Benchmarking on Modern Hardware

Techniques for Performance Comparisons from Day-To-Day Experimenting to Paper Writing

May 2026

JOHANNES KEPLER
UNIVERSITÄT LINZ

Got a Question?
Please Interrupt Me!

Agenda

- Why is benchmarking hard?
- What do we need for a research paper?
- What do we need for day-to-day engineering?
- A practical process

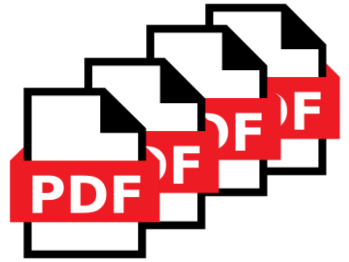
Why Is Benchmarking Hard?

A Hypothetical Benchmark

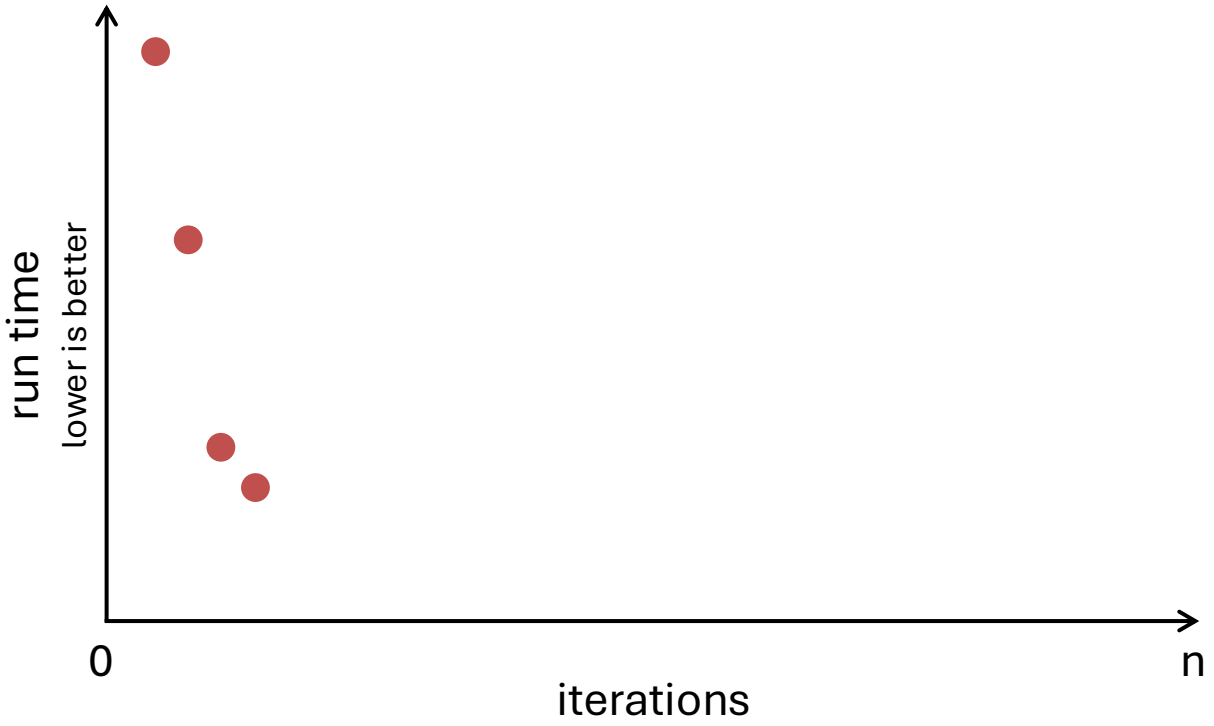
- Large Java application
- Running on HotSpot JDK 25

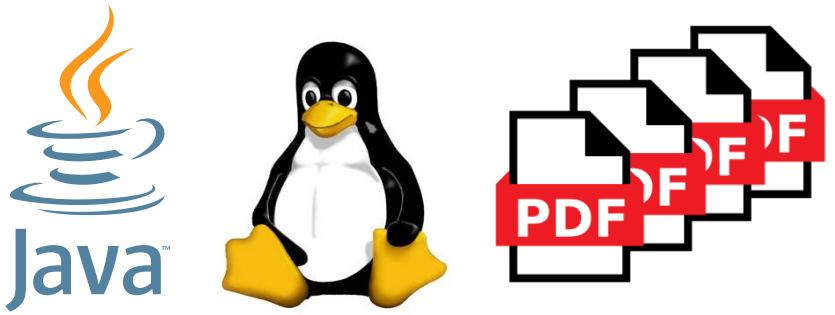
- On an up-to-date Ubuntu Linux 24.04

- It’s “reasonably deterministic” workload
  - a batch job, e.g., document processing



Measurement Results

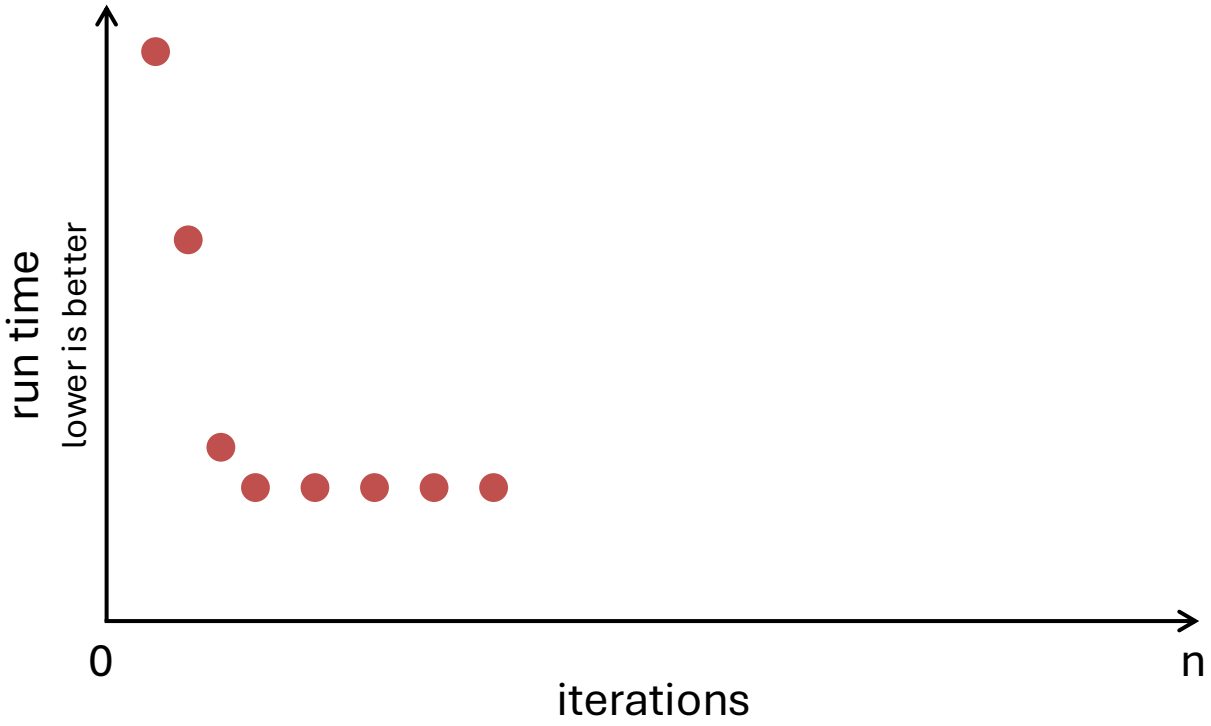




What are we likely observing here?

