.54% increase in outer-loop cycles when the inner loop runs for 100 iterations. But for larger loops with a large body of execution, or loops with a large activation count, we see that the measurement influence of BuboL is significantly reduced, falling to 1.02% once the inner loop reaches 3,600 iterations.

When testing accuracy, this raises the question of how much the outer loops affect the accuracy of the final results. We know that the outer loops will suffer due to the self-observer effect, but for the inner loops this should be much less of an issue, because they have no additional BuboL probes inside them. This leaves the question of whether this influence is infrequent enough to cause a misattribution in the level of accuracy in the cycles, a question we aim to test in RQ3.

Threats to Validity

A threat to the validity of this experiment is accuracy of the sampling profiler used to measure performance behaviour. While we rely on Async to be consistent between runs, the possibility of sampling skid, and missing debugging information in other parts of the program, may influence the results. The individual cost of RDTSC may increase the likelihood of skid due to the amount of CPU time spent executing a single instruction. Likewise, if debugging information is missing immediately before or after a probe, a sample taken there may be misattributed in a way that would

not occur if the probe were not present. These factors may mean that some samples are not fully reliable. While they are unlikely to account for the full performance behaviour change observed, they are known sources of bias and could contribute to the findings.

Future work could further validate these results by measuring BuboL’s self-observer effect in a simulated environment, where the expected time spent in each loop is known before instrumentation is added. Comparing this expected behaviour with the instrumented execution would make it possible to measure more precisely how much additional time is introduced by the probes themselves.

6.4.7 Accuracy of profilers in measuring the proportion of CPU time spent in loops

To address RQ3, i.e., *do instrumentation-based profilers and sampling-based profilers record the proportion of CPU time spent in loops accurately?* We evaluate the accuracy of three profilers, Async, JFR, and BuboL, with respect to measuring the CPU time spent in loops. To precisely evaluate this, we once again make use of deliberate slowdown. As shown in [Chapter 5](#), deliberate slowdown provides a methodology that allows evaluation of profiler accuracy by deliberately slowing down a program and then assessing whether a tool accurately detects the resulting change through its measured proportions of performance behaviour. We use this in combination with VTune results for the time spent in loops within a program. This allows loops to be deliberately slowed down, and for assessment of how close each tool’s reported change in percentage of CPU time is for that loop, by comparing its value to VTune.

Methodology

In this chapter, we are interested in evaluating profiler accuracy at the level of loops. To do this, we build on the whole-program accuracy methodology established in [Chapter 5](#). To do this, we first have to establish what part of a program’s execution is in a loop, and then how to accurately slow down loops. Similar to [Chapter 5](#) we make use of the idea of slowing down the whole program to the target slowdown,

but we also look further into the VTune profiles to see the individual overhead of every machine-code basic block that is in a loop.

We implement an additional debug phase in the Graal pipeline, the `GTLoopDataDebug`.¹⁹ This, as the final LIR phase, dumps information about what the current control-flow graph looks like, and which LIR blocks belong to which loops. Using this information from a separate run, we use our `GTMarkerPhase` (Chapter 5) to map VTune basic blocks back to the corresponding LIR blocks in the Graal compiler. Therefore, once we know which Graal blocks are in a loop, and which loop they belong to, we can determine which VTune blocks are in which loop. This then allows us to derive VTune time for individual loops, giving us a VTune value to compare against what BuboL, Async, and JFR each report for loops.

To establish a baseline for the time spent in each loop, we first execute the whole program using the slowdown technique from Divining. We then repeat the execution with the `GTLoopDataDebug` phase enabled. This phase allows us to identify the loops in VTune’s results and determine the time attributed to each loop before and after slowdown. By comparing these measurements, we calculate the percentage increase in execution time for every loop.

For this evaluation, we also had to account for the way BuboL changes the generated code. As currently implemented, BuboL emits additional `mov` instructions due to register allocation. Because of how VTune works, it can see these additional register-allocation `mov` instructions as whole new blocks, even though they do not contain any jumps. This matters because the slowdown must be placed in the correct block when determining the VTune baseline. Therefore, we had to build a BuboL-aware version of the slowdown file. This slowdown file accounts for any possible additional `mov` blocks and places the slowdown in the correct block as required.

We can then run each tool both with no program slowdown and with program slowdown, and measure the percentage change in each loop according to that tool. The difference between the percentage change the tool reports and the percentage change VTune reports is our measure of accuracy for that tool, for each loop.

¹⁹While this is purely an information printing phase, the GT name comes from the Ground Truth system established in Chapter 5, allowing us to better understand the ground truth.

This approach allows us to compare the degree of accuracy for each tool, whether it is based on actual CPU cycles, as BuboL produces, or the amount of CPU samples, as Async and JFR produce.

This approach is equivalent to the approach used for evaluating of sampling profiler accuracy in [Chapter 5](#). Where we tested sampling profilers by selecting a specific basic block to run 5 seconds slower and then checking whether the profilers correctly attributed the additional time to the selected target block. Here however, for RQ3 it is more straightforward to slow down each block belonging to the target loop by 100%, since loops can span multiple basic blocks. The underlying idea is the same, we introduce a known amount of slowdown and test whether the profiler attributes it correctly. The difference is that we move from an absolute slowdown, 5 seconds for one block, to a relative slowdown, 100% for each block in the loop.

BuboL, Async and JFR’s overall accuracy

With the above methodology, we now present in [Figure 6.8](#) the results of evaluating each tool’s accuracy, using VTune as the baseline. [Figure 6.8](#) shows the overall accuracy results for each tool, aggregated across the full benchmark set, with a per-benchmark breakdown for each tool found in the appendix ([Figure A.5](#), [Figure A.6](#), [Figure A.7](#))

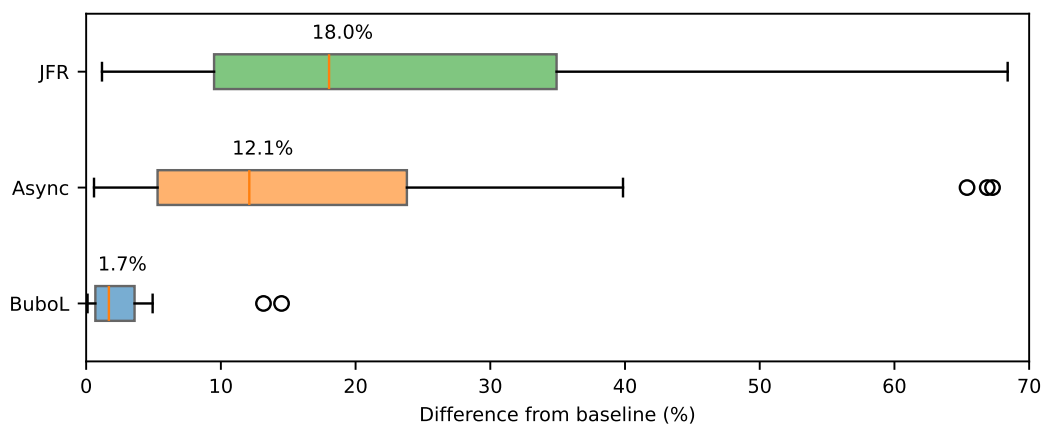


FIGURE 6.8: Accuracy of BuboL, Async, and JFR, comparing their difference in measured loop slowdown compared to the VTune baseline. BuboL shows a median 1.7% (min. 0.10%, max. 14.50%), JFR 18.0% (min. 1.17%, max. 185.75%) and Async 12.1% (min. 0.58%, max. 189.72%)

We expect that, if each tool is accurate, the difference from the baseline should be as small as possible. As [Figure 6.8](#) shows, the most accurate tool is BuboL, with a median difference of 1.7% overall, while Async is 12.1% and JFR has a median difference of 18%. The level of inaccuracy we found with the sampling-based profilers is far larger than we would expect.

Async and JFR's accuracy

The median inaccuracy shown in [Figure 6.8](#) is larger than the inaccuracy we found in [Chapter 5](#), which suggests that measuring smaller and tighter program constructs, such as loops, can lead to greater inaccuracy in JVM sampling. We are surprised that the inaccuracy remained so high, despite ensuring that the debug information of the compilation has loop-specific data as described in [Section 6.4.2](#), which includes setting instructions with no debugging information within the loop.

In providing loop-level debug information there is a consistent region of debug information associated with a single loop. Additionally every instruction in the loop body is given debug information, which is not always the case with method-level debugging. Therefore, we reduce the likelihood that a sampler misattributes samples due to missing debug information, and we reduce the need for it to fall back on heuristics, such as guessing using the next available debug location. Therefore, these factors are unlikely to be the reason why the samplers appear significantly more inaccurate than BuboL.

For Async and JFR, the NBody benchmark is the only case which sees a good level of accuracy, with a median change from the baseline of 3.8% for Async and 1.2% for JFR. However, it must be noted that BuboL achieves a far *smaller change* of 0.1%.

The remaining question is whether this inaccuracy comes from samples being missed, or from samples being attributed to the wrong loops.

Oversampling and Undersampling

Given the level of inaccuracy observed for Async and JFR in [Figure 6.8](#), we investigate further where in the benchmarks samples are being over-attributed or under-attributed.

To establish whether some loops are being over-attributed or under-attributed, we first check whether the measured run-time overhead aligns with the increase in collected samples. For example, if the program is slowed down by approximately 102%, then the total number of collected samples should also increase by approximately 102%. This needs to be established to verify that the observed levels of inaccuracy are not caused by samples being lost, stopped, or not collected for some unknown reason.

Across the benchmark set we evaluated, the measured run-time overhead did closely align with the number of samples collected. We therefore know that, if a profiler reports that one loop increased by more than expected, that loop is being oversampled, and some other region of the program must be undersampled.

To examine this effect, we look at Async profile for the Sieve benchmark. We compare the profile without slowdown against the profile with slowdown, and use the VTune-measured slowdown to calculate the expected increase in samples. For each method or loop, we start with the number of samples collected before slowdown and then predict how many samples should be collected after slowdown based on the VTune-measured increase in execution time for each block belonging to either a loop or method. This approach is similar to our accuracy evaluation in [Figure 6.8](#), but here we examine the change in raw sample counts rather than the change in percentage of run time spent in loops.²⁰

[Figure 6.9](#) shows the distribution of samples for the Sieve benchmark using Async-profiler. We plot the number of samples that are above and below the expected increase. The slowed-down run contains a total of approximately 27 million samples.

Across the profile shown in [Figure 6.9](#), the total oversampled count is 2,720,929 samples, while the total undersampled count is 2,692,424 samples. The difference is 28,505 samples, which is within the expected range of noise and estimation error. As mentioned earlier, the measured overhead and the sample count closely align, and this small difference in samples strongly suggests that the profiler is not failing

²⁰In addition, key difference between this evaluation and the one shown in [Figure 6.8](#) is that here we do not include the percentage of run time consumed by each loop. For example, Loop 3 consumes 50% of the run time, whereas Loop 0 consumes only 1.15%. This is an important factor when determining accuracy. An increase of approximately two million samples has a different significance depending on whether the loop already accounts for 13.5 million samples or only 0.4 million samples.

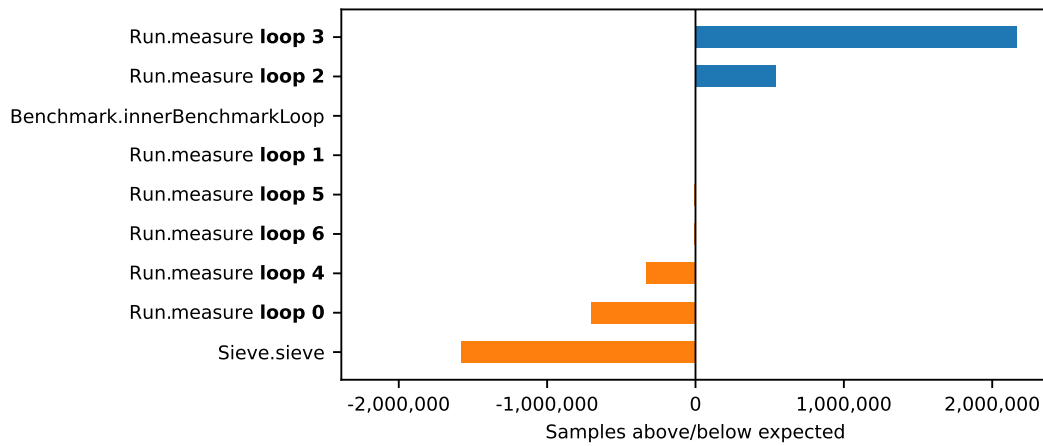


FIGURE 6.9: Distribution of samples for the Sieve benchmark according to Async. The plot shows which methods and loops are oversampled or undersampled relative to their expected sample counts.

to collect samples overall. Therefore, the inaccuracy appears to come from where samples are attributed, rather than from samples being missed.

Figure 6.9 shows that loops **2** and **3** in `Run.measure` are substantially oversampled. Loop **3** is oversampled by approximately 2.1 million samples, and loop **2** is oversampled by approximately 0.5 million samples.

At the other end of the distribution, loops **0** and **4** in `Run.measure` are undersampled. Loop **0** is undersampled by approximately 0.7 million samples, and loop **4** by approximately 0.3 million samples. Most importantly, the method `Sieve.sieve` is undersampled by approximately 1.5 million samples. This represents execution in the Sieve benchmark outside the loops identified by Async.

Below, Figure 6.10 shows the loop nesting structure for `Run.measure`. The most nested loops, loops **2** and **3**, are oversampled, while the outer loop **0** and loop **4** are undersampled. Loop **4** is more significant in terms of execution time, whereas loops 1, 5, and 6 contribute very little to total execution time. These small loops are likely the result of loop peeling and have minimal impact on the overall profile.

