GraalVM Compiler Benchmark Results Dataset (Data Artifact)

Lubomír Bulej
Vojtěch Horký
Michele Tucci
Petr Tůma
first.last@d3s.mff.cuni.cz
Charles University
Faculty of Mathematics and Physics
(and Computer Science)
Prague, Czechia

François Farquet
David Leopoldseder
Aleksandar Prokopec
first.last@oracle.com
Oracle Labs
Zürich, Switzerland

ABSTRACT

Systematic testing of software performance during development is a persistent challenge, made increasingly important by the magnifying effect of mass software deployment on any savings. In practice, such systematic performance evaluation requires a combination of an efficient and reliable measurement procedure integrated into the development environment, coupled with an automated evaluation of the measurement results and compact reporting of detected performance anomalies.

A realistic evaluation of research contributions to systematic software performance testing can benefit from the availability of measurement data that comes from long term development activities in a well documented context. This paper presents a data artifact that aggregates more than 70 machine time years of performance measurements over 7 years of development of the GraalVM Compiler Project, aiming to reduce the costs of evaluating research contributions in this and similar contexts.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Empirical software validation**.

KEYWORDS

benchmark, compiler, dataset

ACM Reference Format:
Lubomír Bulej, Vojtěch Horký, Michele Tucci, Petr Tůma, François Farquet, David Leopoldseder, and Aleksandar Prokopec. 2023. GraalVM Compiler Benchmark Results Dataset (Data Artifact). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra,Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3585025

1 MOTIVATION

Recent research activities in systematic testing of software performance during development [7, 8, 10, 11, 13, 14, 16–18] tackle challenges ranging from the efficiency of the measurement procedure to the reliability of the measurement evaluation. Validation efforts in such research activities are often significant, requiring the researchers to collect performance measurements across a significant history of software versions. With current software stacks requiring relatively long measurement durations [21] it may take years of machine time to collect sufficient measurements, and the validation may end up being restricted for pragmatic reasons such as cost or publication deadlines.

To help improve the validation efforts and support research into software performance testing, we present a data artifact that aggregates more than 70 machine time years of performance measurements over 7 years of development of the GraalVM Compiler Project [4]. The measurements are collected using standard benchmarks executing on the Java Virtual Machine (JVM), with a measurement procedure that avoids major sources of measurement disruptions and provides sufficient information for robust performance evaluation (for example compiler activity tracking for warmup filtering, or multiple workload executions for capturing non-deterministic compilation behavior). The data artifact has been used in [5], and should be suitable for research into performance change detection methods, efficient measurement planning techniques, and related problems.

2 MEASUREMENT EXPERIMENT

The main goal of the measurement experiment is to collect sufficient measurements to identify performance anomalies (both regressions and improvements in software performance) occurring due to development activities (commits) in the GraalVM Compiler Project.

The GraalVM Compiler was originally conceived as a just-in-time compiler that compiles Java bytecode into optimized machine code. It is integrated into the tiered compilation framework [12] of the Java Virtual Machine. When configured to do so, the Java Virtual Machine invokes the GraalVM Compiler to perform profile guided compilation of hot methods, identified and profiled while executing in the interpreter or as native methods compiled by lower compilation tiers.

With just-in-time compilation, the compiler and the application compete for the same execution resources (processor, memory). The performance of the compiler (how long it takes to compile the given bytecode) therefore becomes tied to the performance of the application (how long it takes to run the compiled native code). In general, compiling methods early or fast can imply reaching

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0072-9/23/04.
https://doi.org/10.1145/3578245.3585025

peak performance in shorter time, however, much more complex relationship may exist because the application timing feeds back into the hot method selection and the profile guided compilation.

More recent versions of the GraalVM Compiler also support static (ahead-of-time) compilation of Java applications into native binaries. Depending on the configuration, the compilation may still be profile guided, but the code to be compiled is identified and profiled during a short execution of the application prior to the compilation.

Workloads. We use well-known benchmark workloads as applications whose execution characterizes the compiler performance. These include the DaCapo [6] and ScalaBench [20] suites, a non-compliant modification of the SPECjvm2008 suite [1], and the Renaissance suite [19]. This choice seeks to avoid relatively small workloads (such as microbenchmarks sometimes available with specific applications [15]), which may stray far from typical compiler workloads, and relatively large workloads (such as applications that require starting multiple Java Virtual Machines), which tend to complicate automated execution and consume significant resources.