JIT compiler making our benchmark faster

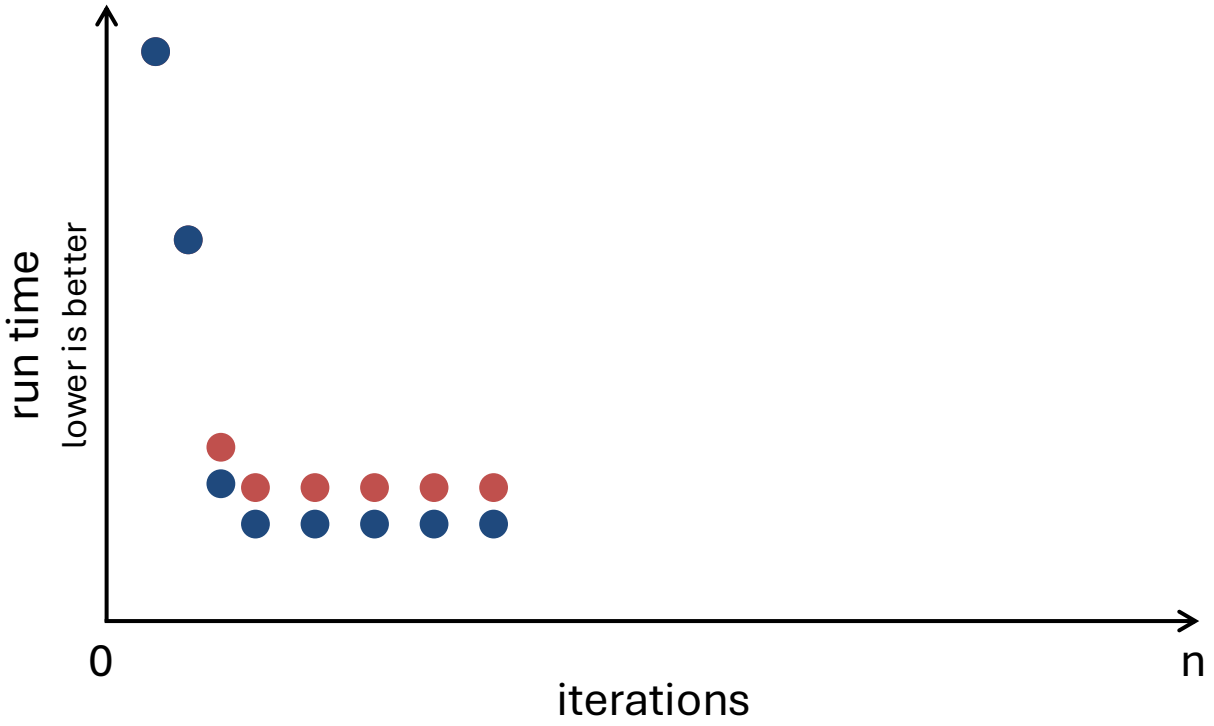
Measurement Results



What do we expect to happen next?

Ideally, we'd have reached "peak" performance

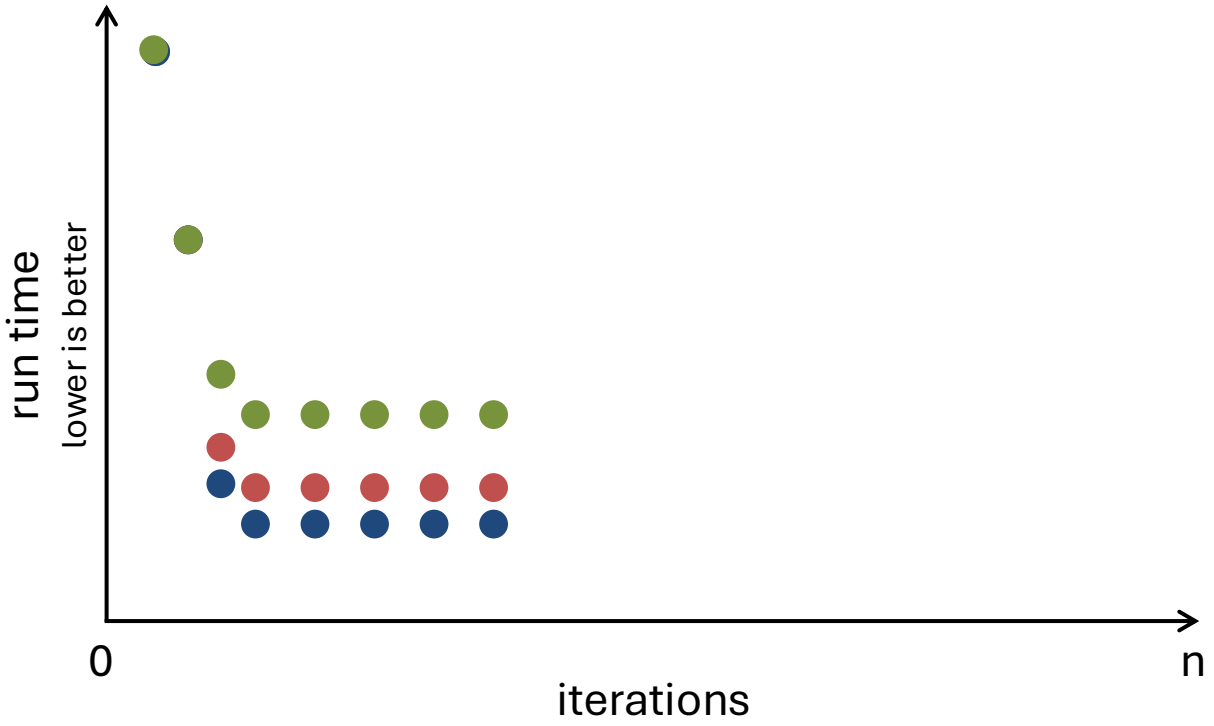
Results of a 2nd Benchmark Invocation



- Different memory location
- Different core type
- Different clock frequency
- Different compiler decisions
- …

Why could it be faster?

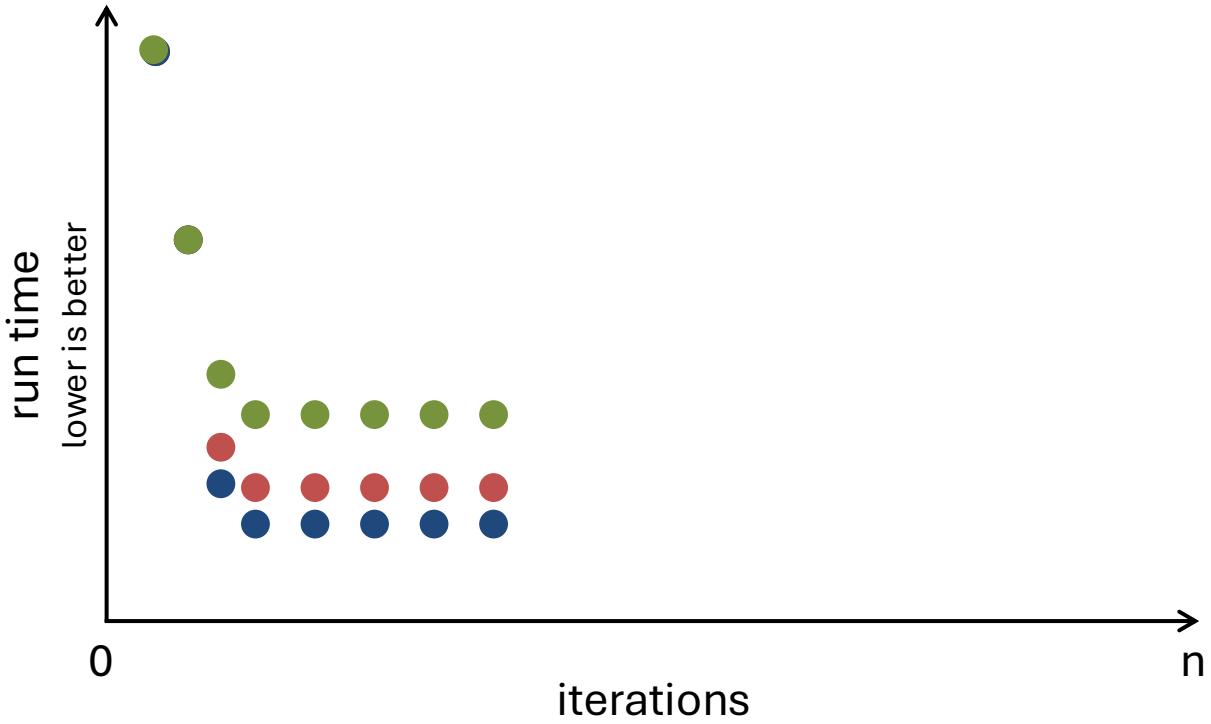
Results of a 3rd Benchmark Invocation



- Different memory location
- Different core type
- Different clock frequency
- Different compiler decisions
- …

Why could it be slower?

