Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy through Machine Code-Level Slowdown

HUMPHREY BURCHELL, University of Kent, United Kingdom
STEFAN MARR, University of Kent, United Kingdom

Optimizing performance on top of modern runtime systems with just-in-time (JIT) compilation is a challenge for a wide range of applications from browser-based applications on mobile devices to large-scale server applications. Developers often rely on sampling-based profilers to understand where their code spends its time. Unfortunately, sampling of JIT-compiled programs can give inaccurate and sometimes unreliable results.

To assess accuracy of such profilers, we would ideally want to compare their results to a known ground truth. With the complexity of today's software and hardware stacks, such ground truth is unfortunately not available. Instead, we propose a novel technique to approximate a ground truth by accurately slowing down a Java program at the machine-code level, preserving its optimization and compilation decisions as well as its execution behavior on modern CPUs.

Our experiments demonstrate that we can slow down benchmarks by a specific amount, which is a challenge because of the optimizations in modern CPUs, and we verified with hardware profiling that on a basic-block level, the slowdown is accurate for blocks that dominate the execution. With the benchmarks slowed down to specific speeds, we confirmed that Async-profiler, JFR, JProfiler, and YourKit maintain original performance behavior and assign the same percentage of run time to methods. Additionally, we identify cases of inaccuracy caused by missing debug information, which prevents the correct identification of the relevant source code. Finally, we tested the accuracy of sampling profilers by approximating the ground truth by the slowing down of specific basic blocks and found large differences in accuracy between the profilers.

We believe, our slowdown-based approach is the first practical methodology to assess the accuracy of sampling profilers for JIT-compiling systems and will enable further work to improve the accuracy of profilers.

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

Additional Key Words and Phrases: Profiling, Sampling, Accuracy, Ground Truth

ACM Reference Format:
Humphrey Burchell and Stefan Marr. 2025. Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy through Machine Code-Level Slowdown. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 402 (October 2025), 32 pages. https://doi.org/10.1145/3763180

1 Introduction

Many applications benefit from just-in-time (JIT) compilation, including browser-based applications, games, and long-running server systems. Once an application gains users, developers often need to optimize the performance of specific features. CPU sampling profilers are an important tool for developers to find optimization opportunities, since they identify the methods in which a program spends its time. Unfortunately, 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 [Burchell et al. 2023; Mytkowicz et al. 2010].

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

Authors' Contact Information: Humphrey Burchell, University of Kent, United Kingdom, h.burchell@kent.ac.uk; Stefan Marr, University of Kent, United Kingdom, s.marr@kent.ac.uk.

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, i.e., it induces an observer effect [Burchell et al. 2024; 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 [Burchell et al. 2023; Mytkowicz et al. 2010].

This paper proposes a methodology to assess the accuracy of sampling profilers for systems with JIT compilation. The main technique is to slow down a program accurately at the level of native code basic blocks using the feedback of CPU instruction profiling support. By comparing profiles for 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 the program's performance behavior remains unchanged.

Given the complexity of modern CPUs with cache hierarchies, out-of-order execution, pipelining, and microcode [Colwell 2021], we start our evaluation by demonstrating 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 behavior, 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.

The contributions of this paper are as follows:

- An approach to assess the accuracy of profilers by approximating the ground truth by slowing down specific basic blocks.
- A technique to accurately slow down a JIT-compiled program at the basic-block level guided by CPU instruction profiling.
- We demonstrate that programs achieve the target slowdown at the overall run time as well as the basic-block level preserving the underlying performance behavior.
- Four JVM sampling profilers, we show that they reach a good level of accuracy but fail when they cannot correctly attribute native code to source code.

2 Background

This section discusses the background on profiling, the Graal compiler, x86, and VTune as a CPU instruction profiler.

2.1 Sampling Profilers

The four JVM-based profilers we evaluated Async, JFR, YourKit, and JProfiler, use a sampling approach to periodically capture execution data. During a program's execution, at intervals set by the user or profiler, the program's execution is temporarily halted, and data is collected. These intervals are typically set to be regular to ensure an equal distribution and probability of sampling every part of the program with the same likelihood. In addition to identifying where a program spends its time, such profilers often also report call stack, CPU utilization, and memory usage. However, sampling-based profilers only collect insights at the sampled points, leaving gaps where activity might occur between intervals.

Such profilers can be subject to safepoint bias [Mytkowicz et al. 2010] and can exhibit inconsistencies between runs [Burchell et al. 2023]. The safepoint bias comes from only collecting details about the executing program when it is in a known state, i.e., at a safepoint [Agesen 1998], which limits and biases the view on the execution. However, profilers such as Async-profiler aim to avoid this safepoint bias.[1] It uses an OpenJDK internal API `AsyncGetCallTrace`,[2] to sample the call stack independent of safepoints.

