Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level

Towards Supporting Fine-Grained Slowdown for Advanced Developer Tooling

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

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

Abstract

Slowing down programs has surprisingly many use cases: it helps finding race conditions, enables speedup estimation, and allows us to assess a profiler's accuracy. Yet, slowing down a program is complicated because today's CPUs and runtime systems can optimize execution on the fly, making it challenging to preserve a program's performance behavior to avoid introducing bias.

We evaluate six x86 instruction candidates for controlled and fine-grained slowdown including `NOP`, `MOV`, and `PAUSE`. We tested each candidate's ability to achieve an overhead of 100%, to maintain the profiler-observable performance behavior, and whether slowdown placement within basic blocks influences results. On an Intel Core i5-10600, our experiments suggest that only `NOP` and `MOV` instructions are suitable. We believe these experiments can guide future research on advanced developer tooling that utilizes fine-granular slowdown at the machine-code level.

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

Keywords: slowdown, x86 instructions, evaluation

ACM Reference Format:
Humphrey Burchell and Stefan Marr. 2025. Evaluating Candidate Instructions for Reliable Program Slowdown at the Compiler Level: Towards Supporting Fine-Grained Slowdown for Advanced Developer Tooling. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3759548.3763374

1 Introduction

Against the trend of focusing on optimizations, a number of developer tools work by slowing down programs. One is race detection [8], where delays are induced to explore different schedules to uncover race conditions. Another use case is *virtual speedup* [6], which is used to assess the performance potential that optimizing specific program parts yields. Finally, slowing down program parts can be used to evaluate the sensitivity and accuracy of profilers [4].

To enable these applications, any slowdown must be targetable to specific code sections, without slowing down others, e.g., as a result of out-of-order execution. Furthermore, we need to be able to adjust the amount of slowdown for a section precisely and with fine granularity. To this end, we explore six x86 instruction candidates to slow down programs predictably by inserting them late in the compilation.

We inject the slowdown instructions into machine-code basic blocks and use hardware counters to determine how many are needed. We implemented our experiments in the Graal compiler, a just-in-time (JIT) compiler and evaluate the candidates on a Java benchmark with 53 basic blocks. The experiments were run on an Intel Core i5-10600 CPU with the Comet Lake-S microarchitecture. As an experiment, we will try to precisely double the run time of a benchmark at the level of basic blocks. This 100% overhead is high enough to be measurable, allowing for a range of slowdown candidates, without being impractically slow or immeasurably small.

We found that `NOP` and `MOV` give us most control and the finest granularity. When aiming for a 100% slowdown, they allow us to achieve a very close 109% and 106% program slowdown, and thus, are the best choice for accurately slowing down programs. We also tested a `PUSH+POP` sequences, which caused 202% overhead, `SFENCE` (405% overhead), a mix of normal and vector instructions (named `Long PUSH+POP`, 410% overhead), and `PAUSE` (11,505% overhead). Though, they are not fine-grained enough for our use cases.

The contributions of this paper are as follows:

- An evaluation of the effectiveness of six slowdown candidates, with `MOV` and `NOP` being most suitable.
- An evaluation of the placement of slowdown, demonstrating that interleaving existing and slowdown instructions better maintains the performance behavior.

VMIL '25, Singapore, Singapore

2 Background

This section gives a brief overview of relevant x86 instructions, microarchitectural optimizations, and VTune.

Slowdown Instructions. Adding deliberate delays into a program requires instructions that consume cycles without changing its logical state. The x86-64 instruction set comes with no-operation (`NOP`) instructions of different length, a `PAUSE` instruction (a hint for spin-wait loops), memory fences, and it allows us to combine instructions into sequences that execute without observable effect. From these, we select five candidates in Section 5 with possibly different performance behaviors, to find the most suitable one.

Pipelining and Out-of-Order Execution. Modern processors increase instruction throughput and performance with *pipelining* and *out-of-order execution* (OOO) [9]. The dynamic reordered and parallel execution enabled by these approaches 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.

Hardware Event Sampling with VTune. Intel's VTune profiler is a tool that samples performance counters of hardware events that track metrics such as CPU cycles and executed instructions. Based on this data, it can determine how much time is spent in a given basic block. However, because counters are sampled, it may miss some events. Furthermore, because sampling is subject to *skid*, i.e., a delay between triggering and recording the counter, results can be misattributed [15] limiting the accuracy of metrics.