Results of Three Benchmark Invocations

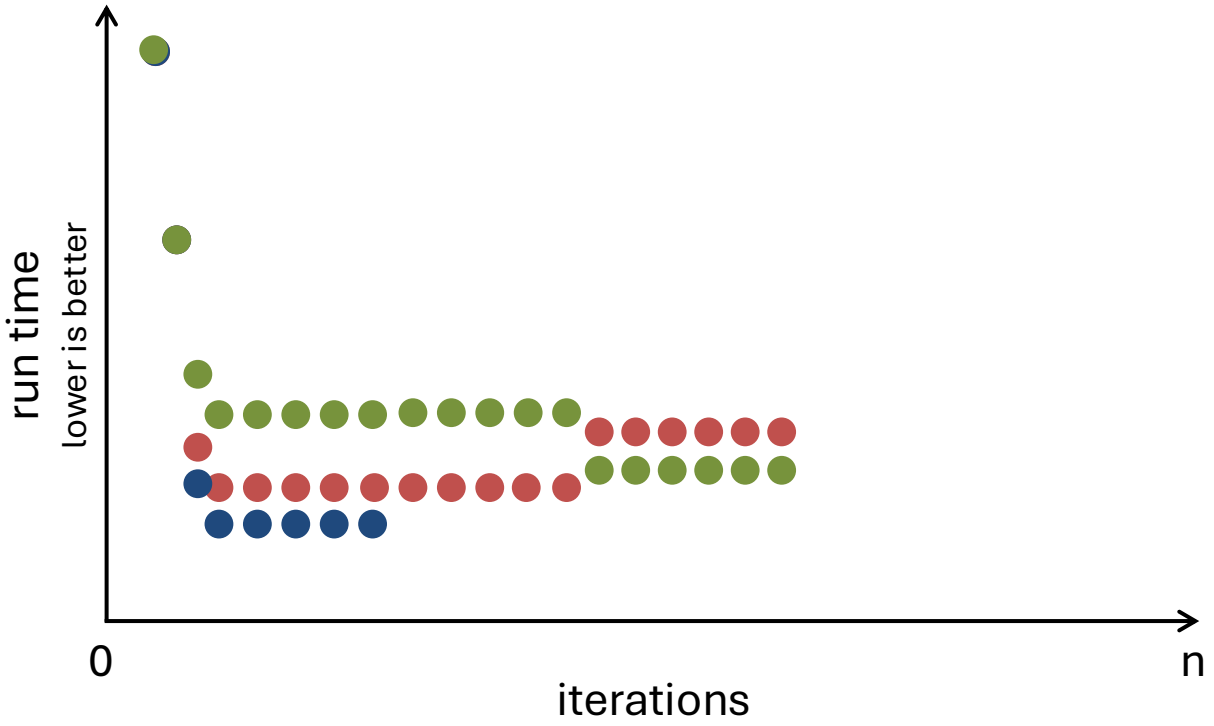


Assuming we made no methodological mistake, they all are.

But, need to check our methodology!

Which one is the correct data set?

Invocation 1 had more data…



- Scheduled on different code/core type
- JVM change memory layout
- …

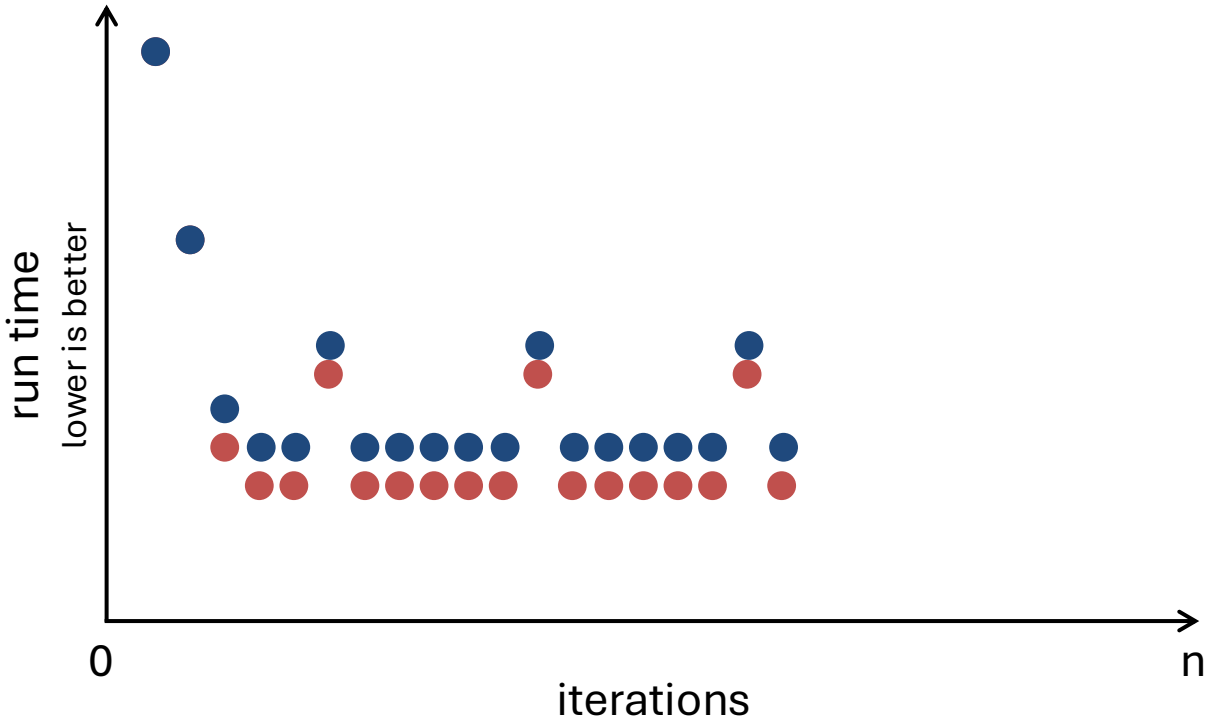
What could have happened here?

Late JVM memory layout change observed in:

Experimental Evaluation Methodology for the Era of No Steady Performance

J. Antoch, W. Binder, L. Bulej, F. Farquet, V. Horký, A. Prokopec, A. Rosà, and P. Tůma. *10 (OOPSLA1): 33 (2026)*
DOI: 10.1145/3798236

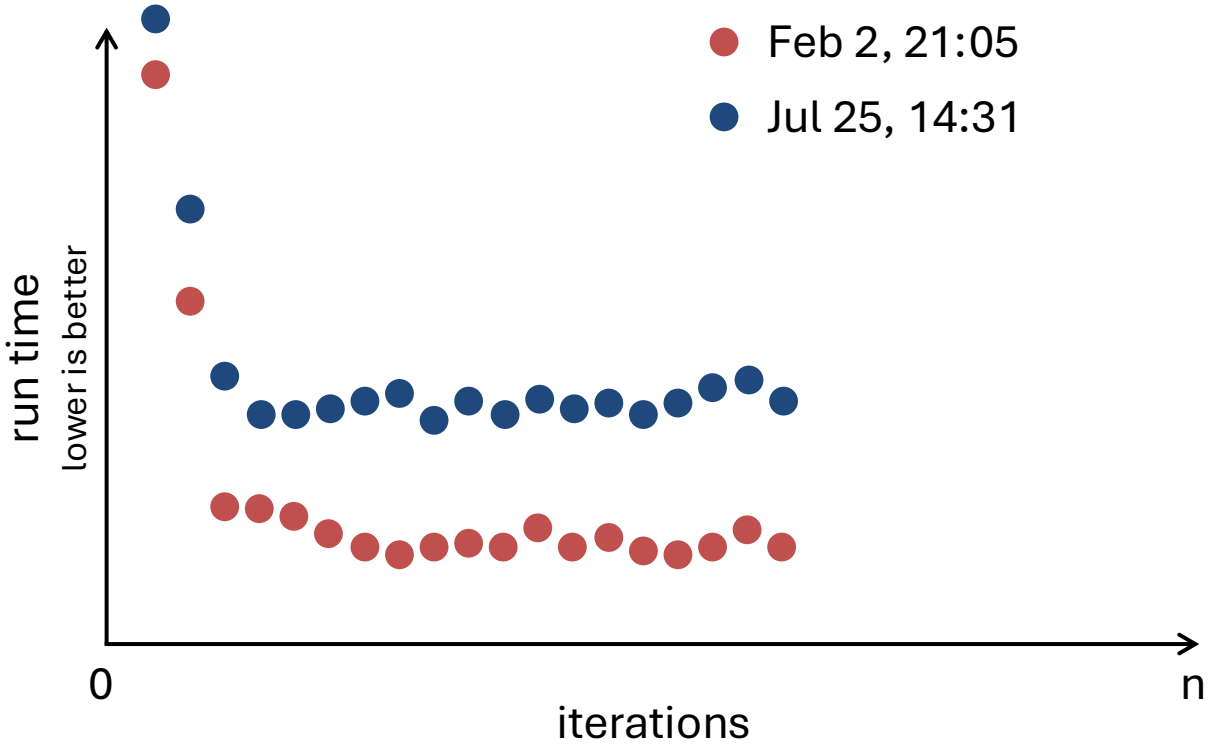
Same Benchmark, Different Input



- GC triggering in regular intervals
- …

What could be happening here?

Same Benchmark, Different Input



- Benchmarking on a laptop
  - Different temperature
  - Plugged into power?
- Software updates
- …

What could be happening here?

What makes benchmarking hard? (1/2)

- Run-time Optimizations
  - Feedback-based compiler decisions
  - Memory changes, garbage collection
- Security mechanism
  - Address Space Layout Randomization
- Hardware complexity
  - Memory hierarchies, caches, locality
  - Cores, core types, …
  - Dynamic clock frequency changes
    - Thermal budgets

What makes benchmarking hard? (2/2)

- Environmental impact
  - Temperature
  - Power source
- Software updates/changes
  - Linking order

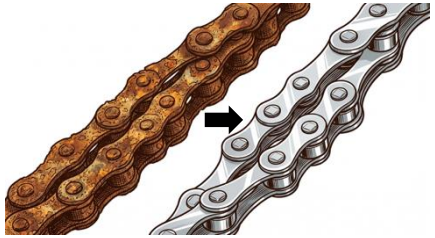
- And, many more?...