Mapping from Native to Source Code. As previously discussed, sample-based profilers periodically halt execution to collect for instance the current instruction pointer. The instruction pointer is then looked up in compiler-generated debugging information to determine which method is executing. Profilers such as Async-profiler and JFR use the aforementioned `AsyncGetCallTrace` API[3] for this. When `AsyncGetCallTrace` encounters instructions without debug information, it attempts to attribute them by identifying the nearest method boundary.[4] 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 and Stinnett 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 does not directly correspond to the source code, making instruction-level source attribution more complex. Kell and Stinnett [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.

2.2 Graal Compiler

The Graal compiler is a just-in-time (JIT) compiler, a system designed to perform compilation during program run time [Duboscq et al. 2013]. JIT compilers leverage run-time information, for instance about types and method calls, but also CPU architecture and system-specific details 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.

[1] https://github.com/async-profiler/async-profiler
[2] https://github.com/async-profiler/async-profiler/blob/master/docs/GettingStarted.md
[3] See 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
[4] instruction pointer Fallback: https://github.com/openjdk/jdk/blob/0460978e7c769624cacdb528277a99914b327e30/src/jdk.hotspot.agent/share/classes/sun/jvm/hotspot/code/NMethod.java/#L287

Compilation Process. The Graal compiler uses several intermediate representations (IRs), to optimize the code and subsequently generate machine code. These stages include:

(1) **Higher Intermediate Representation (HIR):** A higher-level abstraction of the source code 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 representing operations and edges indicating dependencies [Click and Paleczny 1995; Duboscq et al. 2013]. On this HIR, the compiler applies a range of optimizations, such as loop transformations, inlining, and dead code elimination, to improve performance.

Graal LIR Blocks. After the Graal HIR stage, the code is transformed into the lower intermediate representation (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.

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

Machine Code Basic Block. 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 or branch instruction at the end of the sequence, which determines the next block to be executed [Bakhvalov 2024; Hennessy and Patterson 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 Execution (OOO) 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 experiments.

2.4 VTune, A Profiler Using Hardware Event Sampling

Intel's VTune profiler is a run-time analysis tool for systems with Intel (and some AMD) CPUs. It monitors hardware and system usage through sampling and hardware counters, and can reveal instruction-level hotspots. We use VTune's hardware event-based sampling mode, which periodically samples performance counters that track metrics such as CPU cycles or instruction executions. These counters can be programmed for specific events [Bakhvalov 2024]. However, counters are sampled rather than recorded continuously, so there is a level of granularity that may miss 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].

3 Existing Approaches and Challenges 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.

Java Profiler Accuracy and Sampling Bias. Mytkowicz et al. [2010] investigated whether multiple Java profilers would provide consistent results for the same program. They found that different profilers often disagreed, suggesting that some of them were inaccurate. Additionally, they examined whether sampling-based profilers distributed samples evenly across a program's execution. They found that was not the case, and instead profilers tended to sample execution at so-called safepoints, which are VM-specific implementation details. Thus, profiles could not represent the distribution of execution time accurately. Since 2010, modern profilers have developed techniques to avoid such *safepoint bias*. For example, Async-profiler was specifically designed to eliminate it.

Recently, we revisited the question of whether sampling Java profilers exhibited accuracy issues [Burchell et al. 2023]. We found that different profilers still provided different answers for the same program, indicating that some, if not all, were inaccurate. Worse yet, profilers produced different results across multiple runs of the same deterministic program.

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 code 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 7.3.

Challenges in Evaluating Profiler Accuracy: The Need for a Ground Truth. Evaluating the accuracy of profilers is inherently difficult because we do not know what an accurate profile for a program execution would be and obtain such a *ground truth* is a challenge itself.

While sampling profilers do not directly change the executing program, they still influence execution enough, for instance by sampling stacks, to alter how a program is compiled, and thus, induce an observer effect. This observer effect is one explanation for why we saw significantly different results across multiple runs of a profiler [Burchell et al. 2023].

Given the complexity of modern JIT compilation, the software stack, and today's CPUs, it is also not feasible to look at a program's source code and determine how it executes. Even relating source to machine code is a challenge with modern compiler optimizations. Indeed, they can make mapping back and forth not just challenging but ill-defined because of code motion, loop fission and fusion, and other transformations [Assaiante et al. 2023; Kell and Stinnett 2024].

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 behaviors 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.

4 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 behavior. We first describe the idea behind our approach, give an overview of it, and then discuss its implementation in Graal.

4.1 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 behavior 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 behavior 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, because on today's systems measuring t_i changes t_i, due to the observer effect. Instead, we approximate f with a function g such that the difference between actual and approximate time is negligible, i.e.:

$$|g(bb_i) - f(bb_i)| \leqslant \varepsilon.$$

In our case, 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:

$$g(bb_i \times 1.5) = 1.5\, g(bb_i), \qquad g(bb_i \times 2) = 2\, g(bb_i), \qquad g(bb_i \times 2.5) = 2.5\, g(bb_i).$$

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 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 behavior. We will test our assumption of a linear relation by looking for deviations and the effect slowdown has on program behavior in Section 6.2.

Using the Approximated Ground Truth to Evaluate Profilers. To evaluating 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 behavior, e.g. the exact amount of additional run time in the targe code region should be detectable. This allows us to detect profiler inaccuracies at the method level.

