Using the Raspberry Pi and Docker for Replicable Performance Experiments

Experience Paper

Holger Knoche
University of Kiel
hkn@informatik.uni-kiel.de

Holger Eichelberger
University of Hildesheim
eichelberger@sse.uni-hildesheim.de

ABSTRACT

Replicating software performance experiments is difficult. A common obstacle to replication is that recreating the hardware and software environments is often impractical. As researchers usually run their experiments on the hardware and software that happens to be available to them, recreating the experiments would require obtaining identical hardware, which can lead to high costs. Recreating the software environment is also difficult, as software components such as particular library versions might no longer be available.

Cheap, standardized hardware components like the Raspberry Pi and portable software containers like the ones provided by Docker are a potential solution to meet the challenge of replicability. In this paper, we report on experiences from replicating performance experiments on Raspberry Pi devices with and without Docker and show that good replication results can be achieved for microbenchmarks such as JMH. Replication of macrobenchmarks like SPECjEnterprise 2010 proves to be much more difficult, as they are strongly affected by (non-standardized) peripherals. Inspired by previous microbenchmarking experiments on the Pi platform, we furthermore report on a systematic analysis of response time fluctuations, and present lessons learned on dos and don'ts for replicable performance experiments.

ACM Reference Format:
Holger Knoche and Holger Eichelberger. 2018. Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184431

1 INTRODUCTION

Replication of scientific work, in particular of experiments, is a prerequisite for good scientific practice, as it allows independent investigation of scientific claims [18]. Replicating experiments with human subjects is inherently difficult as the subjects, their behavior, and their opinions may differ from experiment to experiment. In contrast, due to the different nature of the subjects, technical experiments such as performance benchmarks appear to be better suited for replication. However, recent work indicates that different obstacles also exist for such experiments. While in some experiments, such as [14], distribution aspects of modern infrastructures were identified as the main inhibitor to replicability, access to similar or identical hardware prevented replication in other experiments [9].

The latter problem could be mitigated by establishing an affordable, common platform for a particular type of experiments. In our previous work, we showed that cheap commodity hardware like the Raspberry Pi allows for good replicability for the MooBench microbenchmark [16]. However, one may also have to accept (so far unexplained) high variances in response time.

The contribution of this paper is a discussion of experiences on using Raspberry Pi devices for replicable performance experiments, in particular, for different forms of benchmarks. Furthermore, we provide an systematic analysis for possible causes of variance in such experiments. We believe that our results and experiences can contribute to a community effort in creating a common platform facilitating replicable performance experiments. We aim at answering the following three research questions:

- **RQ 1** *Which types of performance experiments can be appropriately replicated using the Raspberry Pi platform?* We perform experiments of different scale on different devices and discuss the effects, e.g., the technical setup and the typical performance drop due to a low-cost compute platform.
- **RQ 2** *Can component technologies be applied on a Raspberry Pi to facilitate the replicability of performance experiments?* Combining a standardized platform with (technically) packageable experiments as envisioned by Boettiger [2] would facilitate systematic replicability. Therefore, we analyze the performance observations with and without using the Docker container platform.
- **RQ 3** *Can we identify reasons for the response time fluctuations in microbenchmarks on the Raspberry Pi reported in [16]?* In particular, we wish to investigate whether the fluctuations are caused by the devices themselves, i.e., may affect replication in general, or by (a part of) the software stack.

The paper is structured as follows: In Section 2, we introduce the technical background for the used technologies. The overall approach for setting up our experiments is described in Section 3. In Section 4, we report on the results of different micro- and macrobenchmark experiments on the Raspberry Pi platform. Driven by the experiments, Section 5 investigates causes that may impact performance experiments and their replication, in particular for the fluctuations reported in [16]. Related work is discussed in Section 6, and Section 7 concludes the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184431

2 BACKGROUND

In the following paragraphs, we provide a short technical background on the Raspberry Pi platform as well as Docker. Although there are other single-board computers available, we chose the Raspberry Pi due to its popularity, software support, and widespread availability for purchase.