What Do We Need for a Paper?

Requirements and Paper Types

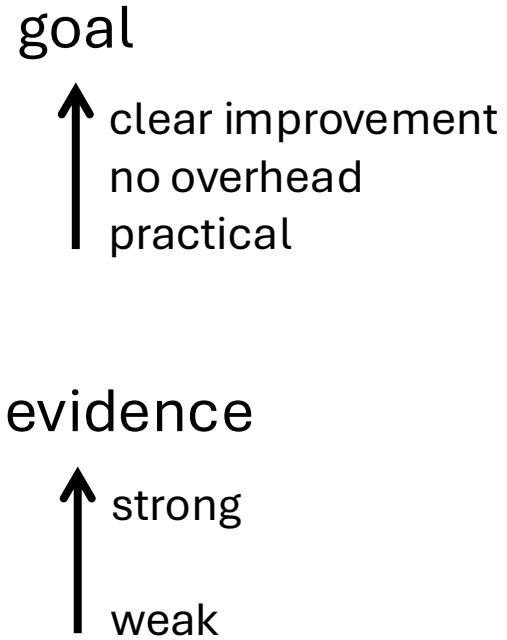
Potential Types of Papers

- A “New” Thing: A new category
  - Focus not on performance
- A “Better” Thing: In existing category, but better
  - Performance “as least as good”
- An “Improvement/Optimization”
  - Needs to improve performance
- An “Observation” Study
  - Observe, Analyze, Report on existing things



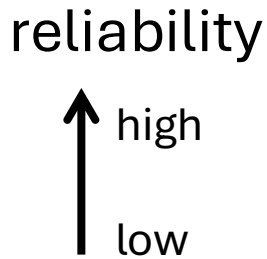
Dimensions of Performance Experiments

- Goal: what one aims to show
  - It's "practical"
  - Has no overhead
  - Clear improvement
- Level of Evidence
  - depth and breadth of experiments
    - number/diversity of benchmarks/workloads
  - type of properties demonstrated
    - run time, memory use, scalability, …



Dimensions of Performance Experiments

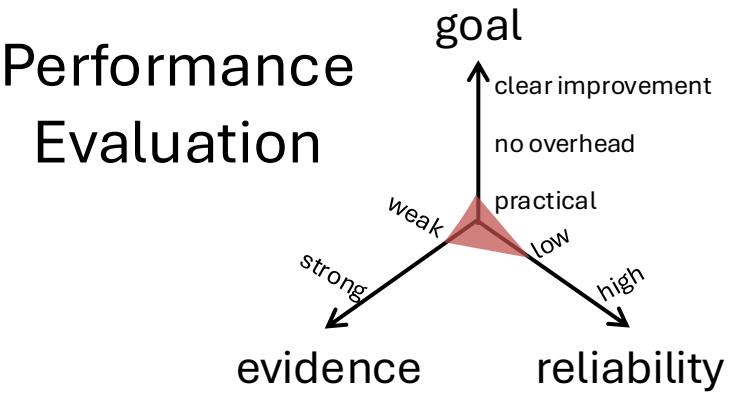
- Statistical reliability
  - number of measurements
  - measurement methodology
  - analysis methodology



- Publication venue
  - Different expectations
    - Workshop
    - Specialized conference
    - Top conference

The “New” Thing

Challenge: no existing benchmarks



Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language properties in order to optimise performance of garbage collection. Namely, Orca relies on the absence of race conditions in order to avoid read/write barriers, and it leverages actor message passing for synchronisation among actors. This paper describes Pony, its type system, and the Orca garbage collection algorithm. An evaluation of the performance of Orca suggests that it is fast and scalable for idiomatic workloads.

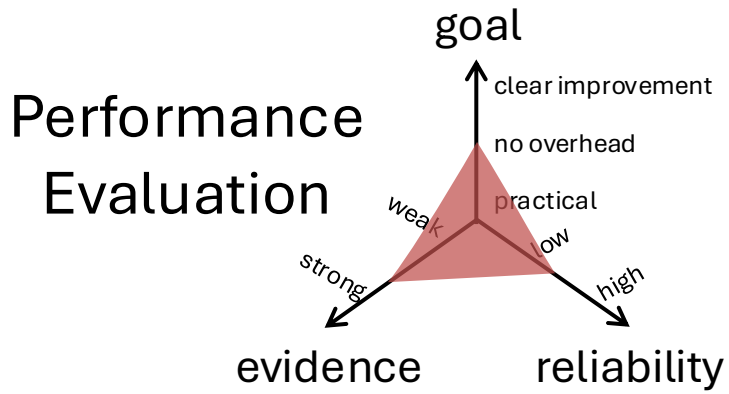
When Concurrency Matters: Behaviour-Oriented Concurrency

LUKE CHEESEMAN, Imperial College London, UK
MATTHEW J. PARKINSON, Microsoft Azure Research, UK
SYLVAN CLEBSCH, Microsoft Azure Research, UK
MARIOS KOGIAS, Imperial College London, UK Microsoft Research, UK
SOPHIA DROSSOPOULOU, Imperial College London, UK
DAVID CHISNALL, Microsoft, UK
TOBIAS WRIGSTAD, Uppsala University, Sweden
PAUL LIÉTAR, Imperial College London, UK

Expressing parallelism and coordination is central for modern concurrent programming. Many mechanisms exist for expressing both parallelism and coordination. However, the design decisions for these two mechanisms are tightly intertwined. We believe that the interdependence of these two mechanisms should be recognised and achieved through a single, powerful primitive. We are not the first to realise this: the prime example is actor model programming, where parallelism arises through fine-grained decomposition of a program's

A “Better” Thing

Challenge: Engineering Cost



JavaScript AOT Compilation

Manuel Serrano
Inria/Université Côte d’Azur
Sophia Antipolis, France
Manuel.Serrano@inria.fr

Abstract

Static compilation, *a.k.a.,* ahead-of-time (AOT) compilation, is an alternative approach to JIT compilation that can combine good speed and lightweight memory footprint, and that can accommodate read-only memory constraints that are imposed by some devices and some operating systems. Unfortunately the highly dynamic nature of JavaScript makes it hard to compile statically and all existing AOT compilers have either gave up on good performance or full language support. We have designed and implemented an AOT compiler that aims at satisfying both. It supports full unrestricted ECMAScript 5.1 plus many ECMAScript 2017 features and the majority of benchmarks are within 50% of the performance of one of the fastest JIT compilers.

CCS Concepts • **Software and its engineering → Object oriented languages**; **Functional languages**; **Compilers**; **Runtime environments**;

Kinoma.js, ...). In this application domain, JavaScript programs execute on small devices that have limited hardware capacities, for instance only a few kilobytes of memory. Just-in-time (JIT) compilation, which has proved to be so effective for improving JavaScript performances [Chang et al. 2009; Chevalier-Boisvert and Feeley 2015, 2016; Gal et al. 2009], is unthinkable in these constrained environments. There would be just not enough memory nor CPU capacity to execute them at runtime. Furthermore memory write operations on executable segments are sometimes impossible on the devices, either because of the type of memory used (ROM or FLASH) or simply because the operating system forbids them (iOS for instance). Pure JavaScript interpreters are then used, but this comes with a strong performance penalty, especially when compared to assembly or C programs, that limits the possible uses.

When JIT compilation is not an option and when interpre-

Rust as a Language for High Performance GC Implementation

Yi Lin[†] Stephen M. Blackburn[†] Antony L. Hosking[†*$] Michael Norrish[*]
[†]Australian National University [*]Data61, Australia [$]Purdue University, USA
[†]{yi.lin,steve.blackburn,antony.hosking}@anu.edu.au [*]{antony.hosking,michael.norrish}@data61.csiro.au

Abstract

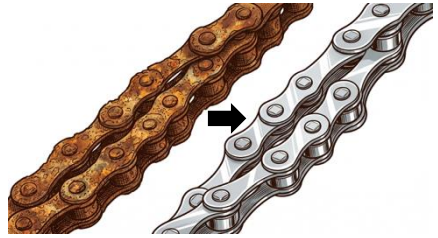
High performance garbage collectors build upon performance-critical low-level code, typically exhibit multiple levels of concurrency, and are prone to subtle bugs. Implementing, debugging and maintaining such collectors can therefore be extremely challenging. The choice of implementation language is a crucial consideration when building a collector. Typically, the drive for performance and the need for efficient support of low-level memory operations leads to the use of low-level languages like C or C++, which offer little by way of safety and software engineering benefits. This risks undermining the robustness and flexibility of the collector design. Rust’s ownership model, lifetime specification, and reference bor-

1. Introduction

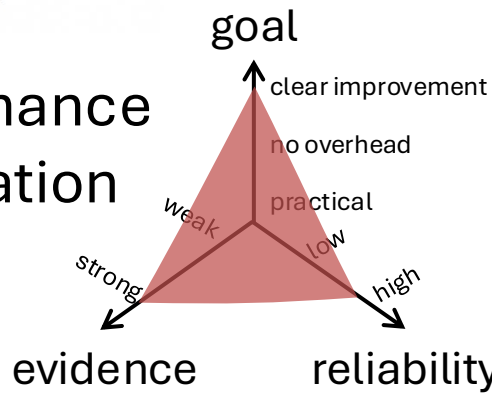
A fast yet robust garbage collector (GC) is the key to garbage collected language runtimes. However, implementing such a GC is not easy. First, a collector must manipulate raw memory, depending on carefully optimized code to do so, making it naturally prone to memory bugs. Second, high performance GCs are rich in concurrency, typically featuring thread parallelism, including thread-local allocation, parallel tracing, and possibly mutator concurrency, making it prone to race conditions and extremely time consuming bugs.