4.2 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 7.2.6 for a discussion). Instead, we need to measure how much time each basic block takes and then insert a suitable amount 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 2.4). 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 1 illustrates the steps of this process.

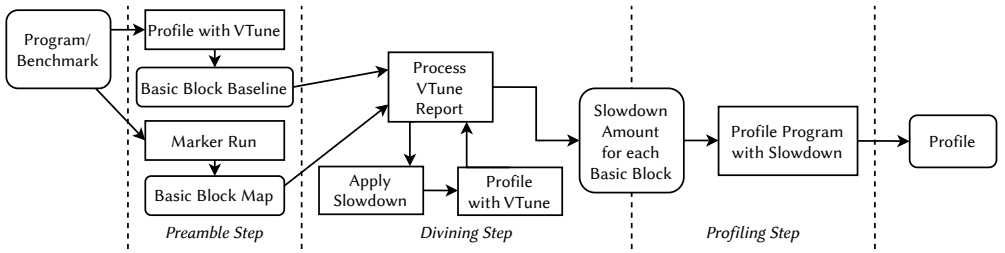


Fig. 1. Illustration of the steps to slow down a program. First, in the preamble step, we identify how the program compiles to native code and how much time basic blocks take without slowdown. In the divining step, we find the right amount of slowdown for each basic block. This slowdown information can 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.

4.3 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 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 instructions that encode Graal LIR block IDs into the native code, as detailed in Section 4.4.1. 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.[5] We parse the assembly text output VTune

[5]We 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.

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 contributes to the total run time. Blocks that contribute fewer than 0.001 seconds are excluded from further processing.[6]

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 value of 1 to 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.[7]

For blocks that do not exceeds 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 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 behaviors between basic blocks of the same compilation unit are not changed.[8]

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.

4.4 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 behavior 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.

[6]Similar 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.

[7]The 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 7.2.3.

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

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 4.3). 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 nontrivial 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 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, 0xfffffffcd
4 jmp <Block 36>
```

(a) Original Machine Code Block

```
1 Block 34:
2 vpblendd xmm0, xmm0, xmm0, 0x2e
3 vpblendd xmm0, xmm0, xmm0, 0x0
4 mov r11, 0x1914
5 mov r10d, 0xfffffffcd
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, 0xfffffffcd
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

Fig. 2. Three versions of a Machine Code Block: (a) the original unmodified block, (b) the version with marker instructions, (c) the slowed-down version.

4.4.1 Marker Phase. The marker phase is only used in the preamble step (see Section 4.3) 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 bits 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 2b illustrates a basic block that was modified by a marking step. Compared to the original block in Figure 2a, Figure 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.

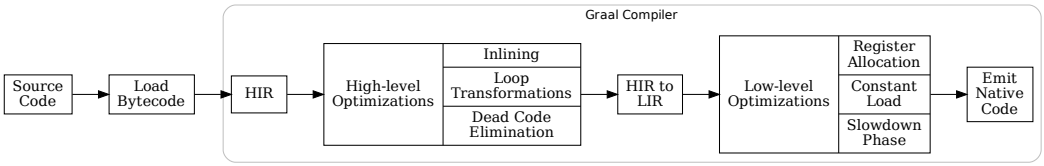


Fig. 3. Sketch of the Graal compiler pipeline, going from source code to emitting native code. Our slowdown phases is one of the last low-level optimizations, after all high-level optimizations, such as inlining and loop transformations happened, and thus, does not impact them.

4.4.2 Slowdown Phase. The slowdown phase is used by the divining and profiling steps to insert a specified number of instructions that slowdown a basic block, without otherwise changing the block's behavior. 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.

This phase is inserted late in the lower-level optimizations; after register allocations and constants loads, as illustrated in Figure 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 behavioral changes. The slowdown phase loads the JSON file and reads the per-block information for the current compilation unit. For each basic block for that 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 `NOP`s to be the only other viable option [Burchell and Marr 2025b].

Figure 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, and assume it works on others. 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 instruction have no debugging information. As alluded to in Section 2.1, 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 would alter 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. However, this approach is not without its limitations, as discussed in Section 7.1.4.

5 Evaluation Methodology

We evaluate our approach to approximating a ground truth profile with three research questions. The goal is confirm its suitability and investigate the accuracy of four JVM sampling profilers. First, we outline the research questions and then the experimental setup.

5.1 Research Questions

The first two questions assess whether our slowdown-based approach works as intended. 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 behavior of the program. For this, we expect that our approach does not influence major compiler optimizations and minimizes impact on the performance behavior at the microarchitecture level. Thus, sampling-based profiler's 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 used microarchitecture.

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 4.1. When experimenting with slowdowns, these gave good results, avoiding too small slowdowns that cannot be created reliably with a `mov` instruction, and spaced out enough to see a possible impact, for instance on the microarchitectural performance behavior.

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 behavior, 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. We disabled dynamic frequency scaling and maximized the process priority to minimize measurement noise.