2.1 Raspberry Pi

The term Raspberry Pi refers to a series of single-board computers, developed by the Raspberry Pi Foundation.[1] Originally conceived as an affordable platform for students to learn programming and computer science, the versatile devices have found many other uses in recent years.

The first models of the Raspberry Pi, the Raspberry Pi 1 Models A and B, were released in 2012. The Model A was designed for a lower retail price, and lacks certain hardware features such as on-board network connectivity. Both models have a single-core 32-bit ARMv6 processor running at 700 MHz, 16KB L1 cache, 128KB L2 cache and originally had 256 MB of RAM shared between the CPU and the GPU. In a later revision, the RAM size was increased to 512 MB. Both models use an SD card reader to host their primary storage device. Peripherals can be attached via one (Model A) or two (Model B) on-board USB 2.0 ports.

In 2015, the second generation of the Model B was released, the Raspberry Pi 2 Model B. This model is based on a quad-core ARMv7 processor running at 900 MHz with 256 KB shared L2 cache. The memory size was increased to 1 GB, and the number of on-board USB ports was increased to 4. Furthermore, the primary storage was changed from SD cards to MicroSDHC cards.

The current generation of the Model B, the Raspberry Pi 3 Model B, was released in 2016. It is equipped with a quad-core 64-bit ARMv8 processor running at up to 1.2 GHz with 512KB shared L2 cache. However, the default firmware configuration currently limits the CPU to running in 32-bit mode, and reports it to the operating system as an ARMv7 CPU. In addition to the new CPU, the Raspberry Pi 3 provides on-board wireless network and Bluetooth connectivity. Shortly after the release of the Raspberry Pi 3, revision 1.2 of the Raspberry Pi 2 was released, which is also based on the new ARMv8 CPU.

In addition to the hardware, the Raspberry Pi foundation also provides an official Linux distribution for all Raspberry Pi models, named *Raspbian*. It is based on the well-known Debian distribution, and offers a large number of software packages for the Raspberry Pi. The Raspberry Pi is also supported by several third-party vendors. In particular, Oracle provides a current Java Virtual Machine for Linux on the ARM platform, and Docker, which is further described below, added support for the Raspberry Pi in 2016 [19]. Furthermore, operating system images from third-party vendors are available, such as Ubuntu and a special edition of Windows 10.

2.2 Docker

Docker[2] is a container-based virtualization solution. In contrast to virtual machines, which use a hypervisor to provide a virtual hardware environment for guest operating systems, containers employ virtualization capabilities of the host kernel to provide a virtual system environment for applications. Scheduling and resource management for all containers is done by the host kernel, which is also responsible for keeping the containers isolated from each other.

This approach makes containers more „lightweight" than virtual machines in several ways. The absence of a guest kernel avoids the resource consumption due to the additional scheduling and resource management inside the virtual environment. Furthermore, the containers do not have to provide an entire operating system, but only their required programs and libraries, allowing for smaller images. And since no guest operating system needs to be booted or shut down, containers can usually be started and stopped very quickly. Due to these properties, containers have become very popular in the industry, as they allow for rapid resource provisioning for building highly elastic applications.

As discussed in [2], Docker has several features that make it also a promising option for replicable research. The fact that a Docker image contains all its required dependencies (except the underlying operating system) greatly facilitates replicating a software environment, and avoids common pitfalls such as wrong library versions. This enables separating individual experiments as well as running variants of an experiment, e.g., the same experiment on different operating systems or system versions. Although it is possible to build Docker images interactively, it is common practice to create images by means of a so-called Dockerfile. A Dockerfile specifies the necessary steps to build an image using a simple syntax. Thus, it provides a human-readable specification that can, for instance, be used to create variants or other derivations of an experiment.