This pattern suggests that samples are being shifted from outer or surrounding regions into the hotter inner loops. A likely explanation could be sampling skid, a sample may be triggered while execution is in an outer loop or adjacent region, but by the time execution is halted and the program counter is recorded, execution has

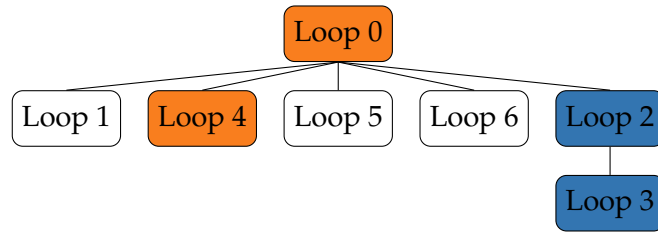


FIGURE 6.10: Loop nesting graph for `Run.measure`. Loops 0 and 4 are highlighted in orange, and loops 2 and 3 in blue, to match Figure 6.9, where orange denotes substantially undersampled loops and blue denotes substantially oversampled loops.

advanced into one of the hotter inner loops.²¹ As a result, the sample is attributed to loops 2 or 3 instead of the region where the sample was originally triggered. Although loop 2 is not nested inside loop 4, loop 4 always leads to loop 2 along the execution path, which makes this kind of attribution shift plausible.

Beyond the loops themselves, the substantial undersampling of `Sieve.sieve` suggests that misattribution of debugging information may also contribute to the result. We assign loop-specific debugging information to the instructions inside loops, but leave code outside loops unchanged, using the debugging information normally generated by Graal. One possible explanation is that some instructions outside the loops still lack debugging information, causing samples from those regions to be attributed to the next available debugging information, which may belong to the loops in question. However, if this were the only cause, we might expect the samples to be attributed primarily to the first available loop, such as loop 0. Nevertheless, the undersampling of non-loop execution suggests that attribution outside loops remains unreliable for this benchmark at least.

Safepoint Bias

As discussed in Section 2.2.2, modern JVM profilers such as Async and JFR can collect stack traces without waiting for a safepoint, but resolving the top frame may still depend on nearby safepoint debug information. Therefore, even when stack-trace collection itself is not restricted to safepoints, attribution can remain biased towards locations where debug information is available.

²¹Error Margin: Skidding + inlining - <https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html>

If the issue of why some loops are oversampled is related to safepoint bias, then we would expect the effect to appear most strongly in the inner loops. A loop back edge is the control-flow edge that returns execution from the end of one loop iteration back to the loop header to begin the next iteration. Loop back edges are likely locations for safepoint, since they are reached repeatedly during loop execution.²² Therefore, if the sampler is biased towards safepoints, the back edges of the hottest inner loops are likely places to be sampled.

Loop	Number of LIR Blocks	Safepoints	Sampling deviation (%)
0	10	1	−2.58
1	2	0	−0.001
2	8	3	2.00
3	3	1	7.97
4	2	0	−1.21
5	2	0	−0.02

TABLE 6.2: Run.measure’s loops, listing the number of LIR blocks and the number of safepoints contained within those blocks, and the sampling deviation as the percentage difference from the expected sample count. Negative values indicate undersampling, while positive values indicate oversampling.

Table 6.2 shows the number of LIR blocks in each loop, together with the number of safepoint operations found inside those blocks, and the sampling deviation. Sampling deviation measures how far the observed sample count differs from the expected sample count, expressed as a percentage of the expected count. Positive values indicate oversampling, while negative values indicate undersampling. Loops 2 and 3 were oversampled, and these loops also contain the highest number of safepoints.

Figure 6.11 shows a section of the CFG for the Sieve benchmark. The full CFG can be found in the appendix in Figure A.8. We can see in Figure 6.11 the distribution of safepoints across these loops in the Run.measure compilation unit.

Loop 2 contains three safepoints, in blocks 18, 25, and 28, all of which lie on loop back-edge paths. Loop 3, the most oversampled loop, contains a safepoint in block 24, which is part of the main loop sequence and on the loop back-edge. This means that each iteration of loop 3 encounters a safepoint. Loop 0, which was undersampled, contains a safepoint in block 30, corresponding to its back edge.

²²See Section 12.6, “Threading and Yieldpoints”, <https://www.jikesrvm.org/UserGuide/AdaptiveOptimizationSystem/index.html>.

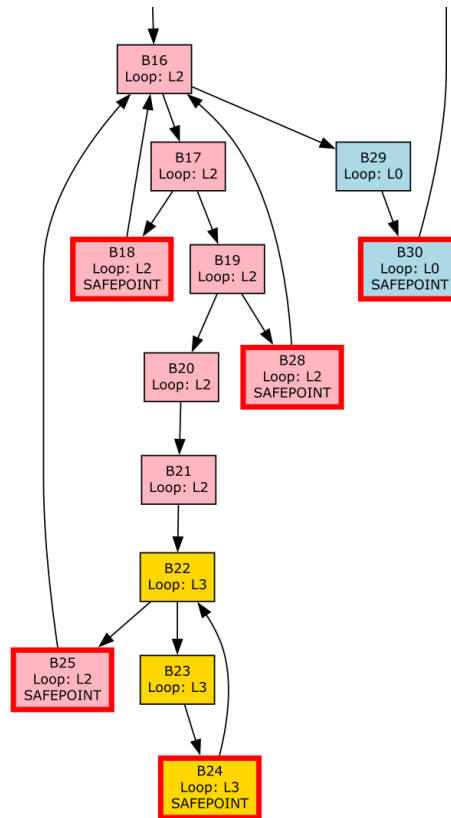


FIGURE 6.11: Control-flow graph of the Graal LIR blocks for the hot loops 2 and 3 in the Sieve benchmark, with safepoint-containing blocks highlighted.

This suggests that sampling bias may align with the concentration of hot back-edge paths, where the safepoints occur. Since these paths are executed frequently, any safepoint-related attribution bias would be concentrated in loops 2 and 3.

Loops 1 and 4 contain no safepoints in their own loop bodies (a full CFG showing these loops is provided in the appendix in Figure A.8). Their nearest safepoints are therefore located either in loops 2 or 3, or on the back edge of loop 0. This may explain why loop 4 was undersampled.

BuboL's accuracy

Overall, BuboL's accuracy is far higher than the sampling-based profilers. We can see from Figure 6.8 that the median difference for each benchmark is in another factor compared to the samplers, and we also find a far tighter range between the minimum and maximum values.

Figure 6.8 plots the absolute change from the baseline. With BuboL, the raw percentage change in cycles for loops was sometimes slightly under and sometimes

slightly over the VTune baseline. In contrast, with the sampling profilers, in almost all cases they were above the baseline and rarely, if ever, under the VTune baseline.²³

The fact that BuboL falls on both sides of the baseline, both under and over, suggests that at this point we are seeing limits of measurement accuracy. This indicates that we are achieving accuracy as close as possible in a real benchmark setting, which gives us good confidence in our results being as accurate as possible in a real world benchmark setting, which is subject to noise, CPU and compiler nondeterminisms. If, for example, we were in a simulation setting where we had tighter control over CPU frequency and execution order, and could limit nondeterminism, we would expect to see more stable results with less variance.

Some note must be made of the two outliers we found in [Figure A.5](#) (in the appendix) : one in LoopBenchmarks and one in Sieve. In the LoopBenchmarks case, one loop BuboL identified, had a difference from the baseline of 13.50%. In the Sieve benchmark, for one loop BuboL had a difference of 13.16%. Looking into the details of these loops at the machine code level, we find that both contain a large amount of vector instructions performing arithmetic, and we found that they had a large amount of variance in CPU time consumption from one run to another. The most likely cause for BuboL's inaccuracy here is that these are tight, short-running inner loops, with most of their CPU time spent in the vector unit, which can be more variable than what the BuboL probe accounts for.

What is also of interest is that, in RQ2, we found higher overhead in Sieve and Mandelbrot, as well as a concern of high self-measurement of BuboL's own probes in the outer loops. However, as one can see in [Figure A.5](#) in appendix, the difference from the baseline for the Mandelbrot and Sieve benchmarks is not as high. While Sieve does have BuboL's highest median difference of 4.2% and the widest range, which likely indicates some effect of the self-observer effect seen in RQ2 in the outer loops, most of the time is spent in the inner loops, which do not suffer from such self overhead. The accuracy, and how close it remains to the VTune time, gives us great confidence in the level of accuracy of BuboL, in comparison to Async and JFR.

²³For the loops that contribute the majority of run time, the samplers report a change far larger than the expected value from the baseline. As shown previously in [Figure 6.9](#), this comes from oversampling the hot loops and undersampling less significant outer loops or methods.

RQ3: Do late late-instrumentation-based profilers and sampling-based profilers record the proportion of CPU time spent in loops accurately?

To address RQ3, we find overall that the sampling-based profilers are not consistently accurate when measuring CPU time spent in loops. Using our methodology (section 6.4.7) i.e., deliberate slowdown of loops, measuring loop time with VTune, and using that time as a baseline, we find an overall median difference of 18% for JFR and 12.1% for Async. This suggests that these profilers struggle to accurately determine the CPU time spent in loops.

In contrast, we find that BuboL, the late-late-instrumentation-based profiler, achieves much higher accuracy, with a median percentage difference of 1.7%. This suggests that instrumentation, when compared to sampling-based profilers, can yield more accurate results. Instrumentation does not suffer from biases such as *safepoint bias* or *instruction skid*, and while its run-time overhead is higher, for this set of benchmarks that overhead comes with a far greater degree of accuracy when measuring the proportion of CPU time spent in loops.

6.5 Related Work

Loop-centric profiling Loop-level profiling has previously been explored as an alternative to method-centric analysis. As mentioned earlier, Moseley et al. 2007 argue that traditional profilers obscure optimisation opportunities by focusing on procedures rather than loops. Similarly, Jain et al. 2017 present an LLVM-based loop profiler, showing that across SPEC²⁴ benchmarks, a small fraction of loops accounts for the majority of loops execution time. Their work reinforces the observation that loops are a meaningful unit of performance analysis and optimisation.

Our work differs in two key ways. First, BuboL operates inside a just-in-time compiler, rather than compiler pass in LLVM. Second, we explicitly evaluate loop-level accuracy using deliberate machine-code slowdown, allowing quantitative comparison between instrumentation and sampling accuracy in profiling loops. Jain et al. 2017; Moseley et al. 2007 loop-centric work focused primarily on insight and optimisation potential rather than measurement accuracy.

²⁴<https://www.spec.org/cpu2006/>

Sampling skid Yi et al. 2020 evaluates Intel’s Precise Event-Based Sampling (PEBS) (See Section 2.8) and demonstrates that PEBS is not truly precise. They identify two key hardware effects that introduce bias: *skid* and the *shadow effect*. Skid refers to the delay between a counter overflow and the recording of the sampled instruction, meaning that the sampled instruction may not be the one that actually triggered the event. Moreover, they show that the shadow effect, the delay between a counter overflow and the arming of PEBS hardware, causes subsequent nearby instructions to be preferentially sampled. Their experiments reveal a severe bias, where samples consistently cluster on specific instructions, even when all candidate instructions should have equal probability of being sampled. This bias depends on instruction type and latency, with longer-latency operations, such as cache misses, disproportionately attracting samples.

Given that JVM-level profilers such as Async-Profiler rely on hardware performance monitoring unit (PMU) events exposed via the OS through Perf, these findings suggest that skid and shadow-related biases may contribute to the poor loop-level accuracy we observe. In particular, tight loops with closely spaced back-edges or low-latency instructions may suffer from systematic misattribution due to hardware-level sampling imprecision. As JFR does not use PMU events, this may indicate that a higher-level sampling bias may still exist, where constraints such as safepoints can bias both where sampling occurs and where debug information can be collected. As a result, instructions are not necessarily sampled in proportion to the time spent executing them, which is a key requirement for a sampling profiler to be accurate.

Strategies for reducing overhead and observer effect Work by He et al. 2024’s in profile-guided optimisation suggests that it is possible to obtain higher-quality profiling information without resorting to full instrumentation. Their implementation of CSSPGO²⁵ combines sampling with pseudo-instrumentation and context-sensitive techniques to approximate the quality of instrumentation-level profiles while maintaining low overhead. Rather than relying on flat, context-insensitive sample counts, it correlates sampled events with control-flow and calling context to improve the reliability of hot-path identification.

²⁵Context-Sensitive Sampling-Based Profile-Guided Optimization

Similarly, Vukasovic et al. 2023 show that partially context-sensitive profiling, which retains a limited amount of call-context information, can significantly improve the accuracy of hotness decisions without incurring the cost of full context sensitivity. These approaches demonstrate that it is possible to more precisely determine which regions of a program truly dominate execution, while still operating within practical overhead limits. This insight can be applied directly to BuboL, instead of instrumenting every loop, it could first perform a lightweight, context-aware sampling pass to rank loops according to their estimated contribution to total execution time. BuboL could then instrument only the most relevant loops, thereby reducing overhead by avoiding instrumentation of small or infrequently executed loops where probe cost would dominate the measurement. Such a strategy would allow BuboL to retain its high measurement accuracy for important loops, while reducing its own overhead and observer effect.

Qasem et al. 2006 show how to speed up whole programs by empirically finding fast loop optimisation settings, instead of relying on simplified static compiler models or manual tuning. They build a prototype iterative system that rewrites loops (tiling and unroll, including unroll-and-jam), compiles the new version, runs it, measures its performance, and then tries different optimisation choices guided by a direct-search strategy. They show this “try, measure, and improve” approach can deliver noticeable speedups while exploring only a small fraction of the variants that an exhaustive search would require. The framework uses HPCToolkit²⁶ to sample hardware-counter for performance metrics and to attribute aggregate costs to loops. In the current prototype the search itself is primarily guided by whole-program execution time, with loop-level feedback positioned as a key direction for future refinement. BuboL could be applied to this kind of iterative optimisation system by replacing hardware-counter sampling with direct loop-level instrumentation. This could yield more accurate measurements, as well as loop-level feedback that could lead to better optimisation decisions.

²⁶<https://hpctoolkit.org/>

6.6 Conclusion

In this chapter, we explored how sampling profilers show fundamental limits in accuracy when attributing CPU time to loops. We showed how sampling-based profilers such as Async and JFR exhibit large inaccuracies when measuring loop-level CPU time, with median deviations of 18% and 12.1%. These inaccuracies become more pronounced for small, tight, and frequently executed loops. This is reflected in our results ([Figure 6.8](#)), where the worst sampling inaccuracy occurs in Mandelbrot and Sieve, both of which contain small, tight loops. The most likely causes include instruction skid and safepoint bias. Instruction skid provides one possible explanation for misattribution, while safepoint bias is examined in more depth in [Section 6.4.7](#).