3 Use Cases, Requirements, and Candidates for Machine-code-level Slowdown

This section discusses use cases that benefit from slowing down programs in more detail, derive requirements for our approach, and select candidate instructions.

3.1 Use Cases

Existing approaches that slowdown programs use coarse-grained techniques, e.g., inserting delays based on the runtime or kernel mechanisms, blocking thread execution, or inserting code at the bytecode level. We propose to insert slowdowns at the compiler level, after optimizations [2] to have full control without biasing performance behavior.

Exposing Race Conditions. A number of approaches propose to detect race conditions by changing the timing of interactions. Musuvathi et al. [12] adapted thread scheduling to find concurrency-based heisenbugs in shared memory concurrency. Stoica et al. [14] insert sleep operations to detect memory-ordering bugs. Endo and Møller [8] propose an approach to explore different event schedules and thereby expose race conditions by instrumenting Node.js and injecting random delays in the event execution. All of these approachs are coarse-grained and change the high-level scheduling.

When inserting slowdowns at the basic-block level, one could use a similar approach to identify race conditions at the instruction level. By inserting slowdown instructions into critical sections, we can widen the vulnerability window enough for other threads to overtake and expose race conditions, while leaving the rest of the code to run at full speed. Because the delay can be incrementally tuned, preserves register state, and does not alter memory layout or optimization decisions, it could give a very fine granularity.

Virtually Speeding Up Code. Virtual speedups can be used to estimate how much benefit an optimization yields [6]. Curtsinger and Berger [6] slow down other code by pausing the other threads. Thus, the target code is *virtually* speed up, by not being slowed down. Though, because they do this on the level of threads, it has a fairly coarse granularity.

Inserting slowdown at a basic-block level could give the same benefit and allow developers to judge the impact of optimizations that may not be visible at the thread level.

Accuracy Testing of Profilers. Mytkowicz et al. [13] and Burchell et al. [3] showed that Java profilers are unreliable. There are some inherent issues such as the JVMs non-determinism, as well as the profiler's safepoint bias. Recently, we showed how slowing down code can be used to estimate a ground truth to assess profiler accuracy [4]. By inserting slowdown after all compiler optimizations, we can determine the degree by which profilers are inaccurate. Though, we simply used `MOV` and investigate here, which the most suitable candidate for this approach is.

3.2 Requirements

For the above uses cases, we will need an approach that allows us to inject slowdown accurately, without side-effects, i.e., without changing the program semantics, and in a way that allows us to distribute slowdown freely inside basic blocks. This is needed to slowdown a program in a controlled and predictable way, while keeping basic block performance behavior intact. With *performance behavior* we mean how time is distributed across program parts for a single execution. It includes where time is spent, i.e., in which methods or basic blocks, and how much time is consumed.

Accurate Amount of Slowdown. To accurately predict speedups and assess profiler measurements, a specific basic block needs to be slowed down proportionally to its original run time. For instance, the block's run time may need to be doubled, as in the experiments reported here. Then, we can assess whether profilers report this slowdown accurately. Similarly, if we want to estimate the impact of speeding up a method by 10%, we would need to make all other methods 10% slower for a virtual speedup. This means, we need to be able to insert precise amounts of slowdown into a basic

block. It also means the slowdown must apply to only the target basic block, without affecting other blocks. If a slowdown intended for one block, e.g. in `methodA`, inadvertently affects another part of the program, e.g., a block belonging to `methodB`, the performance behavior becomes unreliable. Thus, if for any reason the slowdown is only approximate, it would limit the precision available to our use cases.

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

Distributed Slowdown. For virtual speedup and determining profiler accuracy, slowdown needs to be evenly distributed across a basic block. If it is concentrated in few spots, we may reach the target slowdown accuracy, but it may still bias observable performance behavior. With inlining, a basic block can have instructions from different source methods. Thus, for tools that work at source level, slowdown may need to be attributed to specific instructions within a basic block. Though, with the limited precision of hardware counter sampling, we will only aim for evenly distributing slowdown in a basic block to limit biasing the performance behavior from the source-level perspective too much.

3.3 Candidate Instructions

As discussed in Section 3.2, a slowdown instruction or instruction sequence cannot affect the program's underlying logic, must allow for accurate and deterministic slowdown, and we must be able to interleave it with other instructions. Based on these requirements, we selected the six candidates shown in Table 2 in the appendix. They are likely to consume CPU cycles without altering the surrounding program logic or control flow, and thus, satisfy our requirements.

