From Reproducibility Problems to Improvements: A journey

Holger Eichelberger, Aike Sass, Klaus Schmid
{eichelberger, schmid}@sse.uni-hildesheim.de, sassai@uni-hildesheim.de
University of Hildesheim, Software Systems Engineering, 31141 Hildesheim, Germany

Abstract

Reproducibility and repeatability are key properties of benchmarks. However, achieving reproducibility can be difficult. We faced this while applying the micro-benchmark MooBench [5] to the resource monitoring framework SPASS-meter. In this paper, we discuss some interesting problems that occurred while trying to reproduce previous benchmarking results. In the process of reproduction, we extended MooBench and made improvements to the performance of SPASS-meter. We conclude with lessons learned for reproducing (micro-)benchmarks.

1 Introduction

A Benchmark is a standardized software system that aims at measuring the performance of a System Under Test (SUT) [5]. Benchmarks shall be representative, repeatable, robust, fair, simple, scalable, comprehensive and portable [5]. Repeatability requests statistically equivalent results when the benchmark is repeated in the same setup [3, 5]. Further, a benchmark shall be reproducible [3], i.e., other parties observe similar results in the same (or a similar) setup. Reproducibility is fundamental to objective performance comparisons and (scientific) validation of results.

SPASS-meter [4] is a flexible resource monitoring framework, which supports efficient online aggregation of raw monitoring data for user-defined components. In [5], Waller presents a performance evaluation of SPASS-meter using the response time micro-benchmarking framework MooBench. Micro-benchmarks focus on individual, basic concepts of the SUT [5], while macro benchmarks aim at the entire SUT. The results measured by Waller indicated continuous response time fluctuations caused by SPASS-meter. To understand the cause(s) for these fluctuations, we conducted a more detailed performance analysis utilizing MooBench to identify the root cause(s).

In this paper, we discuss problems of reproducing results for the same benchmark and the same SUT in technically similar setups, here applying MooBench to SPASS-meter on machines of our group. Despite significant efforts, our experiments show that we were not able to reproduce the original results. Further, our results indicate different performance issues. For explaining the issues, we use an extension of MooBench for simultaneous benchmarking of response time and memory usage. This allows us to explain the observed issues. While benchmarking is typically (a late) part in the software lifecycle [2], we apply micro-benchmarks for debugging performance issues.

We contribute a discussion of problems that occurred while trying to reproduce original benchmarking results and lessons learned for improving the reproduction. We discuss how micro-benchmarking, in our case of two performance dimensions can be utilized to explain and solve performance issues.

In Section 2, we summarize the original results. We discuss our experiments on reproducing the original results and the issues that we found in Section 3. In Section 4, we derive potential causes, apply the extended MooBench to identify the root causes and to improve SPASS-meter. In Section 5 we conclude.

2 Original Results

We base our performance analysis of SPASS-meter on the results reported in [5]. There, Waller applies MooBench to SPASS-meter for benchmarking the monitoring of a test method of recursion depth 10 for 2.000.000 calls. The first half of the executions is discarded as warm-up phase. The experiment is executed 10 times using a fresh JVM for each repetition.

Waller conducted the experiments on a Dell X6270 Blade Server with two Intel Xeon 2.53GHz E5540 Quadcore processors and 24 GByte RAM running Debian Linux with kernel 3.2.57, some libraries (libc, libdl and libpthread) upgraded to version 2.15 and an Oracle Java 64-bit Server JVM version 1.7.0_55 with 4 GBytes heap space. We will call this *setup S1*.

Figure 1 depicts the results of the experiment for the instrumentation libraries supported by SPASS-meter, javassist and ASM. Waller noted the erratic behavior of the response time and believes that this is caused by the online analysis of SPASS-meter rather than just-in-time compilation or garbage collection.

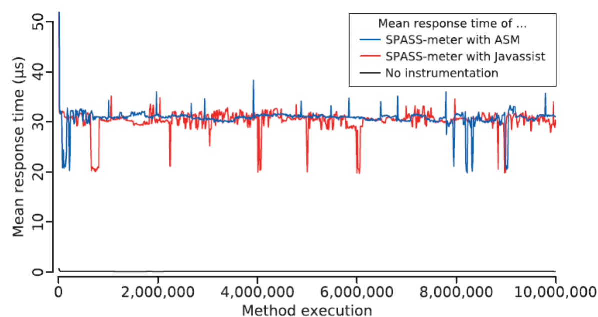

Figure 1: Response time results from [5].

3 Reproducing the Original Results

The first step of our performance analysis was to reproduce the results from Section 2. We aim at a similar behavior for identifying the reasons of the fluctuations. However, no hardware with the same specification as in setup S1 was available to us. As mitigation, we applied an iterative process to narrow down the setup and the differences to S1, starting with the most similar machine available to us using also other machines if the original results cannot be reproduced.