We find that instrumentation can be accurate if placed late enough in the compiler. Traditional instrumentation-based profilers often fail due to excessive overhead and disruption of compiler optimisations. In [Section 6.4.3](#), we discussed how placing BuboL's probes late in the compiler pipeline avoids perturbing optimisations such as inlining and loop transformations. Late-late-stage instrumentation therefore enables accurate cycle measurement while preserving realistic program behaviour.

BuboL demonstrates that loop-level instrumentation can be both accurate and practical. BuboL achieves high accuracy when measuring time spent in loops, with a median deviation of approximately 1.7% from VTune. Its overhead remains practical for realistic workloads, with a median CPU-time overhead of 18.6%. We find that this overhead is proportional to loop activation frequency. Tight and frequently executed nested loops increase BuboL's runtime overhead the most, but higher activation counts reduce the probe cost as a proportion of total execution time.

BuboL does incur an observer effect, but it is concentrated in worst-case scenarios, e.g., small, tight loops where probe cost makes up a meaningful proportion of the total measured time. However, this makes the overhead predictable, and it suggests that probe placement strategies can further reduce bias and overhead.

Overall, instrumentation-based profiling on the JVM is not inherently impractical. When designed carefully, with late placement, it can outperform sampling in loop-level accuracy while remaining usable. This chapter shows that accurate, low-perturbation profiling is achievable and provides a viable direction for future

profiler design.

Chapter 7

Discussion of the Challenges Modern Profilers Face

This chapter focuses on the difficulty of attributing execution time to methods in modern JVM profilers, an ongoing issue identified throughout this thesis. [Chapters 3, 5 and 6](#) showed that sampling-based profilers can be imprecise, can disagree with each other, and can misattribute execution time. [Chapter 4](#) showed that traditional instrumentation-based profilers can introduce high overhead and perturb program behaviour.

Building on these results, this chapter focuses on method-level attribution as a remaining challenge for modern JVM profilers. We argue that this attribution becomes difficult when execution repeatedly crosses method boundaries and must be mapped back to source code through debug information that may be incomplete or imprecise. Inlining, aggressive optimisation, and changing method boundaries therefore make it difficult to determine where execution time should be attributed.

In [Chapter 6](#), we explored loop-level profiling as an alternative to method-level attribution, showing that loops can be a useful and more concrete target for profiling. However, methods remain important because most existing profilers still use methods as their primary reporting unit. Additionally, in Java programs, methods are a core abstraction and one of the main ways program behaviour is organised. For this reason, this chapter builds on that discussion by examining method-level attribution as a remaining challenge, while also considering how future profilers could report alternative regions of interest, such as loops, inlined code, or transformed source-code regions.

To explore this direction, the chapter outlines a possible future profiler design based on compiler-guided, late-stage instrumentation. This approach combines instrumentation with sampling and uses richer compiler information to report how optimised code relates back to the source program.

This chapter contributes to the thesis by identifying method-level attribution as a remaining source of profiler inaccuracy. It also outlines a future profiler design that would retain method-level reporting, while using compiler information to report alternative regions of interest, such as loops, inlined code, and transformed source-code regions. This could allow future profilers to provide more accurate and more useful reports.

7.1 Method Boundaries

A traditional profiler reports which methods account for the most execution time.¹ To report this, a profiler must account for all time spent in each method, which requires identifying when execution moves from one method to another, a *method boundary*.

A *method boundary* is a point in the resulting compiled program, whether that be in the final machine code or IR, where execution transitions from one method to another. This is most commonly identified by the debugging information changing from one method to another. This can be as straightforward as a method call, or it can arise after compiler optimisations such as inlining, where execution moves between instructions that are associated with different methods.

7.1.1 How do profilers identify what method they are in during execution?

Profilers commonly report execution time at the method level. For sampling-based profilers, this is usually done by recording the current program counter and using the available metadata to map that program counter back to a method. This is a low-overhead approach, but it depends on the quality and availability of debug information. It is also vulnerable to sampling skid, which can cause samples to be attributed to a nearby instruction, and therefore potentially to the wrong method.

¹In [Chapter 6](#), we focused on loops. The explanation of why we focus on loops, and why loops make good targets for profiling, can be seen in [Section 6.2.2](#).

For instrumentation-based profilers, the problem is different. If probes are inserted before optimisation, for example during bytecode loading, the profiler can record method entry and exit directly. However, after inlining and other compiler optimisations, the recorded execution may no longer correspond cleanly to the original method structure.²

Alternatively, an instrumentation-based profiler could attempt to account for the method boundaries that exist after optimisation, including boundaries created by inlined code. This would require instrumenting many more points in the compiled program, or at least estimating how much execution time belongs to each method within a mixed region of code. This is costly, and the estimate may still be inaccurate. Therefore, from an instrumentation point of view, method boundaries create a trade-off: ignoring them can lead to inaccurate attribution, but measuring them directly can substantially increase overhead and observer effect.

7.1.2 Issues with method boundaries

For a profiler to accurately report where execution time is spent at the method level, it must account for the number and complexity of method boundaries in the compiled program. The problem becomes harder when method boundaries are frequent and when optimised code contains instructions from many methods mixed together. In such cases, there may no longer be clear regions of execution that belong to one method before moving to another.

This creates problems for both sampling and instrumentation. For sampling-based profilers, dense method boundaries increase the reliance on accurate debug information and make the probabilistic nature of sampling more problematic, since many method transitions may occur between samples. For instrumentation-based profilers, additional method boundaries either increase inaccuracy if they are ignored, or increase overhead and observer effect if each boundary must be measured.

We discussed in the background ([Section 2.6](#)) the issue of profilers having to deal with inlining. We found that, in the compiled IR, the program can be extremely mixed.

²We suspect tools like JProfiler and YourKit do this, because they instrument methods before optimisation and then make no attempt to re-instrument or account for cases where the code is inlined.

Many nodes from many different methods can be heavily interleaved with each other, as shown in [Figure 7.1](#).

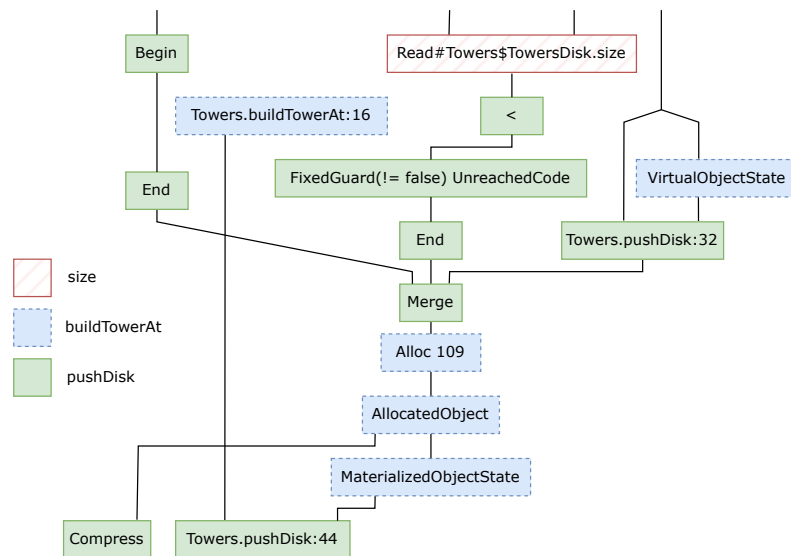


FIGURE 7.1: A Graal HIR graph showing that nodes from inlined methods can end up being intermixed in a compilation unit

But [Figure 7.1](#) is just one section of code at the HIR level. [Figure 7.2](#) demonstrates the real extent of the problem, specifically how common and how frequent method boundaries are in the Are We Fast Yet benchmarks. The number of method boundary sites is shown in the lower bar chart, and the values range from only a few, with Mandelbrot having none,³ to much larger benchmarks like Json having 1,432 method boundaries. These method boundaries do not represent unique methods, but rather sites in the generated code where the attribution of machine code changes from one source method to another.

The top bar chart shows the extent of activation for each boundary, that is, how many times execution crosses a boundary. This can be thought of as the number of times we change method during execution.

To collect these statistics, we used the debug information provided by Graal for each LIR instruction. This allowed us to statically count the number of method-boundary sites in the generated code. To measure how often these boundaries were crossed at run time, we inserted additional machine-code instrumentation at each

³Mandelbrot's source code consists of one method

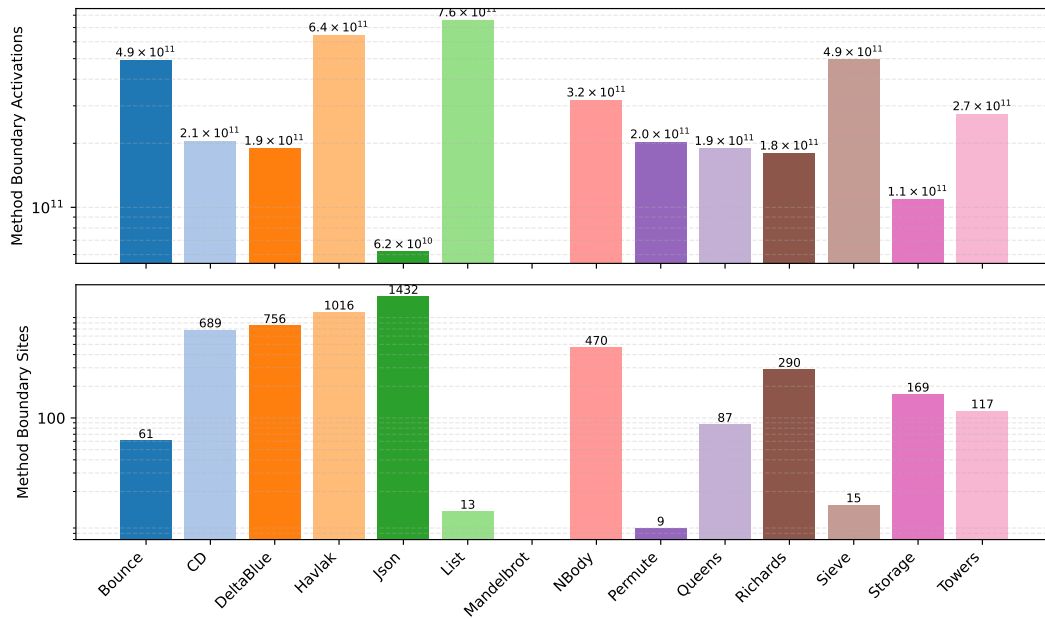


FIGURE 7.2: Number of method-boundary sites and boundary activations per benchmark. The lower bar chart shows the number of static method-boundary sites, while the upper bar chart shows how many times those boundaries are crossed during execution.

boundary. Each probe increments a counter in memory whenever execution reaches that boundary.

Take, for example, DeltaBlue, which is one of the larger benchmarks. We find that, in the resulting machine code, there are 756 points at which execution changes from one method to another. We find that during execution the program changes method 1.9×10^{11} , that is, 190 billion times, in an execution that lasts about 128 seconds.

If we divide that number of method changes by 128 seconds, and then convert it to a 10 millisecond interval (the default sampling frequency of Async), we find that DeltaBlue changes method roughly 14.8 million times every 10 milliseconds. So, between each sample, we change method about 14.8 million times. If we use the lowest sampling frequency for Async,⁴ 10 microseconds, we find that for DeltaBlue 14,800 method changes occur between samples. This is still a large number, adding concern about the amount of unaccounted execution in each method that can occur between samples. However, 10 microseconds is a very high sampling rate. Across the 14 Are We Fast Yet benchmarks, we find that at a sampling rate of 10 μ s the median overhead is 228.7%, which for many use cases is an impractical overhead.

⁴10 microseconds is the lowest sampling frequency we can use on our benchmark machine

Now, in a simpler benchmark, this may just mean switching between the same small set of methods. If you look at [Figure 7.2](#), List and Permute have far fewer method boundary sites, but an extremely high method activation count, which indicates that most of the switching is back and forth between the same small set of methods.

Nonetheless, many of these method changes are unaccounted for by sampling profilers, and we have to rely heavily on the probabilistic nature of sampling to hope that the resulting distribution of samples is representative enough to give a reasonably accurate picture of where time is spent.

If we tried to track all of these method changes with instrumentation, we would likely need to instrument each boundary, meaning that a probe would execute every time execution crossed from one method to another. We know this is infeasible in terms of overhead and observer effect. For the collection of the statistics in [Figure 7.2](#), we used a simple counter that adds 1 to a location in memory for each method change. We find that this simple counting approach added a median overhead of 216.4% across the 14 benchmarks.

These statistics demonstrate how frequently the attribution of machine code changes from one source method to another. Tracking these changes requires measurement at a high level of detail.

7.1.3 A core issue of profiler inaccuracy

We believe that profilers choosing to attribute time at the method level is a core reason why profilers can be so inaccurate. The extent to which inlining and compiler optimisations change and mix method bodies was unlikely to have been foreseen when early profiler tools were used to attribute time to methods. Knowing which method is the hottest sounds simple in principle, and for a very partitioned and well-separated program it is a good first step for guiding manual optimisation. However, the reality of modern compilation means that accurately attributing every line of execution to the correct method is no longer a simple task. The quality of debugging information plays a major part; whether it is accurate in the first place, or even exists at all, has to be considered. There are also cases such as loop fusion or common subexpression

elimination, where work from different source locations may be merged or represented by a single optimised region, causing the resulting cost to be attributed to only one method.

We also see a fundamental issue with how these tools collect this information. Sampling, which is vulnerable to instruction skid and misattribution, also has the problem that, due to the frequency of method boundaries, it can miss large amounts of program execution spent in different methods, simply because of limitations on how often it can sample. This issue of sampling frequency being too coarse seems to be a fundamental issue with sampling. Even if future CPUs allow faster sampling, this does not change the relationship sampling has with program speed. If the CPU is faster then the program is also likely running faster, which means sampling may still miss the same number of method boundaries and the method execution that occurs between samples.