We use a 2-byte instead of a 1-byte `NOP` because the latter seemed be removed by Graal. Since some Intel CPUs optimize `NOP`s early in their pipelines [10, Section 2.1.2.1], we also included the `MOV`, and the `PUSH+POP` instruction sequence for variety. `SFENCE` and `PAUSE` were included since they are likely to consume CPU cycles, without further side effects. Finally, we included the Long PUSH+POP sequence to see whether using a different functional unit, e.g., the vector unit, makes a difference.

4 Inserting Slowdown at the Compiler Level

We implemented our slowdown mechanism as a compilation phase in the Graal just-in-time (JIT) compiler, which can compile Java programs at run time [7]. Java bytecode is first turned into GraalIR [7], which is used for high-level optimizations such as inlining and loop transformations. Afterwards, the GraalIR is turned into the lower intermediate representation (LIR), in which the code is organized as basic blocks, similar to machine code basic blocks. The LIR is used for register allocation and a few other low-level optimizations.

4.1 The LIR Slowdown Phase

We added our slowdown mechanism as a LIR compilation phase. This phase inserts a specified number of LIR instructions, which then emit the desired x86 instructions. The phase takes a *slowdown file* as input, which lists for each compilation units and each LIR basic block the amount of slowdown to be injected. This allows us to add a precise amount of slowdown per basic block.

We insert slowdown instructions interleaved with the existing LIR instructions within each block, to distribute the slowdown and avoid biasing the performance behavior. Furthermore, we do not assign debugging information to the slowdown instructions. This avoids biasing tools towards a specific instruction, when they use the debugging information. As mentioned before, this is needed because basic blocks can contain instructions from multiple methods. Section 6 will evaluate the effectiveness of interleaving instructions.

4.2 Using Hardware Counters to Determine Slowdown Amount

Since some use cases require us to add slowdown proportional to the run time of a basic block, we need the accurate run time of basic blocks considering out-of-order execution, pipelining, caching, and memory access latency. To obtain it, we use the hardware event counters using VTune, but other tools, for instance `perf`, would be suitable, as long as they measure the run time of a basic block including all the effects mentioned above. As simplification, we consider each block without context, i.e., previous control flow.

To determine the needed slowdown for a block, we implemented a feedback-based system that measures the run time of a block, adds more slowdown instructions, and executes the program again, measures the new run time of the block, and repeats this process until the target slowdown is achieved. This iterative approach achieves accurate results and maintains a block's initial performance behavior.[1]

However, this iterative approach takes a long time. Determining the required slowdown for the Towers benchmark takes approximately two hours, even though the benchmark itself only has a 100 second execution time.

Furthermore, because of the sampling and granularity of hardware counters, the timing information for specific instructions is not generally reliable. Therefore, we apply

[1]We call this system the *GroundTruth Scheduler*, and use it to assess profiler accuracy [4]. The paper describes it in more detail.

Table 1. Number of instructions needed to at least double each basic block's run time in the *Towers* benchmark.

Candidate Sequence	Number of Times Inserted
NOP	303
MOV	304
PUSH+POP	72
SFENCE	55
Long PUSH+POP	63
PAUSE	55

slowdown to blocks rather than instructions. While this introduces inaccuracies, for instance when a block contains instructions from multiple methods, and instructions with different costs, Section 5 shows that our approach is sufficient to accurately slow down both individual blocks and the entire program to the desired target overhead.

5 Evaluating Candidates

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

5.1 Methodology

For the evaluation, we use the Towers of Hanoi benchmark from the Are We Fast Yet suite [11]. The Java version JIT-compiles to 53 basic blocks within a single compilation unit, and has a suitable mix of larger and smaller basic blocks. Since it is fully deterministic, it simplifies experiments.

We only use a single benchmark, because to determine how much slowdown to add to all 14 benchmarks takes 3.75 days only for the MOV instruction. While using all 14 benchmarks for all candidates would have been desirable, the MOV instruction gives comparable results for all benchmarks for Are We Fast Yet [4]. We run Tower for 500 iterations and it is fully JIT-compiled after only a few. We rerun the experiment 10 times to determine the achieved overhead.

5.2 Required Number of Slowdown Instructions

For each candidate, we determine how often it needs to be inserted into a basic block to double the block's run time, i.e., reach the 100% target slowdown. Table 1 shows the results. In total, over all 53 basic blocks, we needed to insert 303 NOP or 304 MOV instructions to achieve the slowdown. In contrast, we only needed 55 PAUSE instructions. This already hints at NOP and MOV having the finest granularity, while the other candidates are much coarser, and thus, give less flexibility.