Selected Benchmarks. As benchmarks we chose the Are We Fast Yet benchmarks [Marr et al. 2016], because they are fully deterministic Java programs, representing what we believe to be the best case for Java profilers. The suite includes 5 macro-benchmarks and 9 micro-benchmarks, which made it feasible for us to investigate any compilation issues and ensure that slowdown is applied accurately. While larger benchmark suites such as DaCapo [Blackburn et al. 2025] and Renaissance [Prokopec et al. 2019] would be preferable, due to their size and complexity, the additional engineering effort needed to investigate any issues was beyond what we were able to do.

To achieve practical run times with low noise levels, we configured the benchmarks such that a single iteration takes approximately 100ms.

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. The profilers are:

- **Async-profiler** v3.0:[9] An open-source project aimed at providing low-overhead sampling without safepoint bias.
- **Java Flight Recorder (JFR)**, OpenJDK 21.0.2:[10] Part of the OpenJDK, it can collect run-time statistics and supports sampling-based profiling.
- **JProfiler** 14.0.5:[11] A commercial profiler, advertised as a comprehensive all-in one tool.
- **YourKit** 2024.9:[12] A commercial profiler, advertised as a fully featured low overhead tool.

Hardware and Software Configuration. All experiments were performed on a machine with a 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.[13]

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 files 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, diving, 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.

[9] https://github.com/async-profiler/async-profiler/releases/tag/v3.0
[10] https://openjdk.org/jeps/328
[11] https://www.ej-technologies.com/jprofiler
[12] https://www.yourkit.com/
[13] https://github.com/oracle/graal/commit/049d6d3ab565c74549a590a4b744077a45f7527e

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.

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 to source code and correctly attribute run time to methods in a profile, we set the `-XX:+DebugNonSafepoints` for all profilers. This insures 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 7.1.1 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 behavior and enables comparisons with a consistent baseline.

6 Results

This section presents our experiments and answer the research questions of Section 5.1.

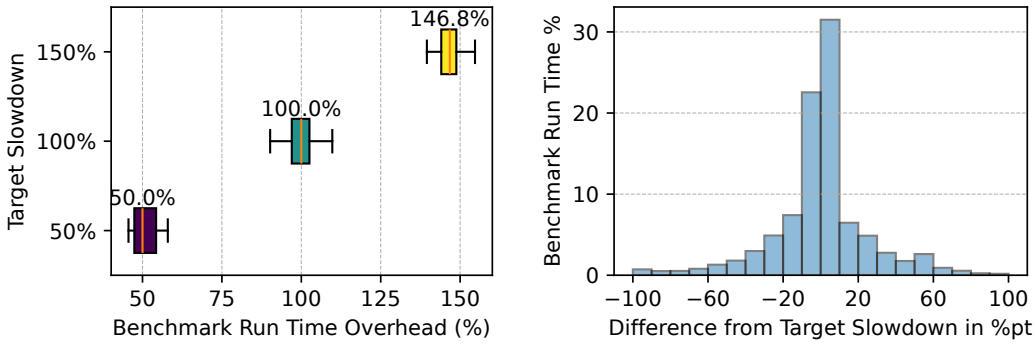
6.1 Experiment 1: The Effect of Slowdown on Program and Basic Block Run Time

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.

As Figure 4a shows, we can successfully slowdown 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. While the median run time overhead across all benchmarks is 50.0%, 100.0%, and 146.8%, respectively. Table 4 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 4b 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 to few 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



(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.

Fig. 4. 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 behavior effectively.

contributions to a benchmark's run time. Figures 7 and 8 in the appendix show the histograms for the 50% and 150% target slowdowns.

Overall, we can successfully and proportionally slowdown programs as intended and answer RQ1 positively. Especially 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 3, previous work inserted slowdown at the bytecode level. For completeness, Section 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.

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 slowdown 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 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.

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.