Instrumentation is in a far worse position. It is not straightforward to track and record all the time spent in these methods. If one tries to probe method boundaries directly, it is likely to end up perturbing performance behaviour due to the overhead cost. If we have any hope of accurately recording the time spent in methods, it seems we either need to think of different strategies for sampling, or different strategies for instrumentation, to account for the time spent in a method.

7.1.4 A proposed strategy for instrumentation to more accurately record time spent in methods

With the goal of enabling instrumentation-based profilers to more accurately record time spent in methods without instrumenting every method boundary, we outline a possible future profiling strategy. We cannot yet conceive of a practical way to directly record the individual time spent in each method, given the large number of method boundaries and their high activation rate. We therefore think the next best option, in terms of accuracy while maintaining low observer effect and practical overhead, is a hybrid strategy that estimates the time spent in each method.

The key idea is to define a *measurement window*: a region of execution between two instrumentation probes. The probes record the absolute time elapsed within the

window, while compiler metadata and hardware event samples provide additional information about what happened inside that region. During post-processing, this information can be used to estimate how the measured time should be divided across methods. This gives the profiler an exact measured total for a region, rather than only a ratio of collected samples, while still avoiding the cost of instrumenting every method boundary.

The proposed profiler has the following components, which we describe in more detail below:

1. **Measurement windows:** late-stage instrumentation probes define the start and end of a region of execution.
2. **Exact timing:** RDTSC records the elapsed cycles for each measurement window.
3. **Compiler metadata:** information about methods, inlining, loops, and instructions is collected during compilation.
4. **Sampling within windows:** hardware event sampling collects program-counter locations and branch events, providing information about where activity occurs inside each measured window.
5. **Method-level estimation:** probe timings, samples, and compiler metadata are combined to estimate how time should be attributed to methods.
6. **Confidence reporting:** dense method boundaries, or disagreement between metadata and samples, can be used to flag low-confidence attribution.
7. **PEBS refinement:** Precise Event-Based Sampling provides a more accurate form of hardware event sampling within each window, where supported.

Measurement windows For a given measurement window, the area we measure could be a section of code, a loop, or a whole compilation unit. The important point is that the profiler does not need to instrument every method boundary inside this region. Instead, it places probes at the start and end of the window and treats the code between those probes as the unit of direct measurement.

We propose that these probes are placed late in the compiler pipeline, as with BuboL. This avoids interference with earlier compiler decisions and gives access to

code that is close to the final resulting machine code. As a result, the measurement window is defined at a point where the compiler has already performed many of the optimisations that make method-level attribution difficult.

Exact timing Instrumentation can record the absolute time elapsed between two points, rather than only a ratio of collected samples. This is useful because it gives an exact total for the measured region. Even if the time must later be divided between individual methods using an estimate, the total amount of time being divided is still a real measurement.

The probes for each measurement window would collect the start and end time, most likely using RDTSC. By contrast, pure hardware sampling, like VTune, estimates where time is spent rather than directly measuring elapsed time or cycles between two points. It records samples after a given number of events, such as cycles or retired instructions, and infers time from where those samples occur. RDTSC, however, directly measures elapsed cycles between a start and end probe.

Compiler metadata During compilation, the profiler can also collect detailed compiler information for each measurement window. This includes debug metadata, such as the information samplers use to map the program counter to source or method information. It can also include richer information, such as what types of instructions are inside the measurement window, properties of specific IR instructions and loop information where relevant.

This information is useful because it records how the compiler understands the code at instrumentation time. For example, the profiler could record compiler-derived method and inlining information for the instructions within each measurement window. This gives the post-processing stage a richer basis for estimating which methods are represented inside the window, even when the final optimised code no longer has clear method boundaries.

Sampling within windows To complement the instrumentation, we also propose using hardware event sampling. Using hardware event sampling, we would collect program-counter locations and branch events, as well as trigger stack unwinding

to collect call stacks. Hardware sampling would be conducted across the whole program, with the instrumentation probes used as markers during post-processing. The probe points let us determine exactly how much time elapsed within each measurement window, while the hardware event samples observed within that same window provide information about where activity occurred. We illustrate this idea in [Figure 7.3](#).

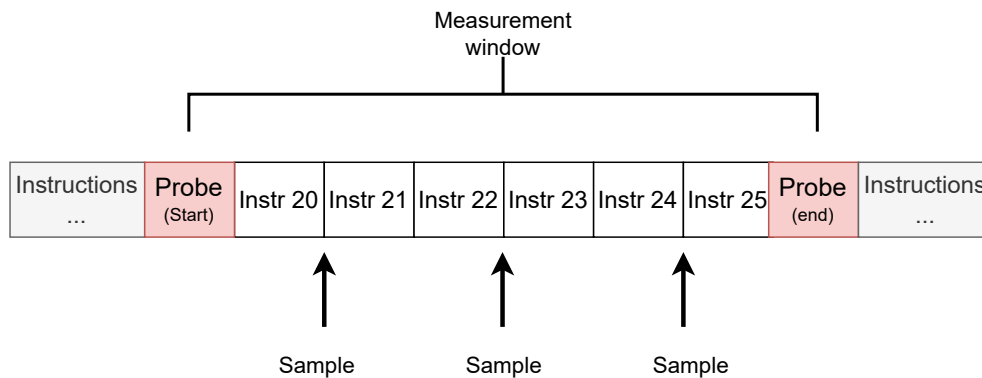


FIGURE 7.3: Concept for a hybrid profiler that instruments a given measurement window while sampling.

This allows the profiler to estimate the ratio of activity per method within the measured window. In terms of overhead, hardware event sampling already has relatively low overhead. Intel reports around 2% overhead at a 1 ms sampling interval,⁵ so the additional overhead from sampling should be minimal compared to the instrumentation’s overhead.

Method-level estimation During post-processing, the profiler combines probe data, hardware event samples, and compiler metadata. The probes provide the exact elapsed cycles for the measurement window. The hardware samples provide a distribution of activity within that window. The compiler metadata then provides the mapping from sampled instruction addresses back to source, method, loop, or instruction-level information.

One way this approach could help address the method-boundary issue is by using the compiler-derived method and inlining information collected for the instructions within each measurement window. Hardware sampling would not observe method

⁵<https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/hw-event-based-sampling-collection.html>

boundaries directly; it would only provide sampled instruction addresses. During post-processing, these sampled addresses could be mapped back to the compiler metadata collected for the window, allowing the measured time for the window to be estimated at the method level.

This approach also allows for more focused reporting. Existing profilers such as VTune can already map from source code to expensive native instructions. The additional opportunity here is that a compiler-guided profiler could also report why that source code became expensive in the optimised program. For example, if a measurement window contains costly memory operations, the compiler metadata could identify the source-level construct and the LIR operations that produced those memory accesses. This would allow the profiler to report not only that a source line or region is expensive, but also whether that cost is linked to a specific compiler decision, such as loop transformation or code motion.

Confidence reporting The compiler metadata can also be used to identify windows where method-level attribution is likely to be unreliable. For example, if the compiler metadata shows that a window contains many method transitions, but the samples collected within that window report little or no method variation, then the timing for the window may still be accurate, but the method-level breakdown is likely inaccurate. In this case, the profiler could report the total time for the window while flagging the method attribution as low confidence.

This confidence signal could also be useful for evaluating sampling-based profilers such as async or JFR. By identifying measurement windows where the compiler metadata suggests dense method transitions but the sampled attribution does not reflect that structure, the tool could highlight regions where sampling-based method attribution is likely to be inaccurate. In other words, boundary density becomes a predictor of where sampling is likely to misattribute time, and the tool can highlight those regions as likely sources of high inaccuracy.

PEBS refinement A further refinement to hardware event-based sampling is Precise Event-Based Sampling (PEBS) (See [Section 2.8](#)). PEBS does not change the fundamental nature of sampling, in that it still estimates time statistically and does not provide

direct cycle measurements or precise region boundaries. However, when combined with our probe-based measurement windows, PEBS can improve the accuracy of the sampled instruction locations within each window. In this configuration, the probes continue to define exact start and end points and measure elapsed cycles via RDTSC, while PEBS samples collected during the window provide a more precise mapping from observed hardware events to individual instructions or source locations. The main caveats with PEBS are that it is not available on all hardware, and that its higher accuracy can come with greater overhead, whereas conventional hardware event sampling is more widely available and typically cheaper in terms of overhead (Yi et al. 2020).

Summary of the proposed strategy. To conclude, we propose an approach that allows instrumentation to more accurately record the time spent in a method without instrumenting every method boundary. We propose the use of a hybrid strategy that places RDTSC probes late in the compiler pipeline, gathers compiler metadata during compilation, and combines this with hardware event sampling.

This gives us the ability, for a given measurement window, to record the exact number of elapsed cycles, while also using hardware event sampling and compiler metadata to estimate the ratio of time spent in individual methods. By collecting method-boundary information at compile time, we can estimate how frequently execution is likely to change method within a given window. From this, we can derive a confidence score for method-level attribution accuracy, based on how dense and frequent those method transitions are expected to be.

7.2 What Is Useful to Report in a Profile

The previous section outlined a possible strategy for improving method-level attribution. However, in doing so, it also raises the broader issue of whether methods are always the most useful unit for a profiler to report. If heavily optimised code often yields methods as many short instructions, spread out and mixed throughout the program, then a method-level view may not always provide the most useful information to the user.

Therefore we consider what information a profiler should report, and consequently what information it should collect. This question applies whether the profiler is based on sampling, instrumentation, or a hybrid of the two. In this thesis, we have already explored different reporting units: BuboC profiled whole compilation units in [Chapter 4](#), while BuboL profiled loops in [Chapter 6](#).

We see that many challenges profilers face result partly from the fact that modern compilers change and optimise code so aggressively that the resulting machine code, and therefore its performance behaviour, can bear very little resemblance to the original source code. Take, for example, a loop that executes 100 iterations and computes a number. It is possible with modern just-in-time compilers, such as the Graal compiler, that this loop is removed entirely, and the result is loaded as a constant because the computation is predictable and deterministic. From the user's perspective, the loop may have become faster, but not in the usual sense, rather, the loop no longer exists in the final machine code. The issue arises when one wants to accurately track how and where time is spent in such a scenario. How is a profiler to associate and report that work if there may no longer be any loop to profile and associate with the original loop?

Therefore, we may need to consider profilers reporting information that is more aligned with how modern compilers transform source code into final machine code.

We believe that reporting aggressive optimisations could be useful to profiler users. If a tool could use compiler-time information to identify how a section of source code was transformed, it could report that information back in a source-level view. This would give the user some understanding of how the program was compiled, rather than only reporting where time was spent in the final machine code.

For example, such a source-level view could highlight:

- Source-code sections that have been eliminated entirely, showing that they no longer execute and therefore may not be useful targets for manual optimisation.
- Loops that have been merged, indicating that the performance behaviour of the resulting loop may come from multiple source-level loops.

- Loops that have been peeled or otherwise split during optimisation, showing that the final compiled structure no longer matches the original source loop directly.
- Calls that are always inlined, showing that the cost of the callee may be distributed through the caller rather than appearing as a separate method-level cost.

St-Amour et al. (2012) argue that optimising compilers can act too much like black boxes. Developers may need to know why the compiler optimised their code in a particular way, or why an optimisation failed. They propose *optimization coaching*, where optimisation successes and misses are presented directly to the developer.

Their approach is implemented in the Racket ecosystem⁶ and reports type-driven specialisation opportunities, such as replacing generic arithmetic with more specialised integer or floating-point operations. It also reports inlining successes and misses, including why inlining did not occur. As St-Amour et al. (2012) show, exposing where optimisations succeed or fail can help developers make changes that improve performance.

This is similar to the direction we suggest for future profilers. Rather than only reporting where execution time was spent, a future profiler could also bring information about compilation back to the source view. This could help users decide which parts of the source program are meaningful optimisation targets, and which parts have already been removed, merged, or transformed by the compiler. In this sense, the profiler would connect execution cost with the compiler transformations that shaped that cost in the final program.

7.2.1 Related work on non-method-centric profiling

Similar ideas about moving beyond method-level reporting have been explored in prior work. For example, Moseley et al. 2007 presents an instrumentation approach for identifying potential parallelism via loop-centric profiling. They argue that most profilers are method-centric and that, for parallelisation and performance understanding, this perspective is often unhelpful. Instead, they propose making loops the

⁶<https://racket-lang.org/>

primary unit of analysis, reporting where execution time is spent in terms of loop nests rather than only in terms of functions.

Their tool builds a loop call graph, a hierarchical view where loops are treated as first-class nodes (alongside the routines they call). For each loop it reports self cost (work done in that loop body) and total cost (work done in the loop plus its nested loops and any called code), and it also records how often the loop is entered and how many iterations it runs. This helps reveal important outer loops that may look unimportant in a method-level profile, but actually dominate run time because they contain hot inner loops.

Ammons et al. 1997 shows that traditional profilers are limited because they attribute execution costs to static units such as procedures or statements and aggregate behaviour across all executions of a procedure. They note that most profilers are both flow- and context-insensitive, reducing the reported information to a single procedure.

By associating total costs with procedures/methods, conventional tools can hide certain execution paths that are expensive in terms of execution and are therefore unable to explain why certain performance issues arise. Ammons et al. 1997 argue that many performance problems are concentrated along a small number of dynamic execution paths, and that tools reporting metrics only at the procedure or statement level cannot isolate these hot paths. As a result, a purely method-centric view can obscure the true causes of performance behaviour in modern systems, motivating profiling approaches that expose context-sensitive structure rather than relying solely on method-level summaries.

7.3 Conclusion

This chapter discussed the challenges that modern JVM profilers face when attributing execution time to source-level program structures. In particular, we focused on method boundaries and method-level attribution. Compiler optimisations, such as inlining and code motion, can mix instructions from many different methods within a single compilation unit. This creates situations where execution can transition from

one method to another within only a few machine-code instructions. This makes it increasingly difficult for profilers to correctly identify which method is executing.

For sampling-based profilers, this means that thousands, if not millions, of method transitions can occur between samples, leaving large amounts of execution unobserved and increasing reliance on a probabilistic view. For instrumentation-based profilers, tracking every method boundary would require a large number of probes, many of which may be activated extremely frequently, leading to impractical overhead and a strong observer effect. This makes accurate method-level attribution a challenge for both sampling and instrumentation.