5.3 Run-time Overhead

Figure 1 shows a boxplot of the run-time overhead observed when running the Towers benchmark for each candidate.

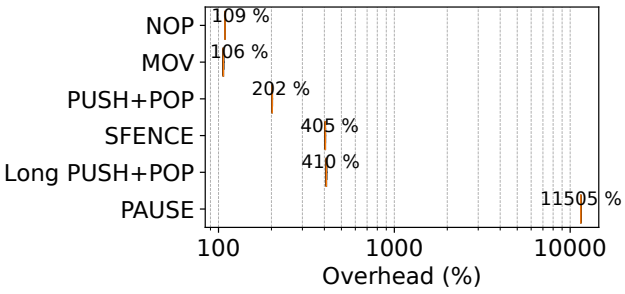

Figure 1. Boxplot with the overall program's run-time overhead achieved with each candidate for a target overhead of 100%. The x-axis uses log-scale. As baseline, we use a benchmark run without adding any slowdown.

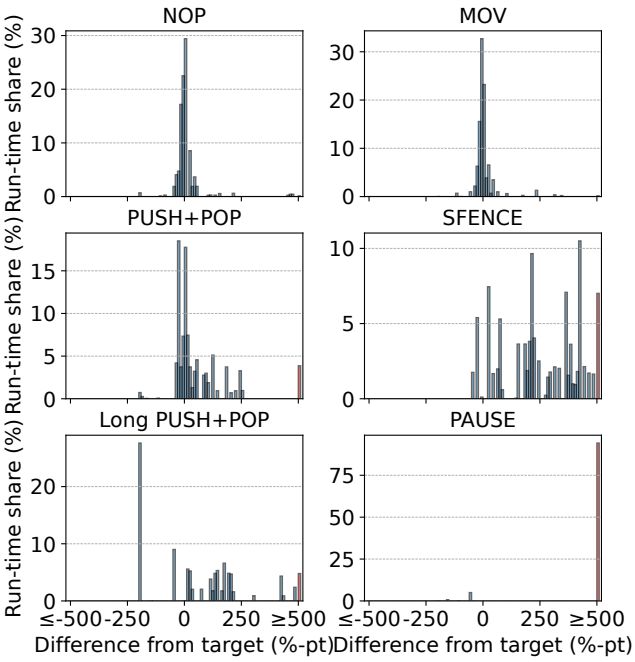

Figure 2. The histogram shows the difference in percentage points from the 100% target slowdown for all blocks on the x-axis. The % of benchmark run time is on the y-axis.

Only the NOP and MOV instructions achieve a run-time overhead close to the desired target of 100%. PUSH+POP causes an overhead of 202%. SFENCE and long PUSH+POP overshoot the target even further with an overhead of about 400%. The highers overhead is caused by PAUSE with 11,504%.

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

would concentrate the slowdown in one part of the block, instead of distributing it throughout.

5.4 Distribution of Slowdown Among Blocks

To determine whether the slowdown is accurately applied to each basic block, we plot their VTune-reported run time in Figure 2 as a histogram. This shows how close each block's slowdown comes to the target of 100%. For each percentage-point difference, the plot shows how much percent of run time the blocks contribute to the overall run time. Ideally, all blocks would fall into the 0-bucket in the middle, which would mean 100% of the run time is covered by basic blocks that reached the target slowdown. Though, since a block may take virtually no time to execute, adding even a single instruction may slow it down by more than the 100% target.

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

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

5.5 Performance Behavior with Slowdown

To evaluate the effect of slowdown on the performance behavior of a program, i.e., how time is distributed across program parts, we compare the profile of normal with the slowed down runs. Specifically, we use async-profiler, a sampling profiler for Java that is designed to avoid safepoint bias.[2]

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

Figure 3 shows normal and slowed down runs on a scatter plot where the top six methods in Towers are shown. The x-axis shows the median run-time percentage of 10 normal runs and the y-axis shows the median of 10 slowed down runs. Ideally, methods are on the x=y line, which means their percentage of run time remains unchanged. Based on the plot, `NOP` and `MOV` give the best results, and the methods' run-time percentages are close to the x=y line, staying roughly the same despite being slowed down. In contrast, Long `PUSH+POP` and the other candidates show major difference in the profile. For instance, for `PUSH+POP`, the `moveDisk` method goes from about 20% to around 40% of the program's run time.