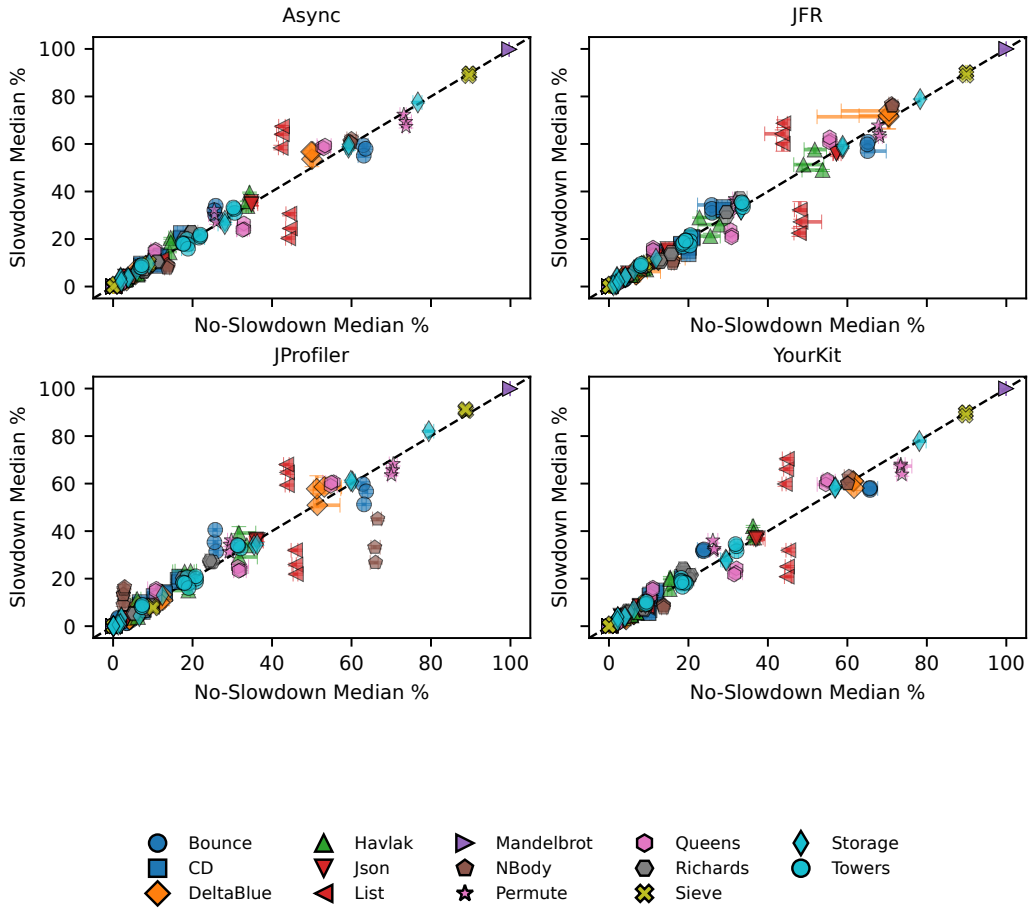


Fig. 5. A scatter plot for each profiler with the median percentage of run time for each of the top five methods in each of the 14 benchmarks. The diagonal $X = Y$ line indicates that a method's run-time percentage remains the same under slowdown and no-slowdown. The plot shows results for 50%, 100%, and 150% slowdown.

Table 1. Correlation and deviation statistics for each profiler. If there is no change between normal and slowed-down runs, 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

Summary Statistics. To summarize Figure 5 and answer RQ2, we look at the correlation coefficient between the report run times for a method, and analyze how strong the deviation is from the expected run time in Table 1. 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 behavior. 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 3, profilers can have sampling bias. In Section 7.2.7, 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 4.4.2, 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 2 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 4.4, a single LIR instruction may emit multiple machine code instructions.

Table 2. 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 2, the List benchmark has the lowest ratio of debugging information to total LIR instructions. This could explain why profilers misattribute run time. We further investigate this in experiment 3 by assigning debugging information to slowdowns.

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 inserted 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 attributed to the correct method.

As seen in Section 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 2.1, profilers could simply assume such an instructions 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.

6.3.1 *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 with valid debugging information, which means from the profiler's and compiler's perspective 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 most time. The goal was to select blocks that are important for the benchmark's execution, and do 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 behavior, 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 5.1.

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 5.

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% percent 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 analysis going forward.

6.3.2 *Results.*

Overall Results. Table 3 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.

Table 3. 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

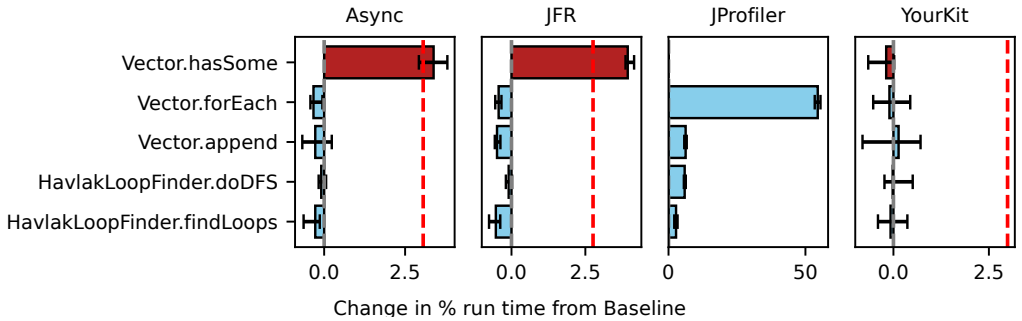


Fig. 6. Change in the percentage of run time with 5 seconds of slowdown added to `Vector.hasSome`. The baseline without slowdown is at 0. The red dashed line indicates the expected change in % of run time. The target method is shown in red.

Example with Mixed Results. To illustrate the experiment, Figure 6 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 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 the triggering of the sample and recording it. If this skid introduces bias, sampling results may misrepresent the distribution of execution time and result in inaccurate profiles.[14]

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.

7 Discussion

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

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.

7.1.1 Tradeoffs of DebugNonSafepoints. In our experiments, we found that setting the DebugNonSafepoints flag (see Section 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 6.3) and 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.[15] 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.