To address this, we propose a hybrid profiling approach as a direction for future work. This approach combines late-stage instrumentation, exact timing via RDTSC, and hardware event sampling. Probes define precise start and end points for each measurement window, while hardware event sampling provides information about activity within those windows. This allows the total elapsed time for a window to be measured directly, while still estimating how that time should be distributed across individual methods. The goal is to avoid instrumenting every method boundary while retaining accurate total time measurements.

We also discussed how compiler metadata collected during late-stage instrumentation could improve profiler reporting. Because late compiler-stage instrumentation has access to information about loops, instructions, inlining decisions, debug metadata, and code transformations, it can support richer attribution than a method-level profile. This information could also be used to identify regions where method-level attribution is likely to be unreliable, for example when compiler metadata indicates many method transitions but sampled attribution reports little variation, allowing confidence in the reported results to be provided alongside the measurements.

Finally, we argued that profiler output should better reflect the structure of optimised code. Modern compilers can transform, merge, eliminate, or replace source-level constructs, meaning that traditional method-centric profiles may not always provide the most useful view of performance. Future profilers should therefore consider reporting information about transformed code, optimisation effects, and loop structure. Such reporting may give developers a more accurate and actionable understanding of where execution time is spent in highly optimising compilation,

dynamically compiled systems.

Chapter 8

Conclusion & Future Work

We now come to the end of this thesis, where we discuss the conclusions of our findings as well as future work.

8.1 Conclusion

The goal of this thesis was to determine and improve the accuracy of CPU profilers in the presence of highly optimising compilation and modern hardware. We showed, for a selection of state-of-the-art Java profilers, that they have accuracy and precision issues. Sampling profilers can disagree with each other, vary between runs, and misattribute execution time. Java bytecode instrumentation-based profilers, meanwhile, can change compilation and execution behaviour to such a degree that their measurements no longer reflect the performance behaviour of the original program.

We first examined sampling-based profilers for Java in [Chapter 3](#), to determine whether known sources of sampling bias still lead to inaccurate results in heavily optimised programs. We found that sampling profilers are generally self-consistent: when the same profiler is run repeatedly on the same benchmark, it usually reports the same hottest methods, but that worst-case scenarios still existed where their results varied substantially. More importantly, when comparing different sampling profilers on the same deterministic benchmark, they often disagreed about which methods accounted for the most execution time. Smaller methods were often unstable, with large changes between runs, making them difficult to track reliably. For the Are We Fast Yet benchmarks we found that sampling profilers generally have low overhead, ranging from 1% to 5.41%.

As an alternative to sampling, in [Chapter 4](#) we examined Java bytecode instrumentation-based profilers to determine the extent to which their overhead and observer effect influence the measured system. We found that state-of-the-art instrumentation-based profilers incur a massive run-time penalty, with overheads ranging from $82\times$ to $374\times$ on the Are We Fast Yet benchmarks. They also have a significant impact on compilation, increasing the amount of generated machine code by a median of between 273% and 341%. In addition, disabling inlining had little to no effect on the resulting profiles, indicating that the instrumentation interferes with the program to such a degree that inlining no longer has a meaningful impact on what is measured.

To further investigate instrumentation, we implemented BuboC, a late-compiler-stage instrumentation-based profiler that records time spent in compilation units. BuboC shows that inserting instrumentation late in the compilation pipeline can reduce interference with compiler decisions and therefore reduce the observer effect. Compared to state-of-the-art instrumentation-based profilers, BuboC has a far lower overhead, with a median overhead of 23.3%, while still providing more precise measurements than sampling-based profilers.

One of the greatest challenges in evaluating profilers is that there is no known ground truth against which to compare their results. With the complexity of modern systems, including dynamic compilation and modern hardware behaviour such as pipelining and out-of-order execution, it is extremely difficult to determine how and where a program truly spends its execution time. To address this, in [Chapter 5](#) we introduced a novel approach based on deliberate machine-code slowdown. By inserting additional instructions into machine-code basic blocks, we induce known slowdowns and can then test whether profilers correctly attribute those slowdowns to the intended regions of the program. To determine where and by how much to slow down the program, we implemented our methodology of *divining*, which determines the required amount of slowdown. We evaluated six x86 instruction candidates and found that MOV and NOP are the most suitable for controlled slowdown.

Using this approach, we evaluated sampling profilers and assessed their accuracy against known performance changes. We found that profilers such as Async, JFR, and YourKit correctly associated the slowdown behaviour in the majority of test cases. However, we also found that tools such as JProfiler failed the majority of

our evaluation tests and incorrectly associated slowdown with the wrong target methods. These results indicate that profiler inaccuracy is strongly affected by the quality of source information and by a tool's ability to associate samples with the correct program location.

To overcome the key accuracy issues identified in the previous chapters, we then experimented with a new type of profiler in [Chapter 6](#). There, we implemented BuboL, a loop-level profiler that instruments code even later in the compilation pipeline, at the LIR level. Using our slowdown methodology, we showed that BuboL is more accurate than sampling profilers when reporting the CPU time spent in loops. BuboL had a median difference of 1.7% from the baseline, compared with 12.1% for Async and 18.0% for JFR. With a practical overhead of 18.6%, BuboL demonstrates that late-compiler-stage instrumentation can achieve greater accuracy than sampling profilers while still preserving realistic program behaviour at manageable overhead.

Finally, using the understanding gained throughout the thesis, [Chapter 7](#) discusses method-level attribution as a central challenge for modern JVM profilers. Profilers commonly report where time is spent at the method level, but compiler optimisations can transform programs so significantly that generated machine code no longer maps cleanly to a single source-level method. We showed this with the Are We Fast Yet benchmarks, where billions of method changes can occur during execution, meaning that millions can happen between samples at the default sampling interval. This makes it likely that sampling profilers miss substantial method-level execution and cannot always accurately account for where time is spent. As future work, we proposed a hybrid profiling approach that combines late compiler-stage instrumentation with sampling. Instrumentation provides precise timing windows, while sampling of the program counter and compiler metadata estimate how time is distributed within those windows and flag regions where method-level attribution is likely unreliable.

8.2 Future Work

Across this thesis we identified multiple areas where further work can improve the accuracy and usefulness of CPU profilers, particularly for JIT-compiled systems.

Compiler-aware profiling feedback For instrumentation-based profilers, an interesting direction for future work is to explore how late-compile-stage instrumentation, where probes are inserted after most major optimisation decisions, could use compiler knowledge to provide more direct feedback to developers and compiler engineers (Chapter 4). Tools such as BuboC demonstrate that profiling at the compiler level provides access to rich information about the program after optimisation. Instead of merely identifying which method takes the most time, future tools could use this knowledge of the optimised program structure to guide developers towards concrete optimisation opportunities based on the compiled code.

A key advantage of compiler-level profilers, as seen with BuboC, is their access to detailed compiler information, such as inlining history, source mappings, and optimisation decisions. For application developers, this could explain why a source-level region becomes slow or fast after optimisation, without requiring them to understand the compiler internals directly. For compiler engineers, it could show how specific optimisation decisions affect overall program performance or particular regions of code. Combined with minimal side effects on optimisation, this approach offers a practical and realistic way to connect the behaviour of the optimised program back to actionable feedback.

Extending machine-code slowdown across architectures Future work is also required to extend the machine-code slowdown methodology introduced in this thesis (Chapter 5). Currently, the approach has only been tested on Intel Skylake microarchitectures. Although the idea should apply more broadly, additional work is required to evaluate it across different CPU architectures and adapt aspects such as the slowdown instructions used, heuristics to prevent pipeline stalling, and other microarchitectural considerations that may affect performance behaviour. Another important direction is compensating for inaccuracies in hardware counter sampling. If we can compensate for effects such as instruction skid, slowdown instructions could be inserted more precisely within basic blocks. Currently, slowdown instructions are distributed evenly, but in cases where instructions from multiple methods appear within the same basic block, it would be beneficial to slow down instructions belonging to only

one of those methods. This would enable the evaluation of profiler accuracy not only at the method level but potentially down to the source-expression or bytecode level.

Applications of basic-block-level slowdown Additionally, the ability to slow down programs at the basic-block level opens opportunities to explore related techniques such as *virtual speedup* (Curtsinger et al. 2015), which can be used to predict the performance benefit of optimising specific code regions.

Richer debugging information We identified in Chapter 5 that a key source of sampling-based profiler inaccuracy is poor debugging information, in particular a lack of available information. For future work, compiler designs could aim to preserve more debug information and provide richer information.

Compiler decisions can generate instructions with no associated debugging information. For example, in the Graal compiler, linear scan register allocation produces many MOV instructions with no debug information, as they are backend operations not directly tied to the source program. As a result, when time is spent in these instructions, sampling-based profilers must guess the correct association, typically using the nearest available information, which can lead to inaccuracies.

We suggest that where clear associations can be made, such as within encapsulated blocks of instructions, debugging information should be assigned accordingly, even at the cost of increased compilation time. This could be provided as a configurable option, similar to `-XX:+DebugNonSafepoints`, enabled when accurate debugging information is required.

Additionally, optimisations such as loop fusion can merge code from multiple methods into a single region, making attribution difficult. As future work, richer debugging information could allow multiple ownerships, enabling profilers to attribute execution time to more than one source. For example, if a loop originates from two methods, assigning time to both may be more accurate than assigning it to only one or none.

Extending loop-level profiling For loop-level profiling (Chapter 6), future work could take forward the idea of loop-level profilers to multiple program constructs.

As outlined in our discussion in [Section 7.1.4](#), a tool that can instrument different levels of program scope could move us towards profiling tools that provide both accurate information and information that is useful to the developer. Instead of generalising all recorded information at the level of individual methods, such a tool could provide feedback that identifies specific sections of source code which, once fully compiled and optimised, become expensive. This would allow developers to focus their manual optimisation efforts on the parts of the program that would most benefit overall program performance.

8.3 Closing Words

Overall, this thesis shows that current state-of-the-art profilers are not adequately adapted to the complexity of highly optimised code. As compilation has become more complex, profiling tools need to understand and account for the transformed program that actually executes, rather than relying only on source-level constructs or assumptions that no longer hold.

Our work demonstrates that profiler accuracy can be determined, current profilers still disagree on where time is spent, misattribute execution time, and are affected by the observer effect. Additionally, moving instrumentation later in the compiler, closer to the final generated machine code, provides a viable path towards more accurate profiling tools. Future profilers therefore need to rethink what they report, as traditional method-level attribution has become increasingly difficult to measure in the presence of highly optimising compilation. This thesis contributes by showing that tools and methodologies can be built to give a more accurate understanding of program performance behaviour in a highly optimised world.

Appendix A

Appendix

The following appendix is organised into sections that denote the relevant data and figures for their associated chapters.

A.3 Chapter 3 Appendix

TABLE A.1: The following table contains part of our raw data to determine the hottest method to complement [Figure 3.5](#) and [Figure 3.11](#).

Profiler	Method	% of run time	Profiler	Method	% of run time
<i>Bounce</i>			<i>Permute</i>		
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%
<i>CD</i>			<i>Queens</i>		
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%
<i>Havlak</i>			<i>Richards</i>		
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%
<i>Json</i>			<i>Sieve</i>		
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%
<i>Mandelbrot</i>			<i>Storage</i>		
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%
<i>NBody</i>			<i>Towers</i>		
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%

A.4 Chapter 4 Appendix

TABLE A.2: A complete version of Table 4.1. 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

A.5 Chapter 5 Appendix

TABLE A.3: Overhead percentages for benchmarks at different slowdown speeds.
These are the data points shown in Figure 5.8a.

Benchmark	Slowdown-50 Overhead	Slowdown-100 Overhead	Slowdown-150 Overhead
Bounce	56.1%	99.1%	142.5%
List	55.6%	102.2%	146.6%
Mandelbrot	53.1%	101.6%	154.7%
NBody	50.0%	100.0%	149.1%
Permute	46.9%	100.0%	146.9%
Queens	50.0%	94.3%	142.9%
Sieve	48.6%	98.9%	146.4%
Storage	54.7%	102.7%	161.5%
Towers	58.0%	106.2%	160.2%
CD	46.4%	96.4%	144.5%
DeltaBlue	50.5%	90.2%	139.5%
Havlak	47.0%	115.0%	148.0%
Json	45.6%	94.1%	147.1%
Richards	48.8%	109.8%	143.9%

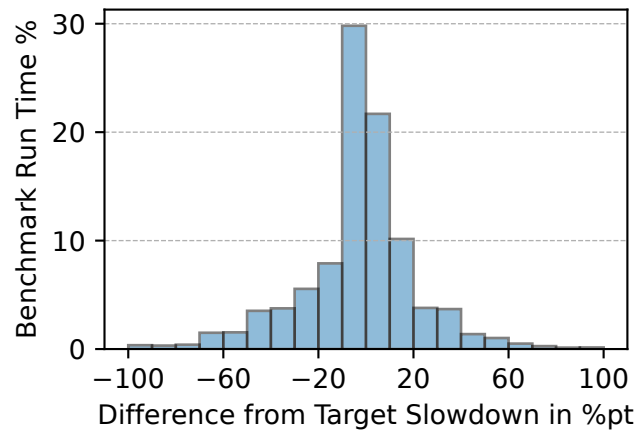


FIGURE A.1: Histogram shows the distribution of overhead differences from the target slowdown of 50% across all benchmarks. The x-axis represents the difference in percentage points. The y-axis shows the percentage of benchmark run time. The results show that the majority of run time is spent in blocks with overheads close to the target slowdown.

TABLE A.4: Summary of target slowdown performance for each profiler. **Total Target Methods** denotes the number of slowdown blocks tested per benchmark. **Detected in Baseline** indicates whether the target method was identified in the baseline run. **Detected in Slowdown** indicates whether the target method was identified in the slowdown run. **Baseline Usage (%)** and **Slowdown Usage (%)** represent the percentage of run time attributed to the target method in the baseline and slowdown runs, respectively. **Actual Change (%)** is the change in usage from baseline to slowdown, **Predicted Change (%)** is the expected change if all additional time were attributed to the target method, and **Error (%)** is the difference between the actual and predicted changes. For each target block, benchmark, and profiler, the reported value is the median of 10 runs.