We have modified the workload harnesses of the benchmark suites to support an execution scenario where the benchmark workload is repeated as many times as needed to reach a preset total execution duration, and where the time of each repetition is recorded and reported as the primary benchmark metric. Where multiple workload sizes were available, we have picked the one which yielded a repetition time in the 1 s to 10 s range if possible. As is common with benchmark suites whose lifetime spans multiple years, we have had to exclude workloads that were not compatible with the Java Virtual Machine environment used by the compiler.

Planning. Lacking a priori information on the performance of the compiler and the application leads us to a conservative measurement experiment design, where the benchmark applications are executed repeatedly (to address non-deterministic compiler behavior) and available performance information is recorded in its entirety for subsequent processing. For each execution with just-in-time compilation, we pick a random total execution duration of 5 min to 10 min, which should typically suffice to get past major warmup artifacts while providing some variability in each execution (we do not require that stable performance is reached, just that enough compilation takes place so that final repetitions tend to execute code produced by the compiler). For executions with static compilation, where warmup artifacts are mostly confined to the first repetition, we simply set the total execution duration to 1 min.

We require a preset minimum of 33 executions for a particular benchmark application on a particular compiler version before computing confidence intervals for the performance metrics (ideally, the minimum number of executions would be tailored to each workload, however, arbitrary performance changes are possible from one compiler version to the next, and lacking a reliable procedure to calibrate the number as the compiler develops, we have started with a pragmatic constant instead). More executions are performed when the confidence intervals for mean performance are too wide to decide on potential performance anomalies, until a preset maximum (to prevent measuring the same benchmark application with the same compiler version for too long when the measurements fluctuate excessively).

With multiple compiler versions (commits) appearing daily, measuring each version with all benchmarks in all configurations of interest would consume resources excessively. Instead, we always measure the most current version of the compiler with each benchmark, and bisect towards older versions when a performance anomaly is detected. We consider only the merge commits in the main development branch of the compiler.

Metrics. Although any and all performance metrics are considered potentially interesting (so far as they can inform the development process), our focus is on the peak performance of the code produced by the GraalVM Compiler. To characterize the workload execution performance, we record the wall clock time and the aggregate thread execution time of each benchmark workload repetition. Additionally, we record selected hardware performance counters and selected events in JVM related to garbage collection and compilation; again for each workload repetition.

2.1 Platform Configuration

We collect measurements on two dedicated clusters of blade servers, where multiple machines with identical hardware and software configuration help increase the measurement capacity while preserving measurement comparability. The two hardware configurations are:

- Intel Xeon E5-2620 V4 (2100 MHz, 8 cores, 20 MB cache) with 64 GB RAM (2133 MT/s), and
- Intel Xeon E3-1230 V6 (3500 MHz, 4 cores, 8 MB cache) with 32 GB RAM (2400 MT/s).

We take common precautions to reduce measurement variance, including disabling hardware multithreading and power management (while these settings can influence application performance, we believe most compiler performance anomalies are not likely to depend on these features being active). To detect possible hardware issues, we log the processor and chipset temperature sensors on the measurement machines.

On the software side, we use the standard Fedora Linux distribution, with updates performed only when compatibility issues prevent running the benchmarks or the compiler. Currently, the measurement machines run Fedora Linux 35 with kernel 5.16 and glibc 2.34. We note that the exact software configuration is not critical, as long as the configuration changes are rare enough so that measurements of consecutive compiler versions remain mostly comparable. A hash of the complete software configuration is recorded with each measurement to permit detecting configuration changes between measurements.

2.2 Measurement Procedure

The benchmark applications and compiler versions to execute are assigned to the measurement machines one execution at a time on a first come first serve basis. Together with the random choice of the total execution duration, this effectively results in random assignment of executions to measurement machines.

To avoid influencing one execution with a profile from another, we launch a new Java Virtual Machine instance with each just-in-time compilation measurement. Analogously, we compile the benchmark application into a new native binary with each static compilation measurement.