7.1.2 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 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 many 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

[14] https://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html
[15] 8307057 JFR: Enable -XX:+DebugNonSafepoints when JFR starts https://github.com/openjdk/jdk/pull/14147

that they use different techniques to access debugging information or heuristics to determine the corresponding method.

7.1.3 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 Java stepping debugger can rely on deoptimization to obtain the needed information, a profiler however will want to use the have 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 merge 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 behavior. 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.

7.1.4 Debugging Information for Slowdown Instructions. As mentioned above, because of the various compiler optimizations, 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 experiment 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 of the target method to slowdown instructions and inserted them contiguously, i.e., without interleaving 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 7.2.3. 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.

7.2 Hardware- and JVM-Related Factors Affecting Accuracy

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

7.2.1 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.[16] 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].

7.2.2 Microarchitecture-Specific Slowdown. As outlined in Section 4, 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.

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.

7.2.3 Accuracy of Sampling Hardware Counters. Our basic-block-level slowdowns relies 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 to 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.

7.2.4 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 my 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

[16] The 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

the micro-architecture can optimize, might let us apply a smaller slowdown. A systematic approach could use throughput prediction [Abel and Reineke 2022] to select instruction sequences that increase block latency while preserving semantics.

7.2.5 Run-Time Overhead of VTune. We also examined the overhead introduced by VTune during the divining process. There may be concerns that if the overhead of VTune 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 these overhead per benchmark can be found in Figure 9 in the appendix.

7.2.6 Using a Static Performance Model Instead of an Instruction Profiler. Instead of relying on sampling hardware counter, 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 and Lipasti 2005], which have a significant impact on performance. Consequently, we rely on sampling of hardware counters for are a more accurate reflection of execution time of each block.

7.2.7 Sampling and Safepoint Bias. One of the key observations by 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 behavior.

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 10 in the appendix. We plotted the absolute change in the percentage of run time attributed to each method. This means, 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.

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

As discussed in Section 3, 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 anymore. 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 where 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 additionally 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 as the final steps in the compilation (see Figure 3) is more reliable.

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.

7.5 Overall Divining Time and Future Optimization Opportunities

The overall divining process, as detailed in Section 4.2, is currently fairly time consuming. Determining how much slowdown to add to slowdown 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 6 in the appendix provides a detailed break down 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 and Berger 2015] or race detection [Endo and Møller 2025] based on inserting slowdown instructions.

7.6 Experimental Time and Effort

In addition to the diving itself, running all benchmarks for our research questions took about 139h, i.e., 5.79 days. Table 7 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, 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.

8 Related Work

Related work on the accuracy of JVM profilers was discussed in Section 3. Here, we discuss the wider related work.

The Coz profiler [Curtsinger and Berger 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 *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 method profiler within Graal [Burchell et al. 2024]. 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.

9 Conclusion & Future Work

This paper 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 the Graal compiler, a just-in-time compiler for Java, we insert slowdown at in the lower intermediate representation, which prevents interference with compiler optimizations and minimizes impact on the performance behavior at the microarchitectural level.

Our experiments demonstrate that the approach allows us to slowdown 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 to obtain 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 having 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.

Future Work. In future work, we want to explore how the approach transfers to different microarchitectures to make it portable. Currently, we have only tested it with Intel's Skylake. Though, we are confident that the idea applies more generally, but may need refinement, for instance adapting the instruction used for slowdown, heuristics to prevent pipeline stalling, and possibly other aspects that may change the performance behavior of a slowed-down program.

We also intend to investigate how to compensate for the inaccuracies for sampling hardware counters. If we can compensate for instance for instruction skid, we may be able to introduce slowdown instructions more targeted inside basic blocks. At the moment, we insert instructions evenly throughout, but when instructions in the same basic block come from multiple different methods, we would want to slow down instructions from only one of the methods. This could facilitate assessing the accuracy of JVM profiling not only at the method level, but down to the source expression or bytecode level. This would be important, because when investigating performance optimizations, pinpointing which part of a method takes the most time is typically the next step after identifying a method with optimization potential.

Additionally, with our approach to slowing down programs at the basic-block level, one could investigate the notion of *virtual speedup* [Curtsinger and Berger 2015] to predict the benefit of optimizing a specific bit of code.

10 Data-Availability Statement

The artifact for this paper includes the raw data of all experiments, measurements, slowdown files, as well as the source code of benchmarks, our modified version of Graal and the scripts to process the data and generate all figures, tables, and statistics included in the paper [Burchell and Marr 2025a]. However, because the slowdown is microarchitecture-specific we cannot provide an artifact that is guaranteed to reproduce our results out of the box by rerunning the preamble, divining, and profiling steps of our approach.

Acknowledgments

We thank YourKit and JProfiler for granting us academic licenses for their products.

References

Andreas Abel and Jan Reineke. 2022. uiCA: accurate throughput prediction of basic blocks on recent intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. ACM, Article 33, 14 pages. doi:10.1145/3524059.3532396

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

Cristian Assaiante, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. ACM, 935–947. doi:10.1145/3575693.3575720

Denis Bakhvalov. 2024. *Performance Analysis and Tuning on Modern CPUs, Second Edition.* Independently published. 237 pages.