A particularly interesting property of Docker images is that images are built „on top of" other images, i.e., Docker provides an extension mechanism for images. This mechanism further facilitates variants and extensions of experiments packaged as Docker images. Every Dockerfile must specify its *base image*, i.e., the image it is derived from, with its first instruction [8]. All operations specified by the Dockerfile are then applied on top of the base image, and the resulting image is saved at the end of the build process. To avoid unnecessary data replication, Docker employs a layered file system. Each image only stores the differences to its underlying base image, and all layers are overlaid at runtime to form the complete file system. Moreover, a command can be specified to be executed upon starting a container, i.e., not only the software but also the execution and even the analysis can be packaged in a repeatable manner. By means of environment variables, specific settings can be applied to a container without changing the image itself.

The overlay mechanism is also applied when starting a container off an image. All changes made to the file system by a container are stored in a container-specific layer atop the image, while all other images are immutable. This allows to re-use an image for multiple containers or experiments, at the cost of runtime performance due to the overlay file system.

Docker images can be deployed manually or using a repository. For the latter, Docker provides a mechanism for distributing images over a network. Images can be „pushed" to a registry and „pulled" by the Docker engine on request. By default, Docker interacts with the public *Docker Hub*[3] registry provided by Docker, Inc.

[1] http://www.raspberrypi.org
[2] https://www.docker.com/
[3] https://hub.docker.com/

3 APPROACH

In order to assess the viability of the Raspberry Pi platform for replicable performance experiments, we conducted a series of different experiments on multiple Pi devices in different configurations. The general approach is presented below, while the actual experiments are described in Section 4.

Each author bought a Raspberry Pi set from the same supplier within a time frame of two weeks. Each set comprised a Raspberry Pi 3 device by vendor element14, an 8 GB SanDisk class-4 SD card and a power supply capable of delivering 2.5 A at 5 V. These two devices will be referred to as D_1 and D_2 below; the SD cards will be referred to as C_1 and C_2. The intention behind this was to have two devices that were as similar as possible. To evaluate whether a different production lot or a potential minor revision might affect replicability, we bought a third device D_3 with the same specifications at a local electronics shop several months later. This device was from a different vendor (Allied Electronics). All three devices reported to have BCM2835 CPUs (revision a02082) in `/proc/cpuinfo`.

In a second step, we prepared a master installation images for all three devices[4]. This image is based on Raspbian Stretch Lite, which was released shortly before we conducted our experiments. Raspbian Lite is a minimal variant of the Raspbian distribution without potentially influencing components such as a graphical user interface, a virus scanner, or automated updates. We installed all necessary software to run the experiments, in particular, Oracle JDK 1.8.0_144 for the `armhf` platform, as the OpenJDK version provided by the distribution does not contain a just-in-time compiler. For investigating performance fluctuations in Section 5, where we needed a direct comparison to our previous Raspberry Pi experiments from [16], we used the same Raspbian Jessie Lite image as for the original experiments.[5]

In order to test the effect of different storage devices, we also used two class-10 SD cards, a Transcend Premium 400x (16 GB, C_3) and a SanDisk Ultra (16 GB, C_4) as well as three commodity USB hard disks, a Toshiba STOR.E ALU 2S (500 GB, H_1), a Hitachi Z7K320 (320 GB, H_2) and a TravelStar Z7K400 (500GB, H_3).

4 EXPERIMENTAL EVALUATION

In order to evaluate the replicability of performance experiments on different Raspberry Pi devices, we ran a selection of experiments, which are described in detail below. Experiments 1 and 2 are based on microbenchmarks and, thus, aim at replicability at a low level, while Experiments 3 and 4 address replicability at higher levels. It should be noted that the experiments are conducted with the aim of assessing replicability, not achieving a particularly high score in any of the benchmarks employed.