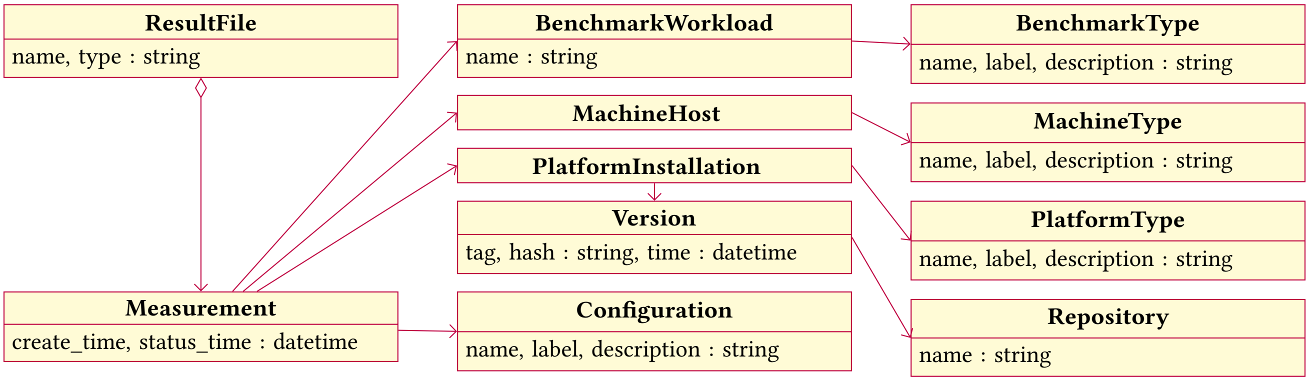
ResultFile
name, type : string
BenchmarkWorkload
name : string
BenchmarkType
name, label, description : string
MachineHost
MachineType
name, label, description : string
PlatformInstallation
Version
tag, hash : string, time : datetime
PlatformType
name, label, description : string
Measurement
create_time, status_time : datetime
Configuration
name, label, description : string
Repository
name : string

Figure 1: Metadata types used in the result archive.

```
measurement
`-- 81
    `-- 27
        `-- 28
            `-- 9282781
                |-- config-hash.txt
                |-- default.csv
                |-- metadata
                |-- platform-command.txt
                |-- platform-stderr.txt
                ...
```

Figure 2: Directory structure storing a single measurement.

Each execution starts with optionally unpacking the required artifacts (compiler, benchmark). To prevent file system activity from interfering with measurements, all pending writes are flushed and the file system cache is populated with the benchmark and compiler files prior to measurement. Because some of the benchmark workloads are sensitive to regular patterns in physical memory allocation, we also shuffle the state of the kernel page allocator by requesting and returning physical memory in random order [22]. Finally, we make sure the system is idle (all processors together are reported idle at least 99 % of the time for at least 1 s by the kernel).

Depending on actual measurement configuration, final precautions taken during benchmark execution may include disabling dynamic heap sizing and forcing garbage collection between individual benchmark workload repetitions.

2.3 Result Format

We publish the measurement results as a collection of result files annotated with metadata describing the compiler version, benchmark workload, and experiment configuration used with each measurement. A diagram of the metadata types and their relationships is shown in Figure 1.

The individual result files are stored in a directory structure under `measurement`, with each `Measurement` occupying a single directory with a unique numerical name that stores the individual `ResultFile` entries. To avoid storing too many `Measurement` directory entries in the top level `measurement` directory, a three-level structure is created as shown in Figure 2.

The exact composition of each `Measurement` directory depends on the benchmark workload and experiment configuration. To facilitate unified result processing, a selected subset of the results is also stored (redundantly) in a `default.csv` file. The file contains one row for each benchmark workload repetition, the typical columns of this file are described in Table 1. Other typical result files are described in Table 2. In addition to the individual `ResultFile` entries, each `Measurement` directory also contains a `metadata` file – this is a JSON file that references the remaining metadata from Figure 1 using unique numerical identifiers, each identifier can be looked up in the corresponding `metadata` file that is a dictionary associating the identifier with the attributes from Figure 1.