We started with the machine that we used for the evaluation of SPASS-meter in [4], a slightly more powerful machine than in S1. *Setup S2* consists of a Dell Optiplex 790 with an Intel Core i5-2500 CPU 3.30 GHz Quadcore processor and 8 GB RAM. For an initial impression, we utilized the (archived) software installation from [4], aiming at a more exact installation in a later iteration. We utilized a Ubuntu Linux 12.04.1 LTS 64 bit server version (updates disabled, no virus scanner/graphical user interface) with versions of JVM, SPASS-meter and MooBench as in setup S1.

The middle plot of Figure 2 depicts the results. In S2, the average response time is more than 10 μs better than in S1 (Figure 1). Even after several repetitions of the experiment, we were not able to reproduce the fluctuations from Figure 1. S2 did not only produce significantly fewer spikes, it did not show any of the downward-spikes visible in Figure 1. However, the spikes we got were much larger and as part of the reproduction effort, we noticed that when comparing multiple result sets without changing the setting, the spikes move randomly along the x-axis.

For excluding environment differences as root causes, we tried to reproduce the software from setup S1 as accurately as possible on our most similar machine to setup S1 besides S2. In *setup S3*, we use a HP ProLiant ML150 with Intel Xeon 5130 Dualcore processor and 6 GBytes RAM. According to the processor specification, S2 should be three times faster than S3. As in S1, we used a Debian 7.5 system and upgraded the libraries. As the required libraries of version 2.15 [5] were not available from the Debian archive, we installed the closest available version 2.23.

The top plot in Figure 2 depicts a representative result for S3. Also in S3, we were not able to replicate the significant fluctuations from S1. Although we used a different operating system version than in S2, the results are similar to S2 and the response time is, as expected, roughly three times higher. One may expect a linear scaling of the S3 spikes along with the slower processor, e.g., around 90μs, but also the spikes may be of the same size and just disappear in the higher average response time. The rather huge spikes we see on S2 with ASM were surprising to us.

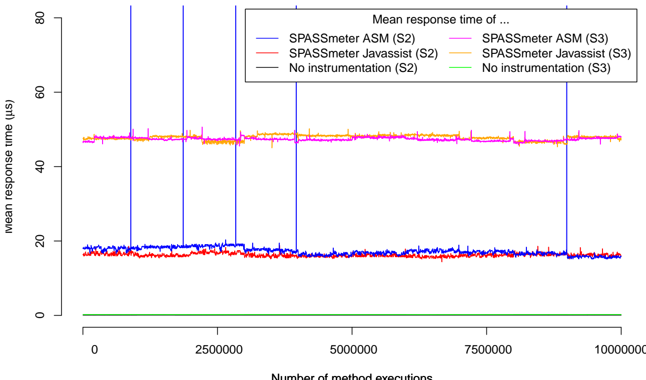

Figure 2: Average results for S2 (middle) and S3 (top).

We believe that installing Debian 7.5 also in S2 would not lead to results that are closer to S1.

4 Analyzing Performance Outliers

While we were not able to reproduce the significant fluctuations shown in Figure 1, the spikes and smaller fluctuations exhibited in setup S2 and S3 raised our concern. Thus, we tried to identify the root cause of these performance outliers. This happened in two phases: (1) we tried to identify potential triggers for the spikes, modify the setup or SPASS-meter and perform the benchmarks with MooBench to analyze the effects. (2) we extended MooBench for memory analysis in order to identify whether the spikes where correlated e.g., with garbage collection.

Due to our internal knowledge about SPASS-meter, we hypothesized first several potential causes attributed to the hardware, the operating system, the JVM, MooBench or SPASS-meter. Examples are operating system memory swapping, JVM garbage collection, the MooBench benchmark itself, the instance pooling in SPASS-meter or, as Waller notes, the online aggregation of SPASS-meter. However, none of the changes in the settings for swapping, garbage collection or the benchmark provided any insights into the origins of the spikes. Although explicit object pooling does not seem to be beneficial on modern JVMs [1], we equipped SPASS-meter with optional object pools for performance experiments. Disabling the object pools did not significantly affect the response time spikes. Finally, we systematically disabled parts of response time recording functionality of SPASS-meter line-by-line to narrow down the cause. As a result, it seems that a JMX call for obtaining the CPU time consumption of a thread caused most of the spikes.

Based on the JMX hypothesis, we aimed at finding the reason through micro-benchmarking more performance dimensions. As a first step, we extended the benchmark for monitoring memory consumption. Similar to measuring the response time, we obtained the overall JVM memory use before and after executing the test method of the benchmark and recorded the difference. We also collected the number of garbage collections before and after execution to identify the points in time when garbage collection happens. The results show that each spike corresponds to a garbage collection after a phase of increasing memory use. Profiling showed that the memory use can be attributed to the JMX call, but also to a high number of list elements created as part of the internal event processing.