4.1 Experiment 1: Microbenchmarks using the Java Microbenchmark Harness

The Java Microbenchmark Harness[6] (JMH) is a test harness for running microbenchmarks on the Java Virtual Machine (JVM), provided by the OpenJDK team. Due to the dynamic compilation performed by the JVM, carrying out such benchmarks can be difficult, and subtle errors can happen easily. The JMH facilitates such benchmarks by automatically inserting warmup phases, forking multiple VM instances, measuring execution times, and calculating important statistical figures at the end of a benchmark run. Furthermore, the JMH provides facilities to conveniently influence the behavior of the just-in-time compiler. For instance, methods can be prevented from being inlined or even being compiled at all.

```
public void testMethod(final int depth) {
  if (depth == 0) {
    return;
  } else {
    this.testMethod(depth - 1);
  }
}
```

Listing 1: Test method for JMH microbenchmark

We used the JMH to conduct a total of six microbenchmarks. Each microbenchmark was executed 10 times with a freshly instantiated JVM, with a warmup phase of 20 seconds and a measurement phase of 20 seconds for each run. The first four benchmarks measured the throughput of calling a simple recursive method (see Listing 1) with a recursion depth of 10. This setup is similar to MooBench, the micro-benchmark we evaluated in [16], which is also used in Experiment 2 and in the analysis in Section 5. Benchmark 1 was run with default compilation, Benchmark 2 explicitly requested inlining of the test method, Benchmark 3 explicitly supressed inlining, and Benchmark 4 suppressed any compilation of the test method.

The two remaining microbenchmarks aimed at a rough comparison of the input/output (I/O) behavior of the different devices. Benchmark 5 measured the throughput of a method, which wrote four bytes of data to a file in each invocation, and synced the writes to disk every 100,000 invocations. Four bytes per invocation were chosen as to prevent excessive growth of the test file, so that the benchmarks could also be run on the SD cards. Benchmark 6 was similar, but sent the data to a remote machine via TCP.

Selected results from the microbenchmarks are shown in Table 1. As evident from comparing lines 1 and 4, there is no significant difference between the two devices D_2 and D_3 in Benchmark 1 with the same peripherals, as the confidence intervals overlap. The same is true for Benchmarks 2 and 3 (lines 5 to 8). For Benchmark 4, the confidence intervals do not overlap; however, the gap between the intervals is almost neglegible. As evident from line 2, the benchmark runs slightly slower under Docker, with a slightly higher variance. Although not shown in Table 1, the results for D_1 are rather similar.

As expected, the results from Benchmark 5 vary significantly with the storage devices; none of the peripherals was able to saturate the Pi's storage interface. Again, exchanging only the Pi devices yielded no significant difference in the results (see lines 11 and 17). Surprisingly, hard disk H_1 achieved a considerably higher mean throughput when running under Docker, however, with a much higher variance (see line 12). This behavior seems to be device-specific as it did not occur with disk H_2 (see lines 13 and 14), but was replicable in other runs. Possibly, sync requests are handled differently for the native file system and the overlay file system

[4] All (raw) material is available on https://doi.org/10.5281/zenodo.1100975
[5] https://doi.org/10.5281/zenodo.1003075
[6] http://openjdk.java.net/projects/code-tools/jmh/