Profiler	Benchmark	Target Block	Target Method	Detected in Baseline	Detected in Slowdown	Baseline Usage (%)	Slowdown Usage (%)	Actual Change (%)	Predicted Change (%)	Error (%)
Async	Havlak	1	Vector.hasSome	True	True	33.78	37.17	3.38	3.06	0.32
		2	HavlakLoopFinder.doDFS	True	True	2.13	6.73	4.6	4.52	0.08
		3	HavlakLoopFinder.lambda\$stepD\$6	False	False	—	—	—	—	—
	Json	1	String.equals	True	True	34.83	43.34	8.51	8.26	0.25
		2	Vector.append	True	True	2.17	15.07	12.89	12.41	0.49
		3	String.equals	True	True	34.83	44.07	9.24	8.26	0.97
	List	1	List.isShorterThan	True	True	42.48	48.53	6.05	3.05	3
		2	List.tail	True	True	12	9.61	-2.4	4.67	-7.07
		3	List\$Element.getNext	True	True	44.5	47.28	2.78	2.94	-0.16
	Richards	1	TaskState.isTaskHoldingOrWaiting	True	True	7.81	19.55	11.73	9.92	1.81
		2	Scheduler.schedule	True	True	19.73	28.96	9.23	8.64	0.6
		3	TaskState.isWaitingWithPacket	True	True	2.39	13.36	10.96	10.5	0.46
JFR	Havlak	1	Vector.hasSome	True	True	40.8	44.73	3.94	2.76	1.17
		2	HavlakLoopFinder.doDFS	True	True	2.29	7.89	5.6	4.56	1.05
		3	HavlakLoopFinder.lambda\$stepD\$6	False	False	—	—	—	—	—
	Json	1	String.equals	True	True	42.3	50.6	8.3	7.3	0.99
		2	Vector.append	True	True	2.85	17.5	14.64	12.29	2.35
		3	String.equals	True	True	42.3	51.19	8.88	7.3	1.58
	List	1	List.isShorterThan	True	True	43.52	50.5	6.98	2.96	4.01
		2	List.tail	True	True	7.46	5.77	-1.69	4.86	-6.55
		3	List\$Element.getNext	True	True	48.16	50.23	2.07	2.72	-0.65
	Richards	1	TaskState.isTaskHoldingOrWaiting	True	True	8.8	21.84	13.04	9.95	3.09
		2	Scheduler.schedule	True	True	22.42	32.08	9.66	8.46	1.2
		3	TaskState.isWaitingWithPacket	True	True	2.68	14.65	11.97	10.62	1.35
JProfiler	Havlak	1	Vector.hasSome	False	False	—	—	—	—	—
		2	HavlakLoopFinder.doDFS	True	True	2.1	12.7	10.6	4.71	5.89
		3	HavlakLoopFinder.lambda\$stepD\$6	False	False	—	—	—	—	—
	Json	1	String.equals	False	False	—	—	—	—	—
		2	Vector.append	False	False	—	—	—	—	—
		3	String.equals	False	False	—	—	—	—	—
	List	1	List.isShorterThan	True	True	43.8	90.3	46.5	2.99	43.51
		2	List.tail	True	True	9.7	8.9	-0.8	4.8	-5.6
		3	List\$Element.getNext	False	False	—	—	—	—	—
	Richards	1	TaskState.isTaskHoldingOrWaiting	True	True	8	0	-8	10.16	-18.16
		2	Scheduler.schedule	True	True	25.15	17	-8.15	8.26	-16.41
		3	TaskState.isWaitingWithPacket	False	False	—	—	—	—	—
YourKit	Havlak	1	Vector.hasSome	True	True	36.51	36.32	-0.19	2.99	-3.18
		2	HavlakLoopFinder.doDFS	True	True	2.23	7.36	5.13	4.6	0.53
		3	HavlakLoopFinder.lambda\$stepD\$6	False	False	—	—	—	—	—
	Json	1	String.equals	True	True	37.45	45.84	8.39	7.93	0.46
		2	Vector.append	True	True	2.33	15.89	13.56	12.38	1.18
		3	String.equals	True	True	37.45	46.19	8.74	7.93	0.81
	List	1	List.isShorterThan	True	True	44.64	50.05	5.41	2.94	2.47
		2	List.tail	True	True	9.47	7.9	-1.57	4.81	-6.38
		3	List\$Element.getNext	True	True	44.84	46.84	2.01	2.93	-0.93
	Richards	1	TaskState.isTaskHoldingOrWaiting	True	True	8.99	20.04	11.05	9.83	1.22
		2	Scheduler.schedule	True	True	18.87	28.38	9.52	8.76	0.76
		3	TaskState.isWaitingWithPacket	True	True	2.59	14.09	11.5	10.52	0.98

TABLE A.5: Divining time and slowdown-file metrics for each benchmark at 50%, 100% and 150% imposed slowdown. The *Total* refers to the number of slowdown instructions inserted into the final machine code of the benchmark while the *Median* is the median number of instructions inserted into a basic block at the final machine-code level. Note, the data for this table is extracted based on the slowdown files, while Table 5.4 is directly reported by Graal. Furthermore, they are from different runs, which means there can be small differences in the data.

(a) 50% slowdown

Benchmark	Comp Units	Native Basic Blocks	Runs	Elapsed (hh:mm)	Total	Median
Bounce	1	21	36	01:03	120	3
List	2	59	32	01:56	162	2
Mandelbrot	1	11	53	01:24	150	3
NBody	1	1	61	01:46	553	553
Permute	1	16	19	00:32	45	2
Queens	1	107	14	00:17	283	2
Sieve	1	17	68	03:08	916	2
Storage	2	80	56	02:32	505	4
Towers	1	52	13	00:34	162	3
CD	9	343	244	07:18	1663	3
DeltaBlue	14	230	319	15:28	1744	3
Havlak	9	401	236	11:13	5986	3
Json	10	470	250	06:06	2550	3
Richards	3	84	99	02:32	389	3

(b) 100% slowdown

Benchmark	Comp Units	Native Basic Blocks	Runs	Elapsed(hh:mm)	Total	Median
Bounce	1	21	51	01:55	214	5
List	2	59	49	03:18	325	3
Mandelbrot	1	11	94	02:51	232	7
NBody	1	1	50	01:28	1072	1072
Permute	1	16	24	00:44	93	6
Queens	1	107	22	00:30	547	4
Sieve	1	17	82	04:47	2014	6
Storage	2	80	74	03:39	806	7
Towers	1	52	20	00:52	304	6
CD	9	343	367	11:12	4216	6
DeltaBlue	14	230	491	25:06	3086	7
Havlak	9	401	400	21:25	23200	7
Json	10	470	329	08:36	5197	7
Richards	3	84	141	03:45	946	6

(c) 150% slowdown

Benchmark	Comp Units	Native Basic Blocks	Runs	Elapsed(hh:mm)	Total	Median
Bounce	1	21	78	03:29	299	8
List	2	59	61	04:38	450	5
Mandelbrot	1	11	80	02:46	307	9
NBody	1	1	109	04:42	1558	1558
Permute	1	16	42	01:29	142	9
Queens	1	107	22	00:35	737	6
Sieve	1	17	98	06:53	5036	10
Storage	2	80	80	03:02	1777	14
Towers	1	52	26	01:14	447	9
CD	9	343	386	12:17	6587	11
DeltaBlue	14	230	564	29:18	4673	11
Havlak	9	401	511	29:14	43785	10
Json	10	470	383	10:08	9402	13
Richards	3	84	194	05:29	948	8

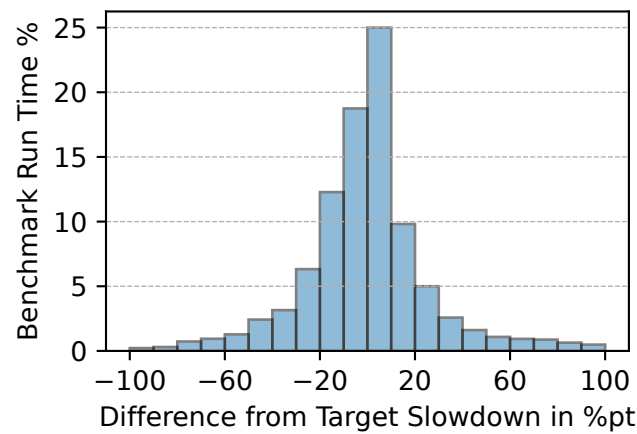


FIGURE A.2: Histogram shows the distribution of overhead differences from the target slowdown of 150% across all benchmarks. The x-axis represents the difference in percentage points. The y-axis shows the percentage of benchmark run time. The results show that the majority of run time is spent in blocks with overheads close to the target slowdown.

TABLE A.6: Benchmarking time and number of unique runs per research question

Research Question	Benchmarking Time (h)	Unique Runs
RQ 1	36	1 260
RQ 2	93	3 360
RQ 3	10	480
Total	139	5 100

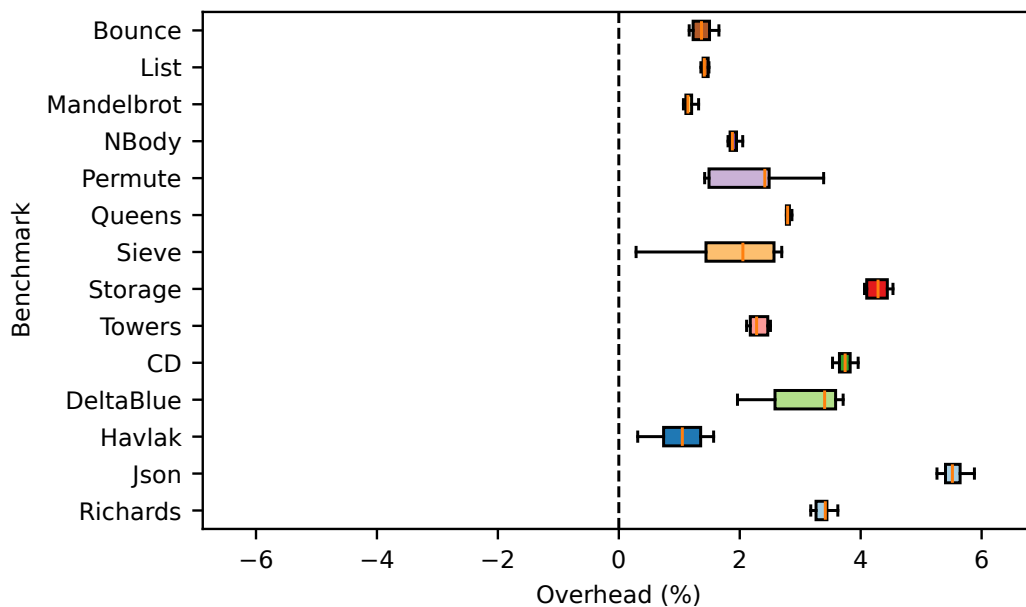


FIGURE A.3: Displays the overhead measured across 10 runs for each benchmark when using VTune in hardware sampling mode with stack collection. The baseline represents the median runtime with no profiling tools attached.

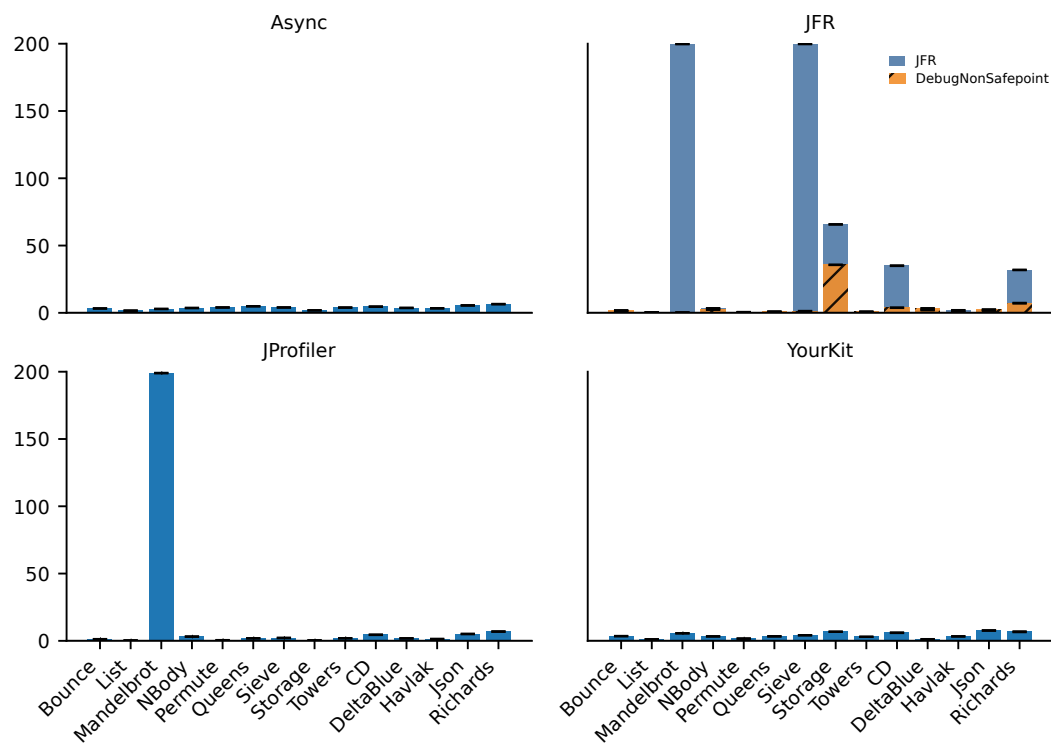


FIGURE A.4: Bar chart showing the absolute change in percentage points for all methods for each benchmark, when transitioning from safepoints being present to removed. The purpose is to demonstrate how profilers adjust their results when no safepoints are available in the program. JFR includes two versions, illustrating the difference in results when the `-XX:+DebugNonSafepoints` flag is enabled.