What makes the situation worse is that the implementation language usually does not provide help in terms of memory safety and thread safety. The imperative of performance encourages the

An “Improvement/Optimization”



Challenge: Engineering Cost, Suitable Benchmarks



CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers

ADITYA ANAND, Indian Institute of Technology Bombay, India
VIJAY SUNDARESAN, IBM Canada Lab, Canada
DARYL MAIER, IBM Canada Lab, Canada
MANAS THAKUR, Indian Institute of Technology Bombay, India

Just-in-time (JIT) compilers typically sacrifice the precision of program analysis for efficiency, but are capable of performing sophisticated speculative optimizations based on run-time profiles to generate code that is specialized to a given execution. On the contrary, ahead-of-time static compilers can often afford precise flow-sensitive interprocedural analysis, but produce conservative results in scenarios where higher precision could be derived from run-time specialization. In this paper, we propose the first-of-its-kind approach to enrich static analysis with the possibility of speculative optimization during JIT compilation, as well as its usage to perform aggressive stack allocation on a production Java Virtual Machine (JVM).

Our approach of combining static analysis with JIT speculation – named `CoSSJIT` – involves three key contributions. First, we identify the scenarios where a static analysis would make conservative assumptions but a JIT could deliver precision based on run-time speculation. Second, we present the notion of “speculative conditions” and plug them into a static interprocedural dataflow analyzer (whose aim is to identify heap objects

Zero-Overhead Metaprogramming

Reflection and Metaobject Protocols
Fast and without Compromises

Stefan Marr
RMoD, Inria, Lille
France
stefan.marr@inria.fr

Chris Seaton
Oracle Labs / University of Manchester
United Kingdom
chris.seaton@oracle.com

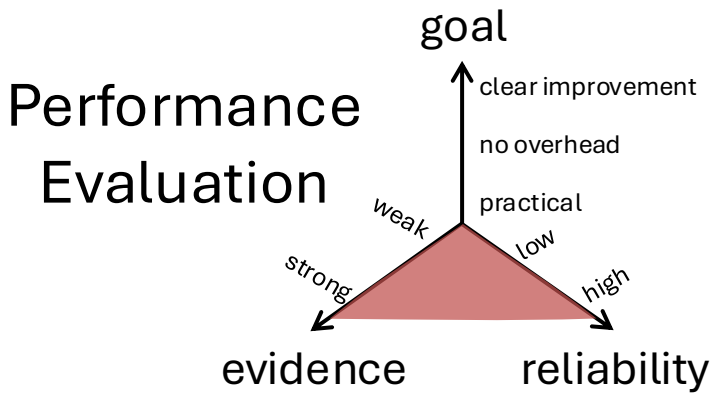
Stéphane Ducasse
RMoD, Inria, Lille
France
stephane.ducasse@inria.fr

Abstract

Runtime metaprogramming enables many useful applications and is often a convenient solution to solve problems in a generic way, which makes it widely used in frameworks, middleware, and domain-specific languages. However, powerful metaobject protocols are rarely supported and even common concepts such as reflective method invocation or dynamic proxies are not optimized. Solutions proposed in literature either restrict the metaprogram-

Ruby, or Smalltalk, the runtime metaprogramming facilities enable DSLs designers to taylor the host language to enable more concise programs. Metaobject protocols (MOPs), as in Smalltalk or CLOS, go beyond more common metaprogramming techniques and enable for example DSL designers to change the language’s behavior from within the language [Kiczales et al. 1991]. With these additional capabilities, DLSs can for instance have more restrictive language semantics than the host language they are embedded in.

An “Observation” Study



Challenge: needs depth to create new knowledge

Virtual Machine Warmup Blows Hot and Cold*

EDD BARRETT, King’s College London, UK
CARL FRIEDRICH BOLZ-TEREICK, King’s College London, UK
REBECCA KILLICK, Lancaster University, UK
SARAH MOUNT, King’s College London, UK
LAURENCE TRATT, King’s College London, UK

Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling those parts into machine code; subsequently the program is said to be at a steady state of peak performance. Measurement methodologies almost always discard data collected during the warmup phase such that reported measurements focus entirely on peak performance. We introduce a fully automated statistical approach, based on changepoint analysis, which allows us to determine if a program has reached a steady state and, if so, whether that represents peak performance or not. Using this, we show that even when run in the most controlled of circumstances, small, deterministic, widely studied microbenchmarks often fail to reach a steady state of peak performance on a variety of common VMs. Repeating our experiment on 3 different machines, we found that at most 43.5% of ⟨VM, benchmark⟩ pairs consistently reach a steady

AST vs. Bytecode: Interpreters in the Age of Meta-Compilation

OCTAVE LAROSE, University of Kent, UK
SOPHIE KALEBA, University of Kent, UK
HUMPHREY BURCHELL, University of Kent, UK
STEFAN MARR, University of Kent, UK

Thanks to partial evaluation and meta-tracing, it became practical to build language implementations that reach state-of-the-art peak performance by implementing only an interpreter. Systems such as RPython and GraalVM provide components such as a garbage collector and just-in-time compiler in a language-agnostic manner, greatly reducing implementation effort. However, meta-compilation-based language implementations still need to improve further to reach the low memory use and fast warmup behavior that custom-built systems provide. A key element in this endeavor is interpreter performance. Folklore tells us that bytecode interpreters are superior to abstract-syntax-tree (AST) interpreters both in terms of memory use and run-time performance.

This work assesses the trade-offs between AST and bytecode interpreters to verify common assumptions and whether they hold in the context of meta-compilation systems. We implemented four interpreters, each an AST and a bytecode one using RPython and GraalVM. We keep the difference between the interpreters as small as feasible to be able to evaluate interpreter performance, peak performance, warmup, memory use, and the impact of individual optimizations.

Our results show that both systems indeed reach performance close to Node.js/V8. Looking at interpreter-

What Do We Need for Day-to-Day Engineering?

Requirements and Paper Types

Testing Needs to be Trivial!

```
git commit
git push
```

If it takes more,
things will break
without us noticing!

Benchmarking Needs to be Trivial!