Line #	Benchmark	Mean Throughput *(in invocations / s)*	99.9 % CI Throughput *(in invocations / s)*	σ *(in invocations / s)*
1	Benchmark 1 (D_2 – H_1, native)	12,322,204.022	[12,314,425.859 ; 12,329,982.186]	32,933.231
2	Benchmark 1 (D_2 – H_1, Docker)	12,299,546.551	[12,290,438.552 ; 12,308,654.549]	38,563.836
3	Benchmark 1 (D_2 – H_2, native)	12,299,680.408	[12,291,599.801 ; 12,307,761.015]	34,213.796
4	Benchmark 1 (D_3 – H_1, native)	12,314,181.630	[12,307,645.303 ; 12,320,717.957]	27,675.217
5	Benchmark 2 (D_2 – H_1, native)	12,323,493.925	[12,315,117.890 ; 12,331,869.960]	35,464.657
6	Benchmark 2 (D_3 – H_1, native)	12,328,094.938	[12,320,806.145 ; 12,335,383.730]	30,861.204
7	Benchmark 3 (D_2 – H_1, native)	6,416,150.780	[6,413,796.051 ; 6,418,505.508]	9,970.068
8	Benchmark 3 (D_3 – H_1, native)	6,417,104.725	[6,414,305.345, 6,419,904.106]	11,852.753
9	Benchmark 4 (D_2 – H_1, native)	410,968.302	[410,577.922 ; 411,358.681]	1,652.890
10	Benchmark 4 (D_3 – H_1, native)	411,604.745	[411,525.095 ; 411,684.395]	337.244
11	Benchmark 5 (D_2 – H_1, native)	553,927.284	[541,361.418 ; 566,493.149]	53,204.662
12	Benchmark 5 (D_2 – H_1, Docker)	882,408.673	[852,324.902 ; 912,492.445]	127,376.572
13	Benchmark 5 (D_2 – H_2, native)	773,246.941	[767,199.859 ; 779,294.023]	25,603.722
14	Benchmark 5 (D_2 – H_2, Docker)	699,276.759	[692,319.247 ; 706,234.271]	29,458.541
15	Benchmark 5 (D_2 – C_2, native)	491,016.010	[421,129.074 ; 560,902.946]	295,905.663
16	Benchmark 5 (D_2 – C_3, native)	682,755.400	[659,149.054 ; 706,361.746]	99,950.747
17	Benchmark 5 (D_3 – H_1, native)	548,804.364	[536,529.590 ; 561,079.138]	51,972.161
18	Benchmark 6 (D_2 – H_1, native)	195,719.580	[192,310.748 ; 199,128.413]	14,433.212
19	Benchmark 6 (D_2 – H_1, Docker)	188,548.713	[184,943.513 ; 192,153.912]	15,264.641
20	Benchmark 6 (D_2 – H_2, native)	202,397.887	[200,041.480 ; 204,754.293]	9,977.172
21	Benchmark 6 (D_3 – H_1, native)	195,727.533	[192,631.759 ; 198,823.306]	13,107.698

Table 1: Selected results from the JMH microbenchmarks (similar for device D_1)

employed by Docker, evoking this maybe even erroneous behavior of the drive.

For Benchmark 6, there were again no signficant differences between the Pi devices (see lines 18 and 21). The throughput under Docker was significantly lower (see line 19), which was to be expected due to the additional network stack of the container.

Summary: *The Raspberry Pi devices show highly replicable behavior in all microbenchmarks. In I/O-related benchmarks, the storage devices had a high influence on replicability, and one even showed highly unexpected behavior when used with Docker.*

4.2 Experiment 2: MooBench

MooBench [23] is a microbenchmark for measuring the runtime overhead of (instrumenting) monitoring frameworks such as Kieker and SPASS-meter, which inject so-called *probes* into an application to collect statistical data at runtime. By default, MooBench executes 2,000,000 calls of a recursive test method (recursion depth 10) and iterates the test 10 times. As baseline, MooBench performs a 'dry' run on the test method without any instrumentation. In [16], we applied MooBench to Kieker and SPASS-meter on a Raspberry Pi 3 platform and concluded that replicating results is possible. Here, we extend these experiments to compare benchmarks running in a Docker container against 'native' runs without Docker. To allow for comparisons of the collected data, we used the specific MooBench setup for Kieker as reported in [16], i.e., a recursion depth of 5 and 1,000,000 calls in 10 iterations.