A.6 Chapter 6 Appendix

Benchmark	Workload focus
Fibonacci	Fibonacci sequence computation
Mandelbrot	Mandelbrot set iteration, different implementation to AWFY's version
JSON parsing	Character scanning and state machine parsing
Matrix multiply	Dense matrix multiplication
Prime sieve	Sieve of Eratosthenes style prime marking
Histogram	Histogram computation over integer data
CRC32	CRC32 style checksum computation
Polynomial	Polynomial evaluation using Horner's method
Edit distance	Levenshtein distance dynamic programming
Monte Carlo pi	Monte Carlo estimation of π
Bubble sort	Bubble sort over an integer array
Image blur	2D stencil blur over an image grid
Base64 encode	Base64 encoding over a byte array
Ray sphere	Ray sphere intersection computation

TABLE A.7: The LoopsBench benchmark suite, consisting of 14 loop-focused benchmarks.

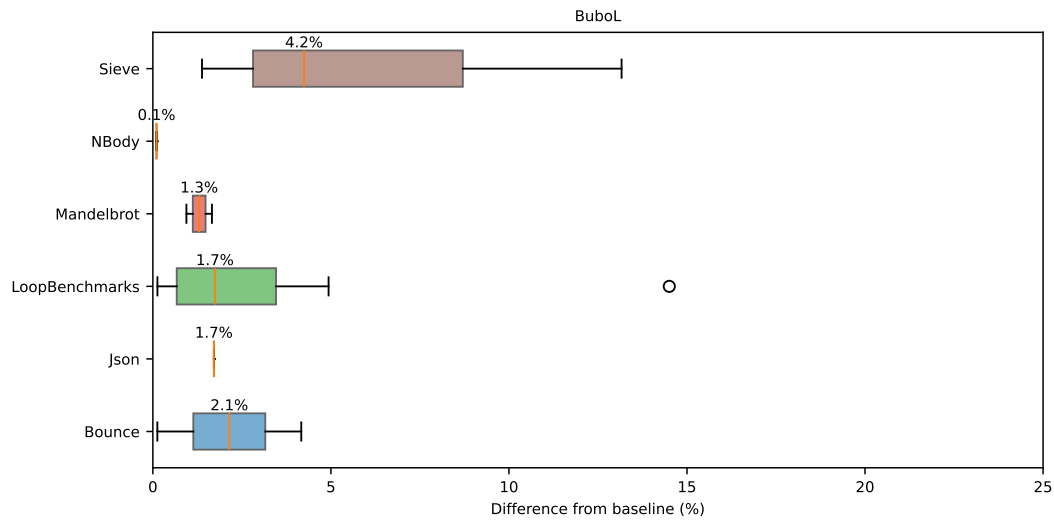


FIGURE A.5: Accuracy of BuboL loop measurements compared to VTune.

Benchmark	Tool	Min (%)	Max (%)	Median (%)
Bounce	BuboL	0.13	4.17	2.15
	JFR	12.254	18.078	15.166
	Async	12.413	14.066	13.240
Json	BuboL	1.71	1.71	1.71
	JFR	35.384	58.191	46.788
	Async	5.726	67.282	36.504
LoopBenchmarks	BuboL	0.13	14.50	1.74
	JFR	1.923	141.31	16.468
	Async	0.583	65.391	10.763
Mandelbrot	BuboL	0.94	1.66	1.30
	JFR	11.235	185.758	98.496
	Async	11.808	189.725	100.766
NBody	BuboL	0.10	0.10	0.10
	JFR	1.175	1.175	1.175
	Async	3.920	3.920	3.920
Sieve	BuboL	1.38	13.16	4.25
	JFR	9.598	68.405	28.925
	Async	5.158	66.888	39.852

TABLE A.8: Per-benchmark breakdown of the accuracy results in Figure 6.8. For each tool, the table shows the minimum, maximum, and median absolute difference between the reported loop slowdown and the slowdown measured by VTune across all evaluated loops. Smaller values indicate greater accuracy.

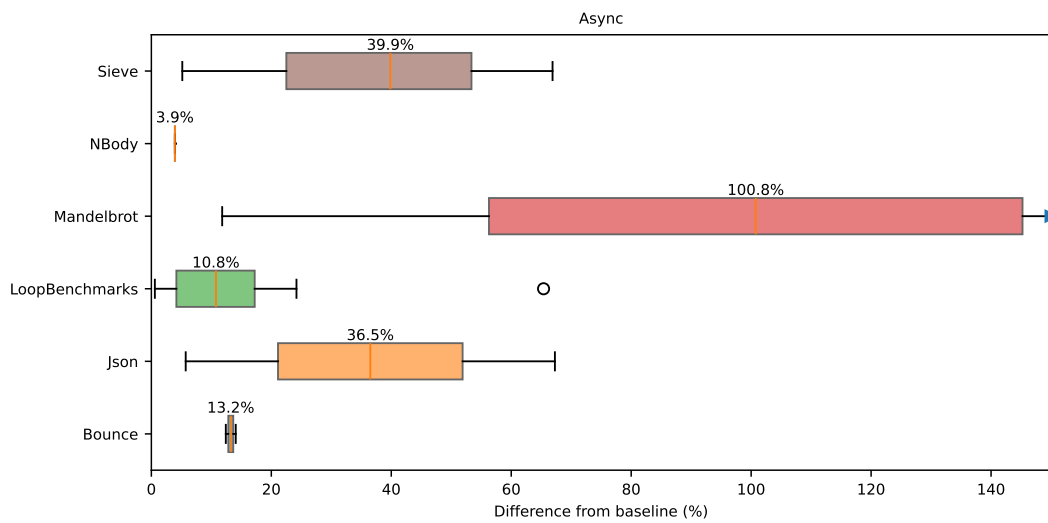


FIGURE A.6: Accuracy of Async-Profiler loop measurements compared to VTune.

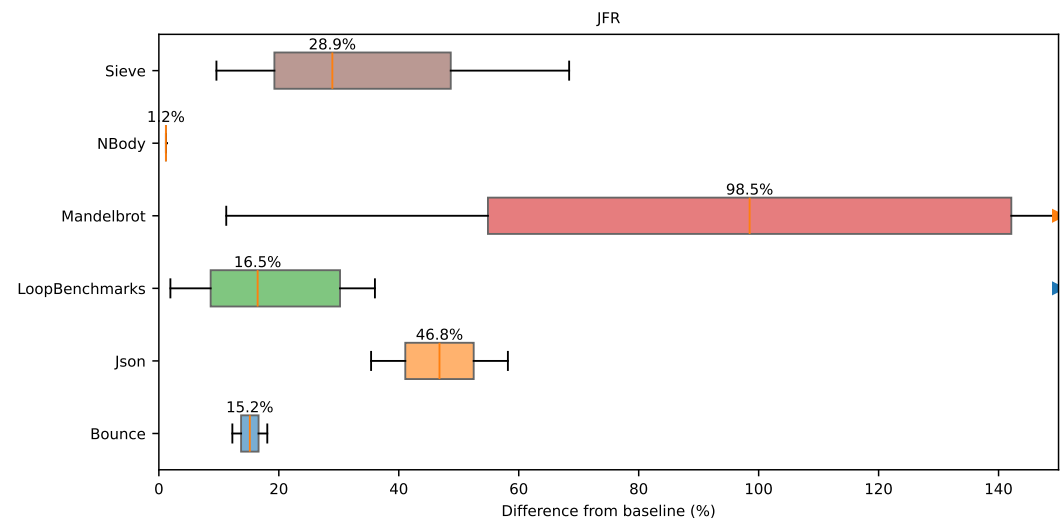


FIGURE A.7: Accuracy of JFR loop measurements compared to VTune.

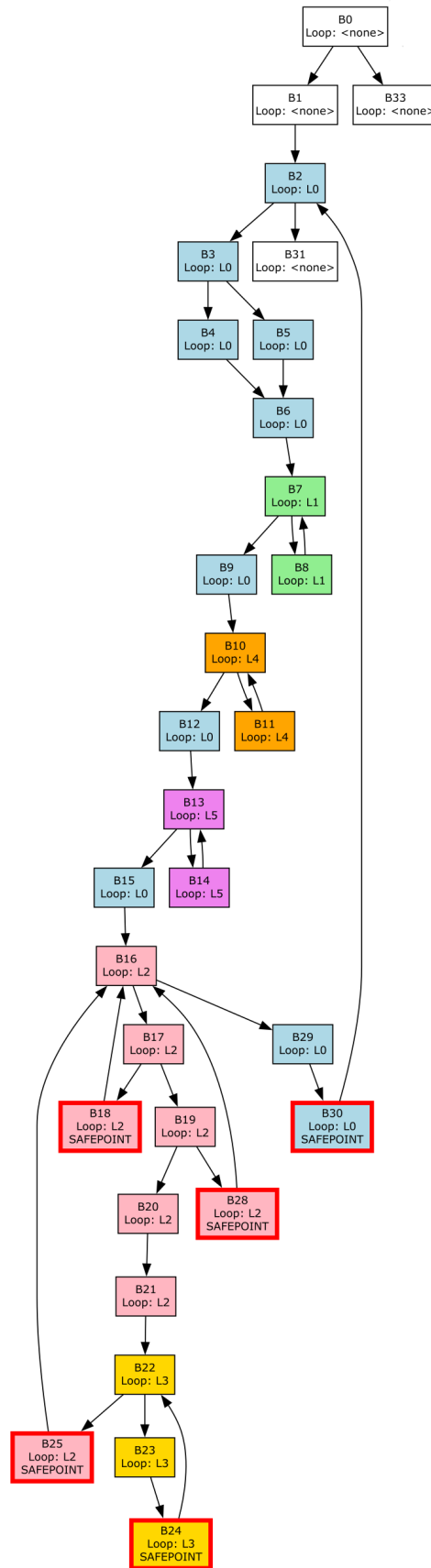


FIGURE A.8: CFG of the Graal LIR blocks for the Sieve benchmark, showing loops and highlighting blocks that contain safepoints.

Bibliography

- Abel, A. and J. Reineke (2022). “uiCA: accurate throughput prediction of basic blocks on recent Intel microarchitectures”. In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS ’22. Virtual Event: ACM. ISBN: 9781450392815. DOI: [10.1145/3524059.3532396](https://doi.org/10.1145/3524059.3532396).
- Agesen, O. (Dec. 1998). *GC Points in a Threaded Environment*. Tech. rep. SMLI TR-98-70. Sun Microsystems.
- Åkerblom, B. and E. Castegren (Feb. 2024). “Arrays in Practice: An Empirical Study of Array Access Patterns on the JVM”. In: *The Art, Science, and Engineering of Programming* 8. DOI: [10.22152/programming-journal.org/2024/8/14](https://doi.org/10.22152/programming-journal.org/2024/8/14).
- Alpern, B., C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen (Oct. 1999). “Implementing Jalapeño in Java”. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA’99. ACM, pp. 314–324. ISBN: 1-58113-238-7. DOI: [10.1145/320385.320418](https://doi.org/10.1145/320385.320418).
- AMD Inc. (2024). *AMD64 Architecture Programmer’s Manual, Volume 3: General-Purpose and System Instructions*. Publication Number 24594, Revision 3.36.
- Ammons, G., T. Ball, and J. R. Larus (1997). “Exploiting hardware performance counters with flow and context sensitive profiling”. In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. PLDI ’97. Las Vegas, Nevada, USA: Association for Computing Machinery, pp. 85–96. ISBN: 0897919076. DOI: [10.1145/258915.258924](https://doi.org/10.1145/258915.258924). URL: <https://doi.org/10.1145/258915.258924>.
- St-Amour, V., S. Tobin-Hochstadt, and M. Felleisen (Oct. 2012). “Optimization Coaching Optimizers Learn to Communicate with Programmers”. In: *ACM SIGPLAN Notices* 47. DOI: [10.1145/2384616.2384629](https://doi.org/10.1145/2384616.2384629).

- Assaiante, C., D. C. D’Elia, G. A. Di Luna, and L. Querzoni (2023). “Where Did My Variable Go? Poking Holes in Incomplete Debug Information”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: ACM, pp. 935–947. ISBN: 9781450399166. DOI: [10.1145/3575693.3575720](https://doi.org/10.1145/3575693.3575720).
- Bakhvalov, D. (2024). *Performance Analysis and Tuning on Modern CPUs, Second Edition*. Independently published, p. 237. ISBN: 979-8575614234.
- Barrett, E., C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt (Oct. 2017). “Virtual Machine Warmup Blows Hot and Cold”. In: *Proc. ACM Program. Lang.* 1.OOPSLA, 52:1–52:27. ISSN: 2475-1421. DOI: [10.1145/3133876](https://doi.org/10.1145/3133876).
- Basso, M., A. Prokopec, A. Rosà, and W. Binder (June 2023). “Optimization-Aware Compiler-Level Event Profiling”. In: *ACM Trans. Program. Lang. Syst.* 45.2. ISSN: 0164-0925. DOI: [10.1145/3591473](https://doi.org/10.1145/3591473).
- Berger, E. D., S. Stern, and J. A. Pizzorno (July 2023). “Triangulating Python Performance Issues with SCALENE”. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association.
- Blackburn, S. M., R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann (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, pp. 169–190. ISBN: 1-59593-348-4. DOI: [10.1145/1167473.1167488](https://doi.org/10.1145/1167473.1167488).
- Burchell, H., O. Larose, S. Kaleba, and S. 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, pp. 100–113. ISBN: 9798400703805. DOI: [10.1145/3617651.3622985](https://doi.org/10.1145/3617651.3622985).
- Burchell, H., O. Larose, and S. 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 2024. Vienna, Austria: ACM, pp. 82–89. ISBN: 9798400711183. DOI: [10.1145/3679007.3685058](https://doi.org/10.1145/3679007.3685058).
- Burchell, H. and S. Marr (Oct. 2025a). “Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown”. In: *Proceedings of the ACM on Programming Languages*. OOPSLA’25 9.OOPSLA2. ISSN: 2475-1421. DOI: [10.1145/3763180](https://doi.org/10.1145/3763180).
- Burchell, H. and S. Marr (Aug. 2025b). *Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown (Artifact)*. DOI: [10.5281/zenodo.16911348](https://doi.org/10.5281/zenodo.16911348). URL: <https://doi.org/10.5281/zenodo.16911348>.
- Burchell, H. and S. Marr (Oct. 2025c). “Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level: Towards Supporting Fine-grained Slowdown for Advanced Developer Tooling”. In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL’25. Singapore: ACM, p. 8. DOI: [10.1145/3759548.3763374](https://doi.org/10.1145/3759548.3763374).
- Carlson, T. E., W. Heirman, and L. Eeckhout (2011). “Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. Seattle, Washington: ACM. ISBN: 9781450307710. DOI: [10.1145/2063384.2063454](https://doi.org/10.1145/2063384.2063454).
- Click, C. and M. 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, pp. 35–49. ISBN: 0-89791-754-5. DOI: [10.1145/202529.202534](https://doi.org/10.1145/202529.202534).
- Colwell, R. P. (Nov. 2021). “The Origin of Intel’s Micro-Ops”. In: *IEEE Micro* 41.6, pp. 37–41. ISSN: 1937-4143. DOI: [10.1109/MM.2021.3112026](https://doi.org/10.1109/MM.2021.3112026).
- Curtsinger, C. and E. D. Berger (2015). “Coz: finding code that counts with causal profiling”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, pp. 184–197. ISBN: 9781450338349. DOI: [10.1145/2815400.2815409](https://doi.org/10.1145/2815400.2815409).