3 DATASET USAGE GUIDELINES

The measurement results, together with example processing scripts, are made available under the CC BY 4.0 license at [2]. The measurement data can also be viewed online [3] with focus on current measurements and performance anomalies.

The measurement procedure was designed to preserve measurement comparability as much as is technically reasonable, and the measurement results should be fit for any use where a history of benchmark results is needed. Still, caution should be taken when drawing conclusions, subject to the limitations outlined below.

Development. The measurements were collected using development versions of the experiment framework, the compiler, and (sometimes) the benchmarks. Measurements for some combinations of platforms, benchmarks, and metrics were not collected at some periods – if experiment continuity is needed, it should always be tested explicitly using measurement timestamps. The results reflect the development status of the components involved and are therefore not indicative of product performance.

Warmup. The measurements collect a preset number of benchmark workload repetitions, with no regard for whether the values were stable enough at the final repetition. Benchmarks may take a very long time to stabilize or not stabilize at all [21], care must be taken to identify warmup artifacts where needed. In our own evaluation, waiting for a window of reduced compilation activity, as indicated by the reported compiler thread execution duration, provided reasonably reliable results.

Profiles. Profile driven compilation is non-deterministic in principle. As a result, a single execution is not necessarily representative

Table 1: Typical columns present in the default.csv result files.

Column	Unit	Content
total_ms	ms	The duration from the JVM start to the end of the current repetition.
iteration_time_ns	ns	The duration of the current repetition measured using wall clock time.
process_cpu_time_ns	ns	The aggregate thread execution duration of the current repetition.
ref_cycles	cycles	The ref_cycles hardware event count of the current repetition. Also for the instructions, branch_instructions, branch_misses, cache_references, cache_misses, and RAPL package energy events.
JVM_COMPILATIONS	events	The count of just-in-time compilation events reported by the JVM.
compilation_time_ms	ms	The aggregate execution duration of the compiler threads in this repetition.
compilation_total_ms	ms	The aggregate execution duration of the compiler threads from the JVM start.
jmx_memory_used_size	bytes	The size of the used heap at the end of the current repetition.
jmx_memory_used_delta	bytes	The change of the value above during the current repetition.
jmx_memory_old_collection_count	events	The number of old collections at the end of the current repetition. Analogously for the value change during the current repetition, and for the young collections.
jmx_memory_old_collection_total_ms	ms	The aggregate duration of old collections since the JVM start. Analogously for the value change during the current repetition, and for the young collections.

Table 2: Typical result files present in a Measurement directory.

Name	Type	Content
default	CSV	Subset of results extracted in unified format.
default	text or JSON	Complete results as text or JSON if produced by the particular benchmark.
platform-stdout	text	Standard output produced when running the benchmark.
platform-stderr	text	Error output produced when running the benchmark.
platform-command	text	Command line used to run the benchmark.
compiler-stats	JSON	Compilation statistics from static compilation configurations.
config-hash	text	An SHA256 hash of a sorted list of all installed software packages.
sensors-before	text	A dump of the available hardware sensor readings (mostly temperature) before the JVM start.
sensors-after	text	A dump of the available hardware sensor readings (mostly temperature) after the JVM exit.
sensors-log	CSV	Readings of the processor temperature sensors extracted from above.

of the average performance observable with the same benchmark workload and compiler version across multiple executions. Where applicable, results from multiple executions should be used when evaluating performance. In our own evaluation, using results from consecutive compiler versions to estimate performance variability between executions was a reasonable remedy when enough executions of a particular compiler version were not available.

Timeline. Because our measurements are planned, the measurement history does not necessarily cover all existing versions (commits), and different versions may have been measured using different subsets of benchmarks and configurations. Because planning focuses on locating performance anomalies, measurements are more frequent around versions that have exhibited such anomalies. For the community edition of the GraalVM Compiler, a complete version history is available in the project repository [4]. Recent results provide the repository commit hashes, older results may contain build identifiers instead of commit hashes, in that case the commit timestamp may be used to identify the nearest preceding merge commit with reasonable accuracy. For the enterprise edition of the GraalVM Compiler, a complete version history is not available.