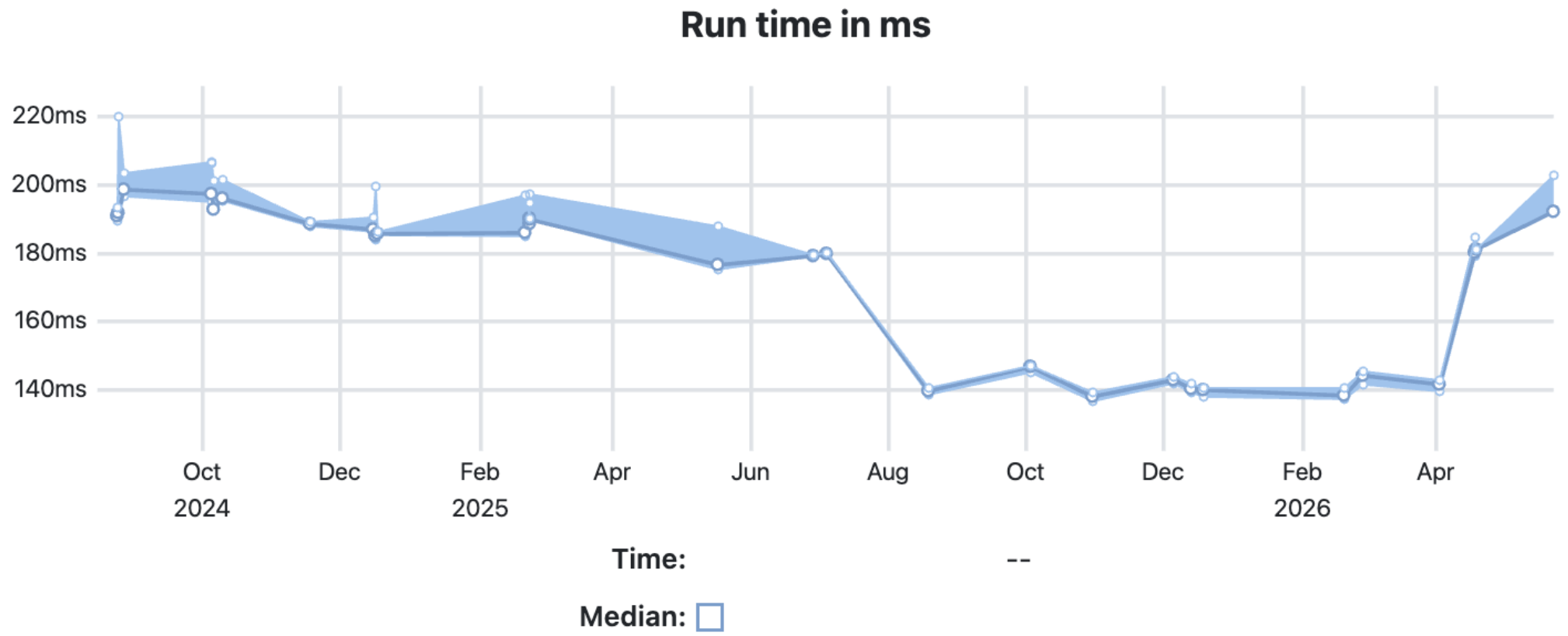
```
git commit
git push
```

If it takes more, things will be slow without us noticing!

You want tracking over time

CD

```
som-native-interp-ast -cp
Examples/AreWeFastYet/CD:Examples/AreWeFastYet/Havlak:Examples/AreWeFastYet/Core:Smalltalk
Examples/AreWeFastYet/Harness.som --gc CD 1 10
```



Comparing Specific Revisions

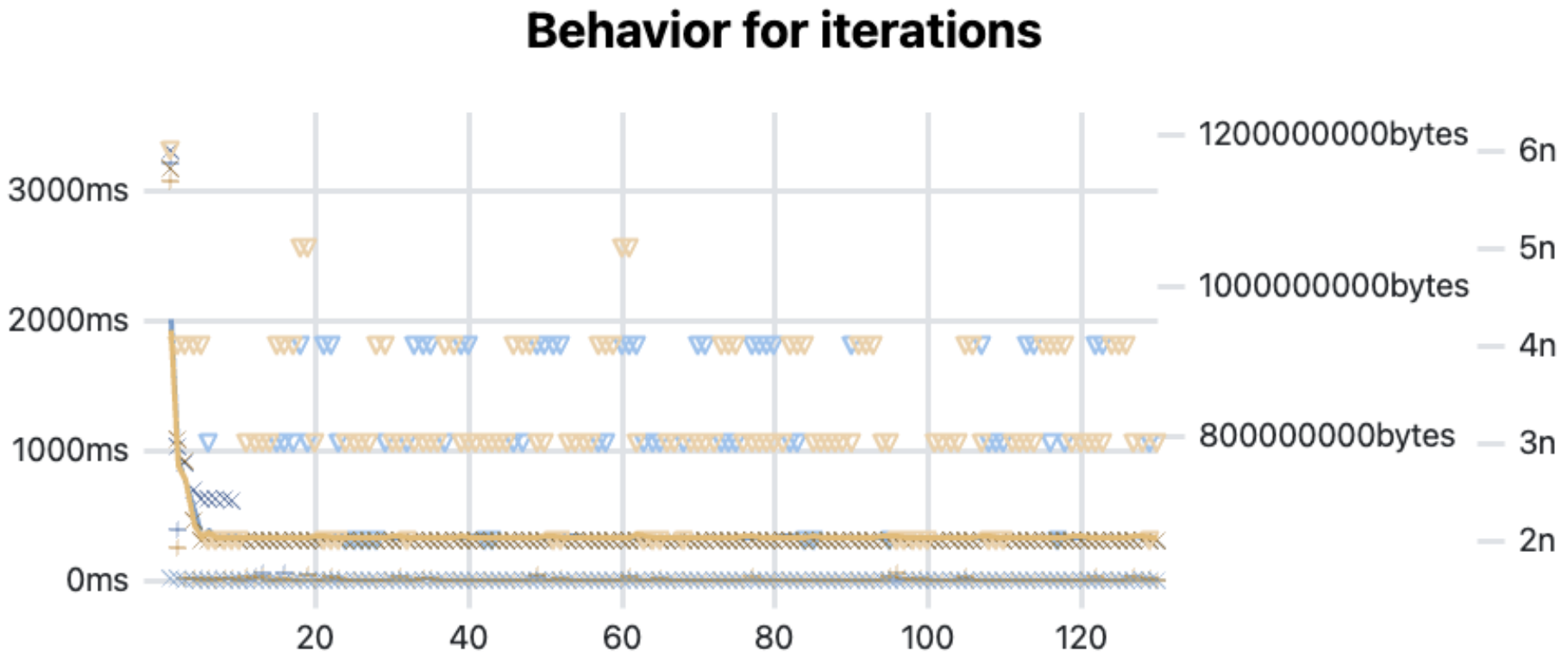
awfy-steady

Executor: TruffleSOM-graal

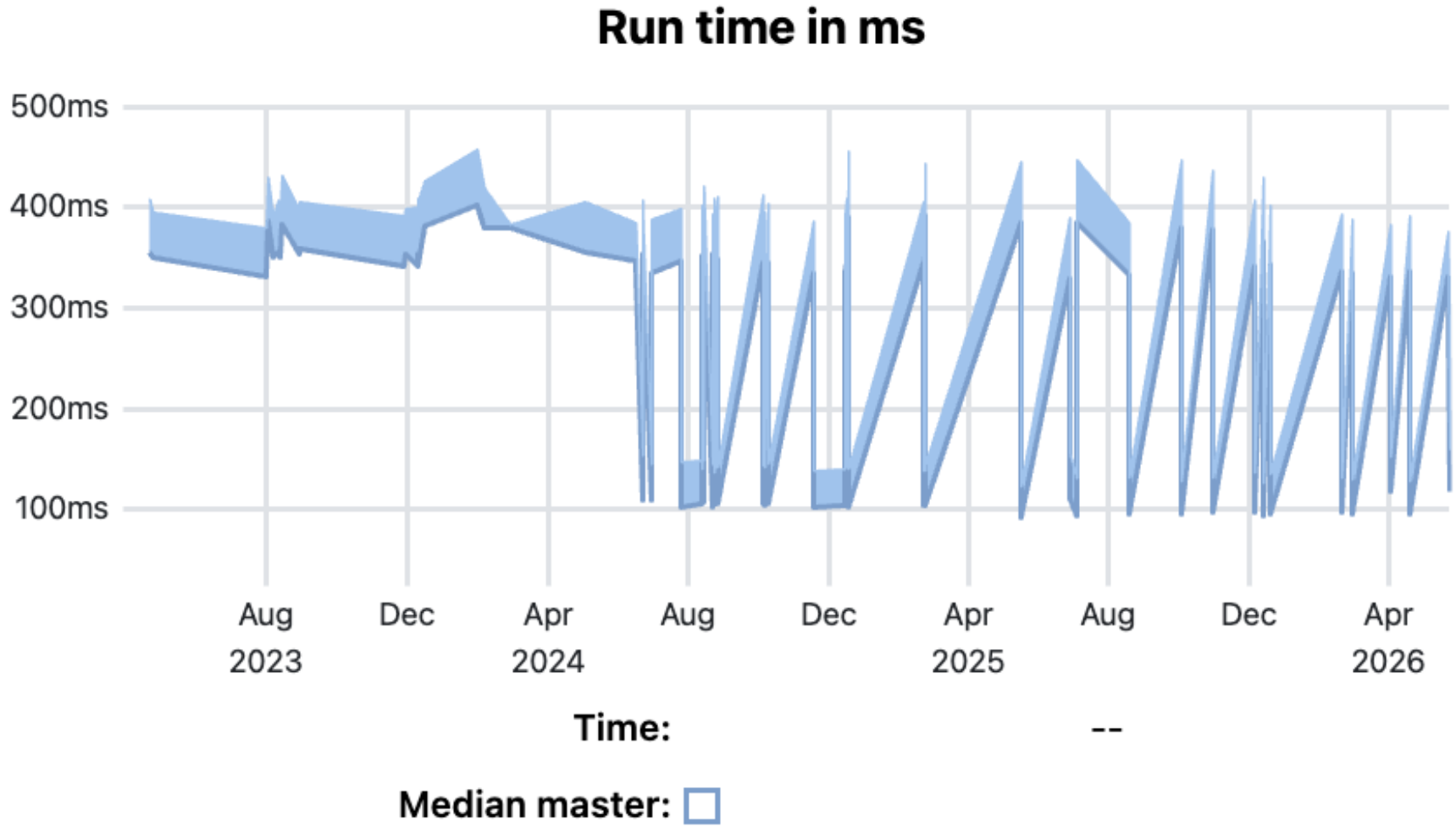
		#M	median time in ms	time diff %	median allocated in bytes	allocated diff %	median compile in ms	comp
CD		130	25.00	0	163MB	0	0	
CD		130	34.74	28	165MB	-1	0	
Havlak		130	335.46	1	632MB	0	0	
Havlak		130	96.76	-17	640MB	-17	0	