As a resolution, we bypassed the JMX call by directly calling the underlying native implementation in an own library using internal JVM interfaces and enabled in-

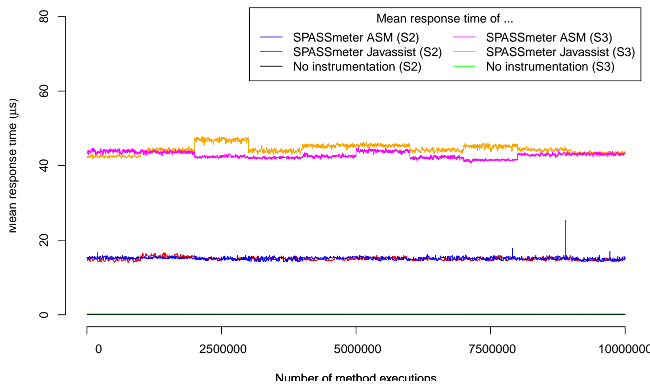

Figure 3: Results for the improved SPASS-meter.

stance pooling for the internal event processing. Figure 3 depicts the benchmarking results of the improved version of SPASS-meter in setting S2. Additional experiments indicate that the remaining spikes may be indicated by thread synchronization issues. Macro benchmarking with SPECjvm2008 as a follow-up of [4] shows that we reduced the response time monitoring overhead of SPASS-meter by half.[1]

5 Conclusions

Reproducing benchmarks is important to compare systems in similar settings and to validate scientific results. This can be difficult, in particular if the required hardware is not available in an identical setup or the software environment cannot be replicated exactly for some reason. While we were not able to reproduce the original results faithfully, micro-benchmarking SPASS-meter with MooBench indicated spikes in response time. Although such spikes could be interpreted as random statistical deviations of normal program execution, they can also indicate performance issues, in our case symptoms in response time caused by garbage collection. We identified a cause for the spikes using an extended version of MooBench for simultaneous micro-benchmarking of two performance dimensions and improved the performance of SPASS-meter, but more optimization work remains to be done.

Based on our experience, we conclude some lessons learned on reproducing micro-benchmarks: 1) If benchmarks record response times in micro-seconds, small differences in the setup can lead to huge differences in results, but also small fluctuations may occur, e.g., due to operating system activities. 2) Replicating the setup of the operating system requires detailed information on the installed components and their configuration. The performance engineer must prevent unintended updates and replicate the configuration of the original system. A detailed installation protocol (as part of standardized benchmarking processes [2]) with a listing of all component versions would be helpful. Also an archive (or downloadable boot-disk) of the entire installation would be even preferable as some versions of components may become unavailable over time (as happened to us). 3) Figures illustrating benchmark analysis results can be misleading. In our analysis we found that the scripts performed averaging of the data, but the actual parameters used remained unclear, but had a huge impact on the final figures. Thus, replicating the analysis requires the raw benchmark data, experiment protocols as well as the scripts/sheets used during the analysis, ideally together with the figures in a repository as a labeled release. It is also important to publish script parameter settings, e.g., for averaging or smoothing data as well as statistical summaries of the obtained time series including information such as minimum, maximum, and 95%-median for judging outliers. 4) Automating experiments as far as possible, ideally ranging from benchmarking to data analysis and archiving even intermediary results can help reproducing and understanding results as it simplifies the replication process and makes it less error-prone. 5) For an exact replication identical hardware is required. Small hardware differences can lead to fluctuations of the response time recording, but also hardware considered similar enough may lead to significant differences. Here, publicly available benchmarking hardware, e.g., in a cloud or standardized virtualized benchmarks may help. Moreover, benchmarks reports should include results for different hardware setups.

We believe that developing good practices for conducting and documenting benchmarks is worth a joint community effort. As contribution, all our software including the extended MooBench and the native JMX bypass, scripts and data are available online[2].

6 Acknowledgments

This work was partially supported by grant 619525 (QualiMaster) funded by European Commission grants in the 7th framework programme. Any opinions expressed herein are solely by the authors and not of the EU.

References

[1] B. Goetz and T. Peierls. *Java concurrency in practice.* Pearson Education, 2006.

[2] M. Woodside, G. Franks, and D. C. Petriu. “The future of software performance engineering”. In: *FOSE ’07.* 2007, pp. 171–187.

[3] K. Kanoun et al. “Windows and Linux Robustness Benchmarks with Respect To Application Errneous Behavior”. In: *Dependability Benchmarking for Computer Sys.* 2008, pp. 227–254.

[4] H. Eichelberger and K. Schmid. “Flexible resource monitoring of Java programs”. In: *J. Syst. Softw.* 93 (2014), pp. 163–186.

[5] J. Waller. “Performance Benchmarking of Application Monitoring Frameworks”. PhD thesis. University of Kiel, 2014.

[1] When comparing the results with [4], it must be taken into account that besides the factor 2 due to the modifications also a more recent JVM on an otherwise identical setup contributes a factor of 2, leading to a total improvement factor of 4.

[2] https://doi.org/10.5281/zenodo.165513