Snapshots. The measurements were collected over multiple years and, due to measurement planning, not necessarily in the same order as the compiler versions. Should a snapshot of the experiment results at certain point in history be needed, each measurement is annotated with start (create_time) and finish (status_time) timestamps. These timestamps can be filtered appropriately.

4 RELATED ARTIFACTS

Research artifacts that describe the performance history of a software project across multiple years are still rare. Most recently, the ICPE 2022 Data Challenge has published the data from the MongoDB performance testing system [9]. In the non-legacy dataset, the artifact focuses on database specific metrics (query throughput, peak query latencies, I/O operation counts and volumes, and so on) collected for 4100 task-test pairs. Each time series included in the artifact contains one measurement per version (commit), the average length of the time series is a bit below 28 with the commit dates ranging across 2 years. Apart from size, major differences compared to our artifact include use of cloud vs dedicated machines, single vs multiple values per workload execution, single vs multiple workload executions for the same version (commit), and the choice of metrics.

REFERENCES

[1] 2008. SPECjvm2008 Project Home Page. https://www.spec.org/jvm2008.

[2] 2023. GraalVM Benchmark Results Artifact. https://zenodo.org/communities/graalvm-compiler-benchmark-results.

[3] 2023. GraalVM Benchmark Results Viewer. https://graal.d3s.mff.cuni.cz.

[4] 2023. GraalVM Project Home Page. https://www.graalvm.org.

[5] Milad Abdullah, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Vojtěch Horký, and Petr Tůma. 2022. Reducing Experiment Costs in Automated Software Performance Regression Detection. In *Proceedings of SEAA 2022.*

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of OOPSLA 2006.* https://doi.org/10.1145/1167473.1167488

[7] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2022. PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits. *IEEE Transactions on Software Engineering* 48, 5 (May 2022), 1529–1544. https://doi.org/10.1109/TSE.2020.3023955

[8] David Daly. 2021. Creating a Virtuous Cycle in Performance Testing at MongoDB. In *Proceedings of ICPE 2021.* https://doi.org/10.1145/3427921.3450234

[9] David Daly. 2021. MongoDB Benchmark Results Artifact. https://doi.org/10.5281/zenodo.5138516

[10] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2017. Perphecy: Performance Regression Test Selection Made Simple but Effective. In *Proceedings of ICST 2017.* https://doi.org/10.1109/ICST.2017.17

[11] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests: Are We There Yet?. In *Proceedings of ICSE 2020.* https://doi.org/10.1145/3377811.3380351

[12] Tobias Hartmann, Albert Noll, and Thomas Gross. 2014. Efficient Code Management for Dynamic Multi-Tiered Compilation Systems. In *Proceedings of PPPJ 2014.* https://doi.org/10.1145/2647508.2647513

[13] Henrik Ingo and David Daly. 2020. Automated System Performance Testing at MongoDB. In *Proceedings of DBTEST 2020.* https://doi.org/10.1145/3395032.3395323

[14] Christoph Laaber, Harald C. Gall, and Philipp Leitner. 2021. Applying Test Case Prioritization to Software Microbenchmarks. *Empirical Software Engineering* 26, 6 (Sept. 2021), 133. https://doi.org/10.1007/s10664-021-10037-x

[15] Christoph Laaber and Philipp Leitner. 2018. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. In *Proceedings of MSR 2018.* https://doi.org/10.1145/3196398.3196407

[16] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of ICPE 2017.* https://doi.org/10.1145/3030207.3030213

[17] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of ISSTA 2017.* https://doi.org/10.1145/3092703.3092725

[18] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2021. Identifying Software Performance Changes across Variants and Versions. In *Proceedings of ASE 2020.* https://doi.org/10.1145/3324884.3416573

[19] 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 PLDI 2019.* 17. https://doi.org/10.1145/3314221.3314637

[20] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of OOPSLA 2011.* https://doi.org/10.1145/2048066.2048118

[21] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards Effective Assessment of Steady State Performance in Java Software: Are We There Yet? *Empirical Software Engineering* 28, 1 (Nov. 2022), 13. https://doi.org/10.1007/s10664-022-10247-x

[22] Petr Tůma. 2018. Frame Allocation Randomizer Project Home Page. https://github.com/d-iii-s/frame-randomizer.