See all data at a glance

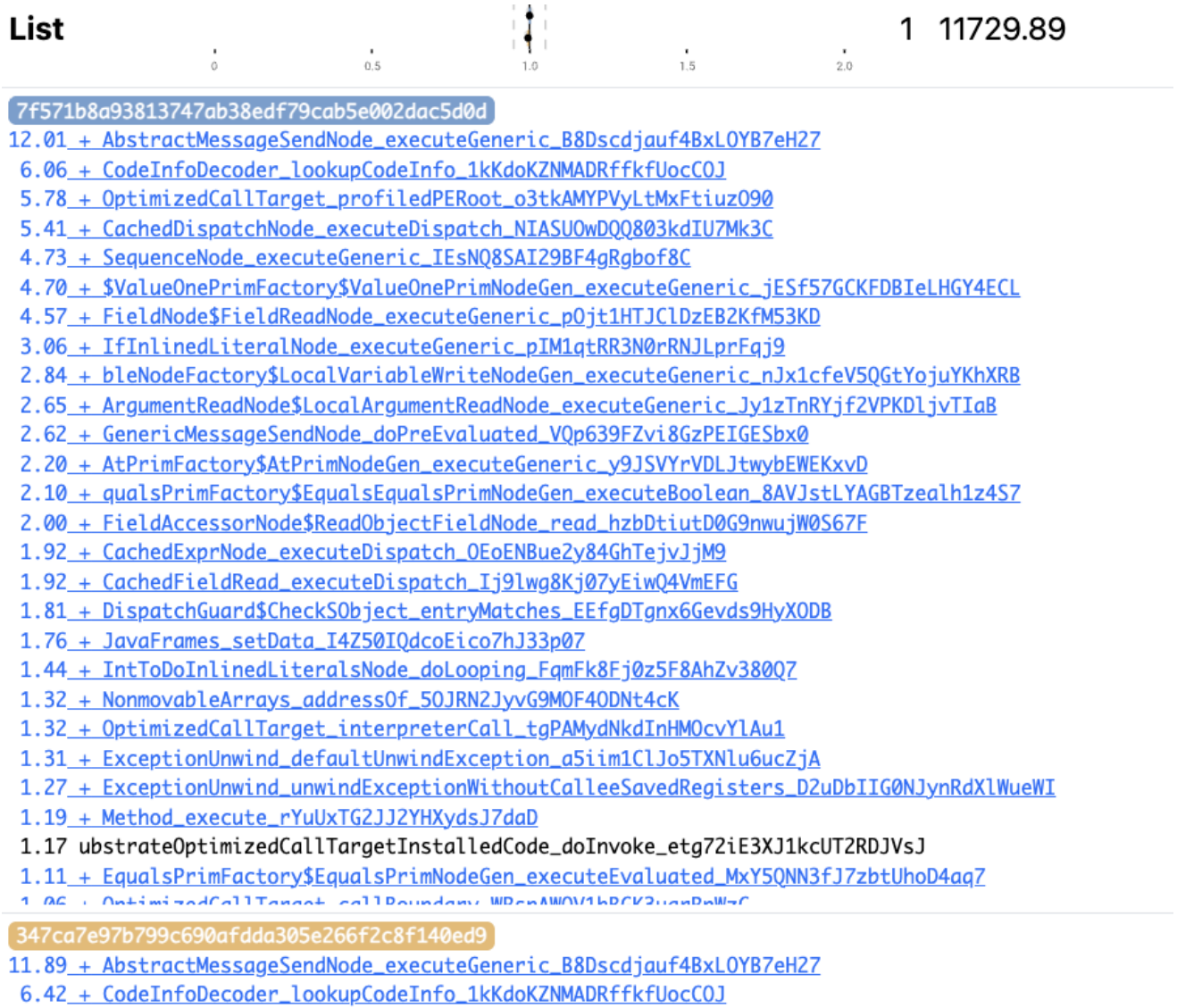
Havlak	130	335.46	1



As well as historic data as context



It’s not a total black box, profiles help!



Denoise

- raise the process priority
- use CPU core shielding
- request CPU to work at constant frequency

```
sudo denoise --num-cores 8 --cset-path /usr/bin/cset init
sudo denoise --num-cores 8 --cset-path /usr/bin/cset minimize
sudo denoise --num-cores 8 --cset-path /usr/bin/cset exec -- ./my-benchmark1 --my-args
sudo denoise --num-cores 8 --cset-path /usr/bin/cset exec -- ./my-benchmark2 --my-args
sudo denoise --num-cores 8 --cset-path /usr/bin/cset restore
```

Which benchmarks to run?

- Feedback needs to be timely, maybe 30min?
- Benchmarks you need for a paper
  - Perhaps in smaller problem sizes
  - Ensure they work. And keep working!
- Enough to have some confidence
  - More signal than noise…
  - Fast enough feedback

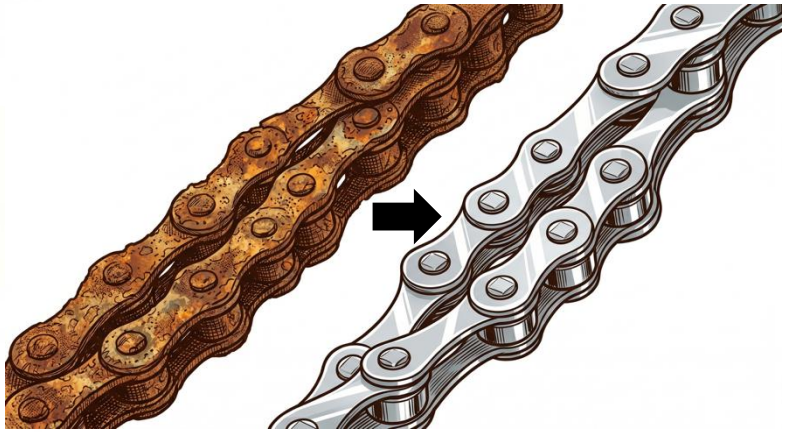
A Practical Process

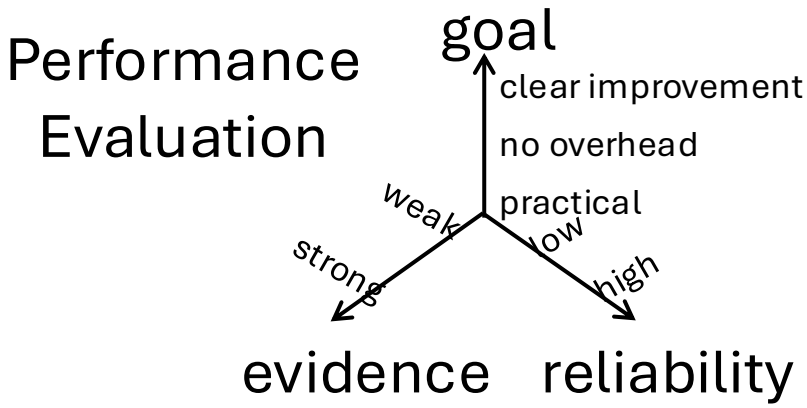
Create, Commit, Benchmark, Analyze, Write Up, Publish

Benchmarking, Automation, Result Tracking, Data Analysis, and Paper Writing

Key Points

Your Paper Needs a Methodology That Suites It





No Silver Bullet, But A Useful Checklist

SIGPLAN Empirical Evaluation Checklist

*This checklist is meant to **support** informed judgement, not **supplant** it.*

Clearly Stated **Claims**

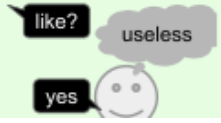
Example Violations

""

Claims not explicit
Claims must be explicit in order for the reader to assess whether the empirical evaluation supports them. Missing claims cannot possibly be assessed. Claims should also aim to state not just what is achieved but how.

"precise analysis"

Claims not appropriately scoped
The truth of a claim should clearly follow from the evidence provided. Claims that are not fully supported mislead readers. 'Works for all Java' is over-broad when based on a subset of Java. Other examples are 'works on real hardware' when evaluating only with (unrealistic) simulation, and 'automatic process' when requiring human intervention.