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. doi:10.1145/3133876

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

Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. ACM, 940–954. doi:10.1145/3669940.3707217

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

Humphrey Burchell, Octave Larose, and Stefan Marr. 2024. Towards Realistic Results for Instrumentation-Based Profilers for JIT-Compiled Systems. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Vienna, Austria) *(MPLR 2024)*. ACM, 82–89. doi:10.1145/3679007.3685058

Humphrey Burchell and Stefan Marr. 2025a. *Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown (Artifact).* doi:10.5281/zenodo.16911348

Humphrey Burchell and Stefan Marr. 2025b. 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* (Singapore) *(VMIL'25)*. ACM, 8. doi:10.1145/3759548.3763374

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

Robert P. Colwell. 2021. The Origin of Intel's Micro-Ops. *IEEE Micro* 41, 6 (Nov. 2021), 37–41. doi:10.1109/MM.2021.3112026

Charlie Curtsinger and Emery D. Berger. 2015. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, 184–197. doi:10.1145/2815400.2815409

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

Andre Takeshi Endo and Anders Møller. 2025. Event Race Detection for Node.js Using Delay Injections. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (LIPIcs, Vol. 333)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 9:1–9:28. doi:10.4230/LIPICS.ECOOP.2025.9

Agner Fog. 2022. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* Technical Report. https://www.agner.org/optimize/instruction_tables.pdf

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

Brendan Gregg. 2013. *Systems Performance: Enterprise and the Cloud.* Prentice Hall, Upper Saddle River, NJ.

John L. Hennessy and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann. 936 pages.

Intel 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2C, Instruction Set Reference, V-Z.* Intel. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2c-manual.pdf

Stephen Kell and J. Ryan 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, 38–53. doi:10.1145/3689492.3690047

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

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

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

Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, 31–47. doi:10.1145/3314221.3314637

John Paul Shen and Mikko H. Lipasti. 2005. *Modern Processor Design: Fundamentals of Superscalar Processors.* Waveland Press. 642 pages.

R. M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* 11, 1 (Jan. 1967), 25–33. doi:10.1147/rd.111.0025

Hao Xu, Qingsen Wang, Shuang Song, Lizy John, and Xu Liu. 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, 284–295. doi:10.1145/3330345.3330371

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

Table 4. Overhead percentages for benchmarks at different slowdown speeds, this are the data points that make up Figure 4a

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%

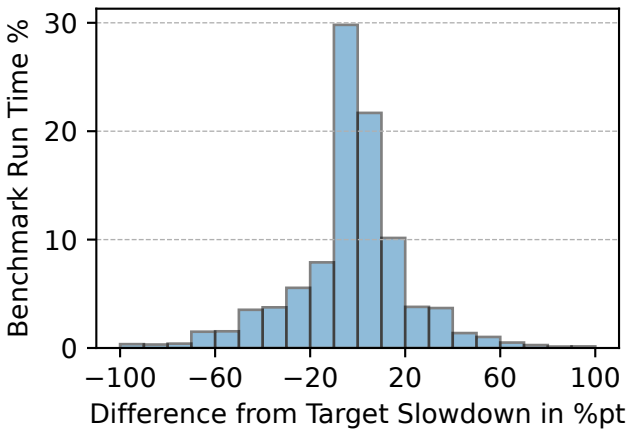


Fig. 7. Histogram shows the distribution of overhead differences from the target slowdown of 50% across all benchmarks. The x-axis represents the difference in percentage points. The y-axis shows the percentage of benchmark run time. The results show that the majority of run time is spent in blocks with overheads close to the target slowdown.

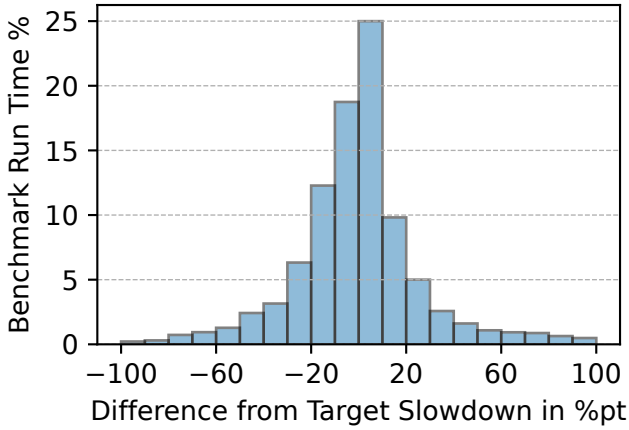


Fig. 8. Histogram shows the distribution of overhead differences from the target slowdown of 150% across all benchmarks. The x-axis represents the difference in percentage points. The y-axis shows the percentage of benchmark run time. The results show that the majority of run time is spent in blocks with overheads close to the target slowdown.