- De Bus, B., D. Chagnet, B. De Sutter, L. Van Put, and K. De Bosschere (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, pp. 29–34.
- Duboscq, G., T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. 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, pp. 1–10. ISBN: 9781450326018. DOI: [10.1145/2542142.2542143](https://doi.org/10.1145/2542142.2542143).
- Endo, A. T. and A. Möller (2025). “Event Race Detection for Node.js Using Delay Injections”. In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. Vol. 333. LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 9:1–9:28. DOI: [10.4230/LIPICS.ECOOP.2025.9](https://doi.org/10.4230/LIPICS.ECOOP.2025.9).
- Fog, A. (2022). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Tech. rep. URL: https://www.agner.org/optimize/instruction_tables.pdf.
- Frenot, L. and F. M. Q. Pereira (2024). “Reducing the Overhead of Exact Profiling by Reusing Affine Variables”. In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. Edinburgh, United Kingdom: Association for Computing Machinery, pp. 150–161. ISBN: 9798400705076. DOI: [10.1145/3640537.3641569](https://doi.org/10.1145/3640537.3641569). URL: <https://doi.org/10.1145/3640537.3641569>.
- Georges, A., L. Eeckhout, and D. Buytaert (Oct. 2008). “Java Performance Evaluation through Rigorous Replay Compilation”. In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOP-SLA’08. ACM, pp. 367–384. DOI: [10.1145/1449764.1449794](https://doi.org/10.1145/1449764.1449794).
- Gottschall, B., L. Eeckhout, and M. Jahre (Oct. 2021). “TIP: Time-Proportional Instruction Profiling”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. ACM, pp. 15–27. ISBN: 9781450385572. DOI: [10.1145/3466752.3480058](https://doi.org/10.1145/3466752.3480058).
- Graham, S. L., P. B. Kessler, and M. K. McKusick (June 1982). “gprof: a Call Graph Execution Profiler”. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler*

- Construction*. SIGPLAN '82. Boston, Massachusetts, USA: ACM, pp. 120–126. ISBN: 0897910745. DOI: [10.1145/800230.806987](https://doi.org/10.1145/800230.806987).
- Gregg, B. (2013). *Systems Performance: Enterprise and the Cloud*. Upper Saddle River, NJ: Prentice Hall. ISBN: 978-0-13-339009-4.
- He, W., H. Yu, L. Wang, and T. Oh (2024). “Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation”. In: *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '24. Edinburgh, United Kingdom: IEEE Press, pp. 322–333. ISBN: 9798350395099. DOI: [10.1109/CGO57630.2024.10444807](https://doi.org/10.1109/CGO57630.2024.10444807). URL: <https://doi.org/10.1109/CGO57630.2024.10444807>.
- Hennessy, J. L. and D. A. Patterson (2017). *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann, p. 936. ISBN: 0128119055.
- Hofer, P., D. Gnedt, and H. Mössenböck (2015). “Lightweight Java Profiling with Partial Safepoints and Incremental Stack Tracing”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, pp. 75–86. ISBN: 978-1-4503-3248-4. DOI: [10.1145/2668930.2688038](https://doi.org/10.1145/2668930.2688038).
- IBM (1964). *IBM System/360 Principles of Operation*. IBM Press.
- Intel (Sept. 2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2C, Instruction Set Reference, V-Z*. Intel. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2c-manual.pdf>.
- Intel (2024). *Optimizing Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency*. Tech. rep. URL: <https://www.intel.com/content/www/us/en/content-details/814199/optimizing-earlier-generations-of-intel-64-and-ia-32-processor-architectures-throughput-and-latency.html>.
- Intel Corporation (Sept. 2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. 253669-060US. Intel Corporation. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf>.
- Jain, S., K. Kumar, S. Purini, D. Das, and R. Upadrasta (2017). *An LLVM Based Loop Profiler*. Presented at the US LLVM Developers' Meeting. Talk slides.

- Kell, S. and J. R. Stinnett (2024). "Source-Level Debugging of Compiler-Optimised Code: Ill-Posed, but Not Impossible". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! '24. ACM, pp. 38–53. ISBN: 9798400712159. DOI: [10.1145/3689492.3690047](https://doi.org/10.1145/3689492.3690047).
- Kumar, N., B. R. Childers, and M. L. Soffa (Sept. 2005). "Low Overhead Program Monitoring and Profiling". In: *SIGSOFT Softw. Eng. Notes* 31.1, pp. 28–34. ISSN: 0163-5948. DOI: [10.1145/1108768.1108801](https://doi.org/10.1145/1108768.1108801). URL: <https://doi.org/10.1145/1108768.1108801>.
- Li, S., Y. Cheng, J. Chen, J. Xuan, S. He, and W. Shang (Jan. 2026). "Performance analysis of AI-generated code: A case study of Copilot, Copilot Chat, CodeLlama, and DeepSeek-Coder models". In: *Empirical Software Engineering* 31. DOI: [10.1007/s10664-025-10776-1](https://doi.org/10.1007/s10664-025-10776-1).
- Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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*. PLDI '05. Chicago, IL, USA: ACM, pp. 190–200. ISBN: 1595930566. DOI: [10.1145/1065010.1065034](https://doi.org/10.1145/1065010.1065034). URL: <https://doi.org/10.1145/1065010.1065034>.
- Machkasova, E., K. Arhelger, and F. 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*. OOPSLA '09. Orlando, Florida, USA: ACM, pp. 757–758. ISBN: 9781605587684. DOI: [10.1145/1639950.1640000](https://doi.org/10.1145/1639950.1640000).
- Mahlke, S., T. Moseley, R. Hank, D. Bruening, and H. K. 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. USA: IEEE Computer Society, pp. 1–10. ISBN: 9781467355247. DOI: [10.1109/CGO.2013.6494982](https://doi.org/10.1109/CGO.2013.6494982). URL: <https://doi.org/10.1109/CGO.2013.6494982>.
- Manner, J. (Dec. 2022). "Black software - the energy unsustainability of software systems in the 21st century". In: *Oxford Open Energy* 2. DOI: [10.1093/ooenergy/oiac011](https://doi.org/10.1093/ooenergy/oiac011).

- Marr, S. (Aug. 2023). *ReBench: Execute and Document Benchmarks Reproducibly*. Version 1.2.0. DOI: [10.5281/zenodo.8219119](https://doi.org/10.5281/zenodo.8219119). URL: <https://github.com/smarr/ReBench/>.
- Marr, S., H. Burchell, and F. Niephaus (2022). “Execution vs. Parse-Based Language Servers: Tradeoffs and Opportunities for Language-Agnostic Tooling for Dynamic Languages”. In: *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2022. Auckland, New Zealand: Association for Computing Machinery, pp. 1–14. ISBN: 9781450399081. DOI: [10.1145/3563834.3567537](https://doi.org/10.1145/3563834.3567537). URL: <https://doi.org/10.1145/3563834.3567537>.
- Marr, S., B. Dalozé, and H. Mössenböck (Nov. 2016). “Cross-Language Compiler Benchmarking—Are We Fast Yet?”. In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS’16. Amsterdam, Netherlands: ACM, pp. 120–131. ISBN: 978-1-4503-4445-6. DOI: [10.1145/2989225.2989232](https://doi.org/10.1145/2989225.2989232).
- Misurda, J., J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa (2005). “Demand-Driven Structural Testing with Dynamic Instrumentation”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, pp. 156–165. ISBN: 1581139632. DOI: [10.1145/1062455.1062496](https://doi.org/10.1145/1062455.1062496).
- Moseley, T., D. A. Connors, D. Grunwald, and R. Peri (2007). “Identifying potential parallelism via loop-centric profiling”. In: *Proceedings of the 4th International Conference on Computing Frontiers*. CF ’07. Ischia, Italy: Association for Computing Machinery, pp. 143–152. ISBN: 9781595936837. DOI: [10.1145/1242531.1242554](https://doi.org/10.1145/1242531.1242554). URL: <https://doi.org/10.1145/1242531.1242554>.
- Musuvathi, M., S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu (Dec. 2008). “Finding and Reproducing Heisenbugs in Concurrent Programs”. In: *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, pp. 267–280.
- Mytkowicz, T., A. Diwan, M. Hauswirth, and P. 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, pp. 187–197. ISBN: 978-1-4503-0019-3. DOI: [10.1145/1806596.1806618](https://doi.org/10.1145/1806596.1806618).
- Orton, I. and A. Mycroft (2021). “Tracing and Its Observer Effect on Concurrency”. In: *MPLR 2021*. Münster, Germany: Association for Computing Machinery, pp. 88–96. ISBN: 9781450386753. DOI: [10.1145/3475738.3480940](https://doi.org/10.1145/3475738.3480940).

- Preuss, A. (2010). *Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure*. Tech. rep. TRID-ID TR10-05, pp. 1–16. DOI: [10.7939/R3GF0MX64](https://doi.org/10.7939/R3GF0MX64).
- Prokopec, A., A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. 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*. PLDI 2019. Phoenix, AZ, USA: ACM, pp. 31–47. ISBN: 9781450367127. DOI: [10.1145/3314221.3314637](https://doi.org/10.1145/3314221.3314637).
- Qasem, A., K. Kennedy, and J. Mellor-Crummey (May 2006). “Automatic tuning of whole applications using direct search and a performance-based transformation system”. In: *J. Supercomput.* 36.2, pp. 183–196. ISSN: 0920-8542. DOI: [10.1007/s11227-006-7957-2](https://doi.org/10.1007/s11227-006-7957-2). URL: <https://doi.org/10.1007/s11227-006-7957-2>.
- Romer, T., G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. 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*. NT’97. Seattle, Washington: USENIX Association, p. 1.
- Roy, A., S. Hand, and T. Harris (2011). “Hybrid binary rewriting for memory access instrumentation”. In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’11. Newport Beach, California, USA: Association for Computing Machinery, pp. 227–238. ISBN: 9781450306874. DOI: [10.1145/1952682.1952711](https://doi.org/10.1145/1952682.1952711). URL: <https://doi.org/10.1145/1952682.1952711>.
- Seelemann, I. (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, pp. 56–68.
- Shen, J. P. and M. H. Lipasti (2005). *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, p. 642. ISBN: 1-4786-0783-1.

- Sjoblom, I., T. Snyder, and E. Machkasova (Jan. 2011). "Can You Trust Your JVM Diagnostic Tools?" In: MICS 2011. Midwest Instruction and Computing Symposium. URL: https://micsymposium.org/mics_2011_proceedings/mics2011_submission_26.pdf.
- Srivastava, A. and A. 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*. PLDI '94. Orlando, Florida, USA: ACM, pp. 196–205. ISBN: 089791662X. DOI: [10.1145/178243.178260](https://doi.org/10.1145/178243.178260). URL: <https://doi.org/10.1145/178243.178260>.
- Stinnett, J. R. and S. Kell (Apr. 2026). "Debugging Debugging Information using Dynamic Call Trees". In: *Proc. ACM Program. Lang.* 10.OOPSLA1. DOI: [10.1145/3798213](https://doi.org/10.1145/3798213). URL: <https://doi.org/10.1145/3798213>.
- Stoica, B. A., S. Lu, M. Musuvathi, and S. Nath (2023). "WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: ACM, pp. 111–126. ISBN: 9781450394871. DOI: [10.1145/3552326.3567507](https://doi.org/10.1145/3552326.3567507). URL: <https://doi.org/10.1145/3552326.3567507>.
- Thompson, K. and D. M. Ritchie (1974). *UNIX programmer's manual fifth edition*. Bell Telephone Laboratories, Inc.
- Tomasulo, R. M. (Jan. 1967). "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM J. Res. Dev.* 11.1, pp. 25–33. ISSN: 0018-8646. DOI: [10.1147/rd.111.0025](https://doi.org/10.1147/rd.111.0025).
- Vukasic, M. and A. Prokopec (2023). "Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code". In: *ACM Transactions on Programming Languages and Systems* 45, pp. 1–64. DOI: [10.1145/3612937](https://doi.org/10.1145/3612937).
- Würthinger, T., C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko (2013). "One VM to rule them all". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, pp. 187–204. ISBN: 9781450324724. DOI: [10.1145/2509578.2509581](https://doi.org/10.1145/2509578.2509581). URL: <https://doi.org/10.1145/2509578.2509581>.

- Xu, H., Q. Wang, S. Song, L. John, and X. Liu (June 2019). "Can we trust profiling results?: understanding and fixing the inaccuracy in modern profilers". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. ACM, pp. 284–295. ISBN: 9781450360791. DOI: [10.1145/3330345.3330371](https://doi.org/10.1145/3330345.3330371).
- Yi, J., B. Dong, M. Dong, and H. Chen (2020). "On the precision of precise event based sampling". In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '20. Tsukuba, Japan: Association for Computing Machinery, pp. 98–105. ISBN: 9781450380690. DOI: [10.1145/3409963.3410490](https://doi.org/10.1145/3409963.3410490). URL: <https://doi.org/10.1145/3409963.3410490>.
- Zheng, Y., L. Bulej, and W. Binder (Oct. 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, pp. 433–450. ISBN: 978-1-4503-3689-5. DOI: [10.1145/2814270.2814281](https://doi.org/10.1145/2814270.2814281).