Fails to acknowledge limitations
A paper should acknowledge its limitations to place the scope of its results in context. Stating no limitations at all, or only tangential ones, while omitting the more relevant ones

Relevant **Metrics**

Example Violations

"energy consumed"

Indirect or inappropriate proxy metric
Proxy metrics can substitute for direct ones only when the substitution is clearly, explicitly justified. For example, it would be misleading and incorrect to report a reduction in cache misses to claim actual end-to-end performance or energy consumption improvement.

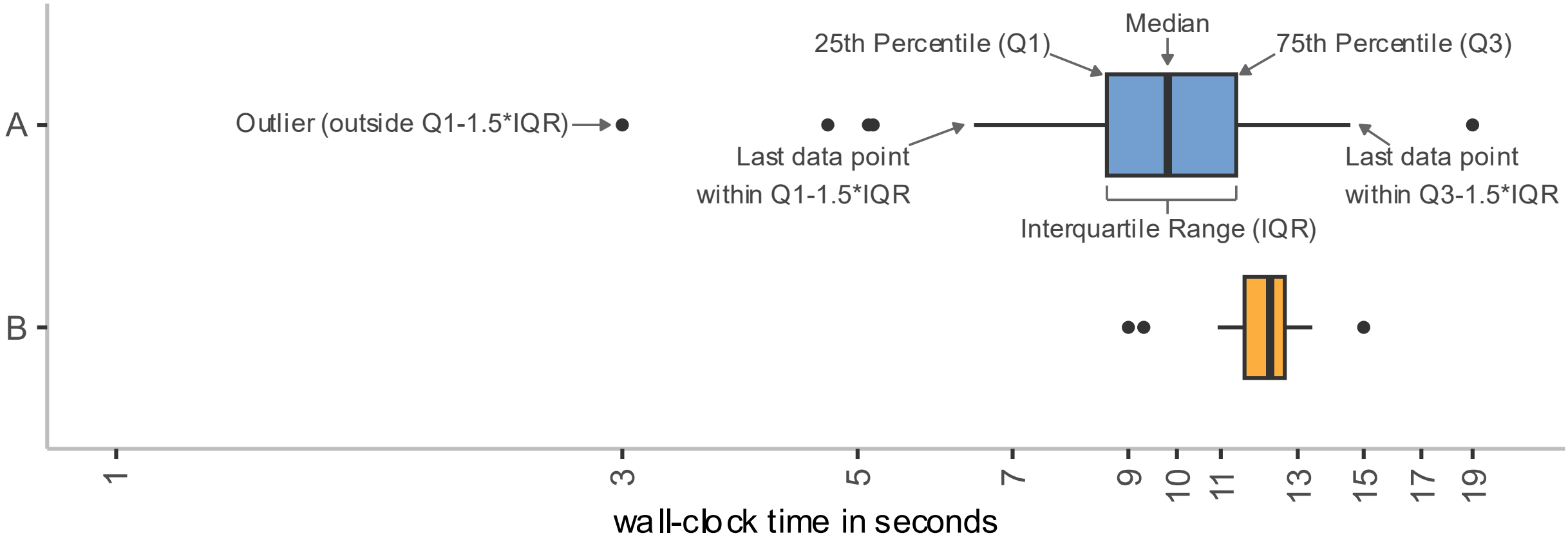
"devs were satisfied"

Fails to measure all important Effects
All important effects should be measured to show the true cost of a system. For example, compiler optimizations may speed up programs at the cost of drastically increasing compile times of large systems, so the compile time should be measured as well as the program speedup. Failure to do so distorts the cost/benefit of the system.

Insufficient information to repeat
Experiments evaluating an idea need to be described in sufficient detail to be repeatable. All parameters (including default values) should be included, as well as all version

https://www.sigplan.org/Resources/EmpiricalEvaluation/

Example: Show Distributions: Boxplots or Better!



https://stefan-marr.de/2024/06/5-reasons-for-box-plots-as-default/

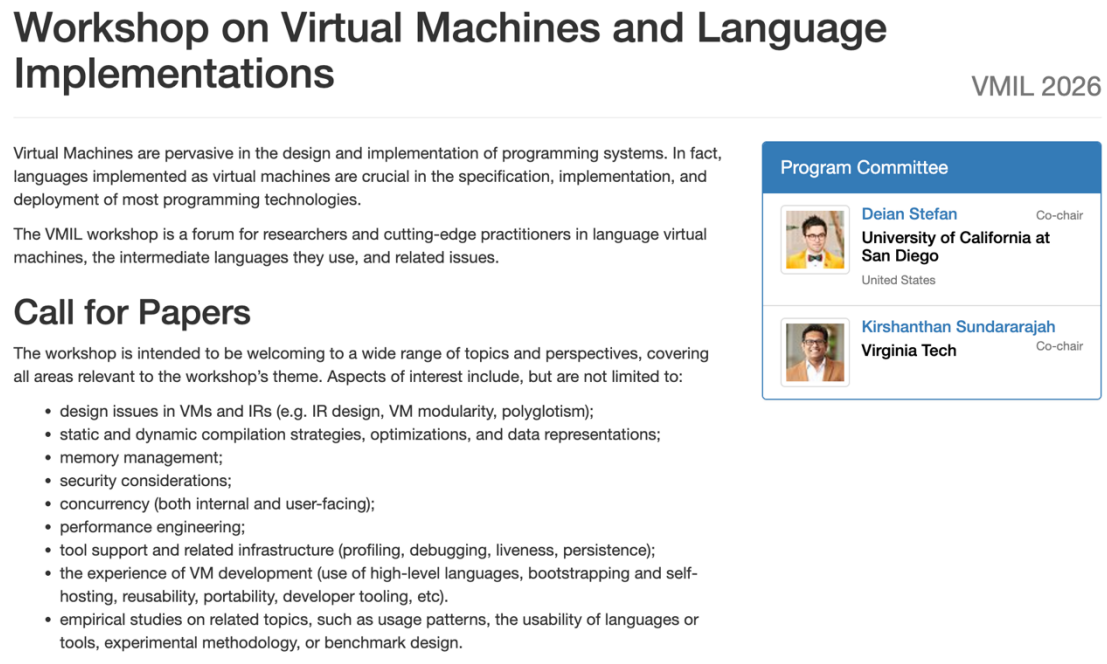
When In Doubt: Get Early Feedback

Experimental Setups

- brief submissions
- describe the experimental setup
- no results

Next opportunity:

VMIL’26



Workshop on Virtual Machines and Language Implementations

VMIL 2026

Virtual Machines are pervasive in the design and implementation of programming systems. In fact, languages implemented as virtual machines are crucial in the specification, implementation, and deployment of most programming technologies.

The VMIL workshop is a forum for researchers and cutting-edge practitioners in language virtual machines, the intermediate languages they use, and related issues.

Call for Papers

The workshop is intended to be welcoming to a wide range of topics and perspectives, covering all areas relevant to the workshop's theme. Aspects of interest include, but are not limited to:

- design issues in VMs and IRs (e.g. IR design, VM modularity, polyglotism);
- static and dynamic compilation strategies, optimizations, and data representations;
- memory management;
- security considerations;
- concurrency (both internal and user-facing);
- performance engineering;
- tool support and related infrastructure (profiling, debugging, liveness, persistence);
- the experience of VM development (use of high-level languages, bootstrapping and self-hosting, reusability, portability, developer tooling, etc).
- empirical studies on related topics, such as usage patterns, the usability of languages or tools, experimental methodology, or benchmark design.

Program Committee

Deian Stefan — Co-chair
University of California at San Diego
United States

Kirshanthan Sundararajah — Co-chair
Virginia Tech

A Starting Point of Relevant Literature

Virtual Machine Warmup Blows Hot and Cold
E. Barrett, C. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. *Proc. ACM Program. Lang., 1 (OOPSLA) ACM* (*2017*) DOI: 10.1145/3133876

Statistically Rigorous Java Performance Evaluation
A. Georges, D. Buytaert, and L. Eeckhout. *OOPSLA '07. ACM* (*2007*) DOI: 10.1145/1297027.1297033

Rigorous Benchmarking in Reasonable Time
T. Kalibera, and R. Jones. *ISMM'13. ACM* (*2013*) DOI: 10.1145/2491894.2464160

Producing Wrong Data Without Doing Anything Obviously Wrong!
T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. *ASPLOS '09. ACM* (*2009*) DOI: 10.1145/1508284.1508275

Relative Factors in Performance Analysis of Java Virtual Machines
D. Gu, C. Verbrugge, and E. Gagnon. *VEE'06. ACM* (*2006*) DOI: 10.1145/1134760.1134776

Scientific Benchmarking of Parallel Computing Systems: Twelve ways to tell the masses when reporting performance results
T. Hoefler, and R. Belli. *SC'15, ACM* (*2015*) DOI: 10.1145/2807591.2807644

Tools Are Great, But!

- Reflect on the offered benchmarks
  - Are they a subset? Why?
- Reflect on the offered measuring approach, and methodology
  - Does it fit my needs?
  - Can I explain it?

If the Tool is a Blackbox, choose something else!