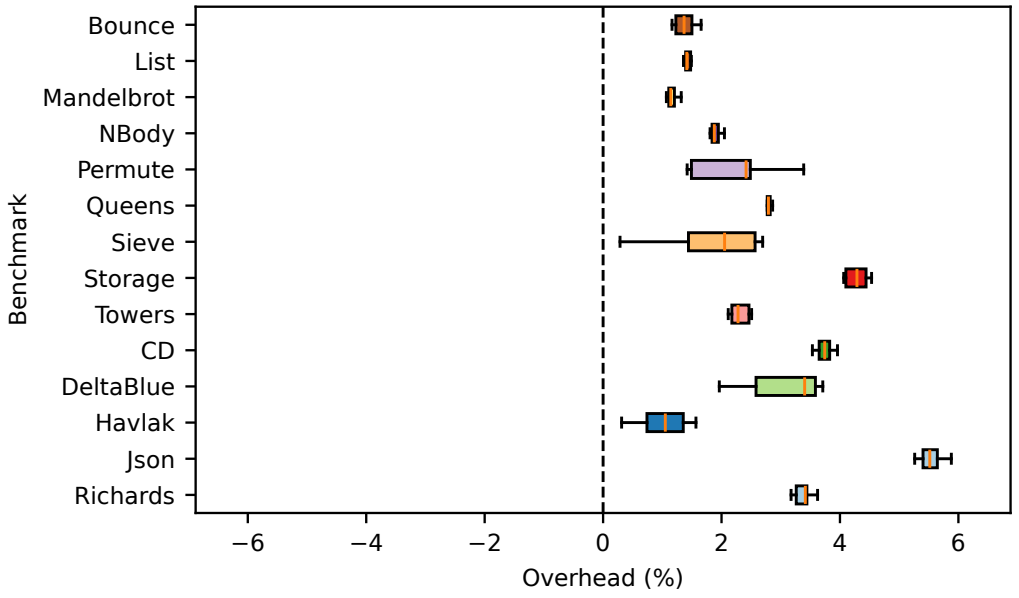


Fig. 9. Displays the overhead measured across 10 runs for each benchmark when using VTune in hardware sampling mode with stack collection. The baseline represents the median runtime with no profiling tools attached.

Table 5. 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 for every benchmark and profiler the values are a 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 6. Divine 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 2 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

Table 7. 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**

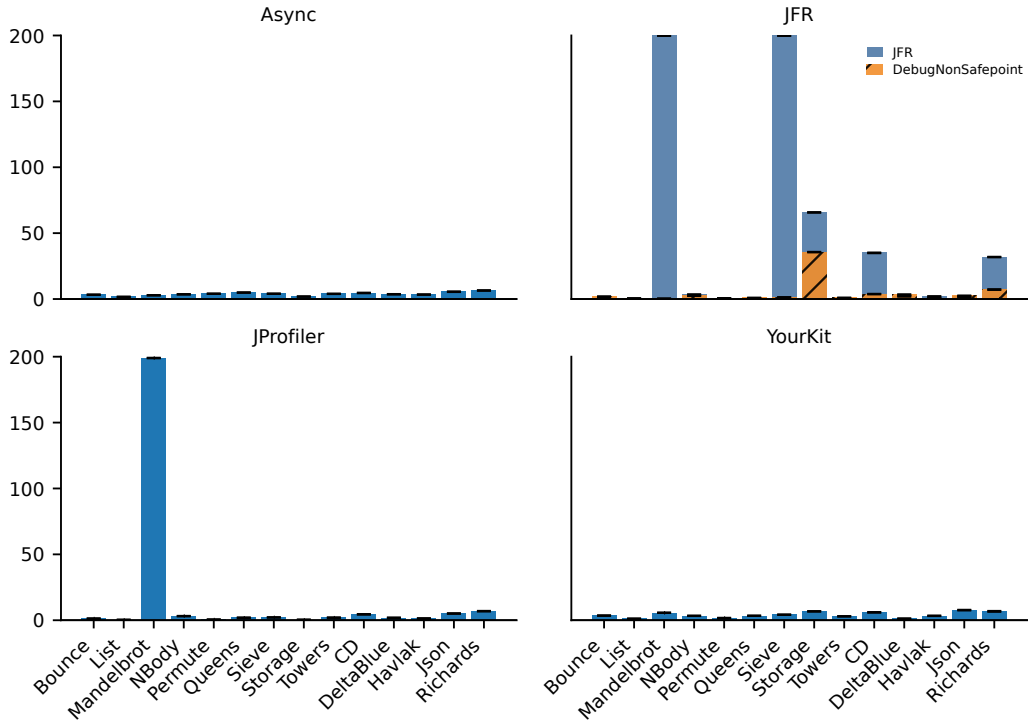


Fig. 10. Bar chart showing the absolute change in percentage points for all methods for each benchmark, when transitioning from safepoints being present to removed. The purpose is to demonstrate how profilers adjust their results when no safepoints are available in the program. JFR includes two versions, illustrating the difference in results when the -XX:+DebugNonSafepoints flag is enabled.