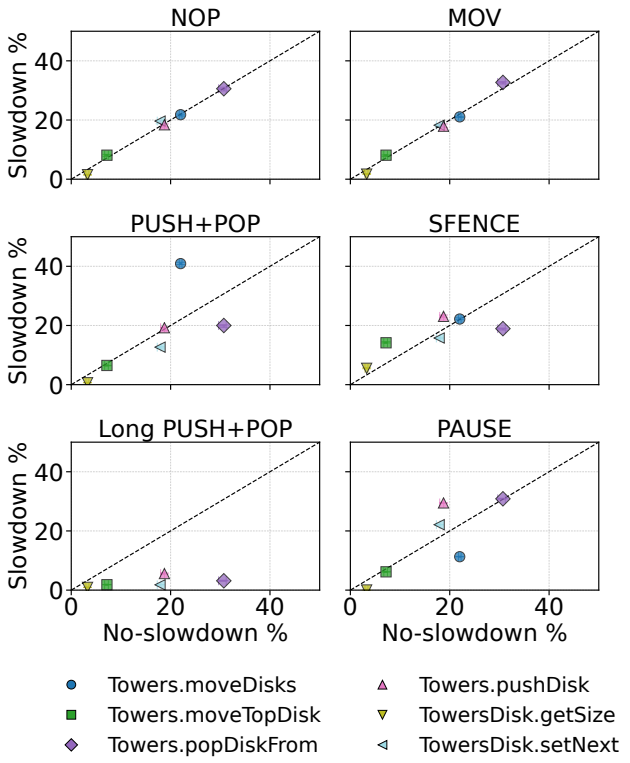

Figure 3. A scatter plot per slowdown instruction with the median run-time percentage for the top six Java methods. The $X = Y$ diagonal indicates that a method's run-time percentage remains the same with and without slowdown.

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

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

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

[2] https://github.com/async-profiler/async-profiler

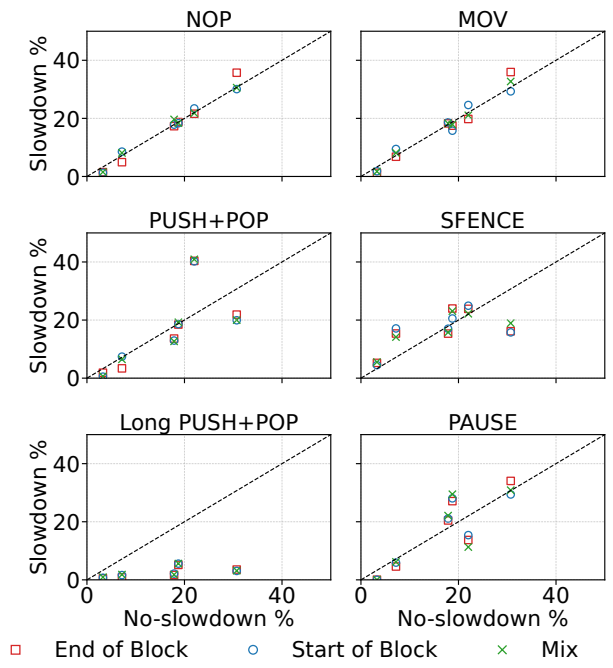

Figure 4. A scatter plot showing the impact of placing slowdown at the start, mixed throughout, or at the end of a basic block. We show the median percentage of run time for the top six Java methods of Towers. For a method on the $X = Y$ diagonal, the run-time percentage remained unchanged.

However, NOP and MOV may be optimized by CPUs [10, Section 2.1.2.1], making them ineffective on other micro-architectures, requiring a reevaluation of the candidates.

For MOV, we had to implement register rotation to prevent pipeline stalls. When many MOV instructions operate on the same register, the used micro-architecture would at a certain number of MOVs cause much larger slowdowns than expected. By rotating registers for the inserted MOVs, we avoided this effect and achieved a more uniform slowdown. However, since the available set of registers and the pipelining approaches are CPU-specific, using MOV instructions may require additional tuning for other micro-architectures.

6 Importance of Distributing Slowdown

As discussed in Section 3.2, the instructions within a block can come from multiple source methods. Therefore, adding instructions only at the start or at the end of the block, we could bias the performance behavior.

To test what effect the placement of slowdown has, we ran experiments that placed the slowdown at the start of the block, mixed throughout the block (our default approach), and with all the slowdown at the end. Similar to Section 5.5, we then used async-profiler to record profiles.

Figure 4 shows that different placements cause async-profiler to see noticeable differences in run time percentage for methods. Specifically, for NOP and MOV we see that placement affects how accurately the performance behavior is maintained. For example, for NOP, placing the slowdown at the end of a block cause a larger change than when the slowdown is mixed in. To quantify the difference, we use the mean squared error (MSE). For NOP with the slowdown mixed throughout, the mean squared error remains 1.2, as in Section 5.5. However, when the slowdown is at the block start, it goes up to 5.7, but when at the end, the MSE is only 1.4. For MOV, the MSE is 1.5 when mixing in the slowdown, and it goes to 4.3 for slow down at the start and 6.4 for slow down at the end. This not only confirms that placement can be important for specific use cases, but also suggests that indeed finer-grained instructions should be used that can be placed more freely and uniformly throughout basic blocks.

7 Related Work

While various projects utilized slowdown [4, 6, 8, 12–14], we are not aware of any study on the most suitable slowdown instruction. However, there is work around simulators that seems relevant. Abel and Reineke [1] present a basic-block cycle-cost simulator capable of predicting, in a simulated environment, the steady-state cycles per iteration of a given basic block. If such a tool were adapted for our use case, it could be used to slow down blocks on a more fine-grained basis, for instance by exploiting instruction dependencies and reduce the time it takes to determine how many slowdown instructions to insert. Carlson et al. [5] introduce Sniper, a full-system simulator that predicts run time by grouping instructions between cache-miss and branch-misprediction events into coarse intervals, This abstraction runs at millions of simulated instructions per second while keeping cycle-count error within roughly 25% of measured hardware. Thus, Sniper could possibly improve the performance of our approach to determining how much slowdown to add.

8 Conclusion & Future Work

This paper explores six x86 candidates for adding accurate, side-effect-free, and distributed slowdown at the basic block level. Of the six candidates, we found NOP and MOV to be the most suitable. They achieved the target slowdown at a program and basic block level, while maintaining the observable performance behavior of the used benchmark. Thus, adding slowdown preserves where the program proportionally spends its time as observed by a Java profiler. This means, the percentage of run time spent in each method remained consistent between the normal and slowed down execution.

In future work, we aim to explore the application of our slowdown technique to the identified use cases of race detection and virtual speedup. Slowdown at the basic-block level may lead to more accuracy and precision compared to coarser-grained approaches. We already started assessing the accuracy of profilers [4].

References

[1] 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

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

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

[4] Humphrey Burchell and Stefan Marr. 2025. Divining Profiler Accuracy: An Approach to Approximate Profiler Accuracy Through Machine Code-Level Slowdown. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 402 (Oct. 2025), 29 pages. doi:10.1145/3763180

[5] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. ACM, Article 52, 12 pages. doi:10.1145/2063384.2063454

[6] 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 (SOSP '15)*. ACM, 184–197. doi:10.1145/2815400.2815409

[7] 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

[8] 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

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

[10] Intel. 2024. *Optimizing Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency.* Technical Report. https://www.intel.com/content/www/us/en/content-details/814199/optimizing-earlier-generations-of-intel-64-and-ia-32-processor-architectures-throughput-and-latency.html

[11] 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

[12] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, 267–280.

[13] 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

[14] Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2023. WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. ACM, 111–126. doi:10.1145/3552326.3567507

[15] 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

A Appendix

Table 2. Candidate slowdown instructions used in our study

Name	Opcode Template	Definition / Purpose	Motivation for Inclusion
NOP (2-byte)	`66 90`	No operation; e.g., used for code alignment.	Has zero side effects and intended for this use.
MOV (reg→reg)	`89 /r /r`	Dummy move of a register to itself, without side effects.	A fast and highly optimized instruction.
PUSH+POP	`50 /r 58 /r`	Push a register onto the stack and then pop it back into the same register.	A small sequence that may consume more cycles.
SFENCE	`0F AE`	Store-fence; ensures all preceding stores are globally visible before later memory operations.	Serializing delay that leaves registers untouched.

Long PUSH+POP sequence	`SUB SP, IMM`
	`VMOVDQU [SP], VEC`
	`VMOVDQU VEC, [SP]`
	`ADD SP, IMM`
		Allocates temporary stack space, spills and refills a vector register, then releases the space—consumes cycles without affecting logic.	Uses the vector unit of the CPU, to compare the performance behavior with a normal PUSH+POP.

PAUSE	`F3 90`	Used in spin-wait loops, as a hint for the CPU to improve their performance.	A purpose-built delay instruction.