Table 2 summarizes the collected measurements results, more precisely the data produced during the second half of the runs where the executing JVM is expected to have reached a steady state [23]. As the response time is measured by MooBench in terms of nanoseconds, which is typically rather imprecise on Java (some technical reports state fluctuations of about 400ns for Linux), we report the results here with one significant decimal place Within one type of experiment (a row in Table 2), the confidence intervals are close to the mean and differ only in a range of at maximum 11 μs for all experiments, even for Docker. In our previous work, we achieved similar results for the experiments using the external hard drive and for the corresponding class-4 SD card with a spread of 21 μs for SPASS-meter and 64 μs for Kieker. However, the narrow confidence intervals and partially high deviations also indicate fluctuations, which we will analyze in more detail in Section 5. Regarding the specific variances in Table 2, we observe that the deviations differ between native execution (6 μs for SPASS-meter, 373 μs for Kieker) and Docker (174 μs for SPASS-meter, 491 μs for Kieker). For Kieker, the deviations between native and Docker execution are rather similar. Moreover, in our previous experiment, the differences for SPASS-meter on the external hard drive were around 17 μs and 1,126 μs for Kieker, and even more than 20,000 μs for runs on the SD card. Thus, we classify the deviations for the I/O intensive Kieker experiments to be within the normal range (probably dominated by the hard drive), while for the less I/O-intensive SPASS-meter experiments, the differences may be caused by the Docker virtualization.

Summary: *The Raspberry Pi devices allow for good replication of microbenchmarks for instrumenting monitoring frameworks. This also applies to running inside Docker containers, provided that we accept a certain deviation in response time.*

Experiment	D_1, C_1, H_3			D_2, C_3, H_2			D_3, C_3, H_2		
	mean	95% CI	σ	mean	95% CI	σ	mean	95% CI	σ
Baseline	0.5	[0.5; 0.5]	0.3	0.5	[0.5; 0.5]	0.2	0.5	[0.5; 0.5]	0.4
SPASS-meter native	153.5	[153.5; 153.5]	48.9	145.0	[145.0;145.0]	50.4	151.6	[151.6; 151.7]	44.8
SPASS-meter Docker	152.0	[152.0; 152.0]	43.4	147.7	[147.6;147.8]	186.0	155.2	[155.0; 155.4]	326.5
Kieker native	121.5	[118.8; 124.3]	3,090.6	115.9	[113.6; 118.3]	2,717.2	118.6	[116.2; 121.1]	2,795.2
Kieker Docker	131.4	[128.7; 134.2]	3,142.1	123.3	[120.8; 125.8]	2,872.9	120.5	[118.2;122.8]	2,651.3

Table 2: Summary of MooBench stable state response times in μs with confidence intervals (CI) and standard deviation (σ).

4.3 Experiment 3: JPA RESTful Web Services

In order to evaluate the replicability of macroscopic experiments with multiple interacting Raspberry Pi devices, we created a simple, RESTful web service which interacts with a relational database via the Java Persistence API (JPA). We decided to build this service using Spring Boot,[7] a platform currently popular in the industry for implementing so-called microservices. For the underlying database, we used PostgreSQL 9.6.5, and Spring Data JPA was used to access the data.

The web service provided three operations, which emulated a very simplistic customer database. The first method generated a random customer entry and returned it without accessing the database at all. This method was intended to serve as a baseline to compare the results of the database-enabled operations against. The second method read an existing customer by his customer number, and the third operation changed the first and last name of a given customer in the database.

For this experiment, we used a pair of devices D_2 and D_3 with hard disks H_1 and H_2. Both Pi devices were connected to the same Gigabit ethernet switch, as was the test driver, a notebook with an Intel Core i7-4500U processor, 8 GB of RAM and a Gigabit ethernet interface. The experiment was conducted in six configurations:

(1) Web server running natively on D_2 with hard drive H_1, database running natively on D_3 with hard drive H_2
(2) Same as (1), but both services running in Docker containers
(3) Web server running natively on D_3 with hard drive H_1, database running natively on D_2 with hard drive H_2
(4) Same as (3), but both services running in Docker containers
(5) Web server running natively on D_2 with hard drive H_2, database running natively on D_3 with hard drive H_1
(6) Same as (5), but both services running in Docker containers

The database was pre-loaded with about 1 GB of data (10 million records) to prevent the server from keeping the whole dataset in memory. For the experiment, the test driver then invoked each method 200,000 times using a pool of 16 threads, and measured the response times. The first 100,000 invocations were disregarded as warm-up. Table 3 summarizes the results.

As evident from the table, switching the Raspberry devices only leads to minor changes in response time (e.g., Lines 1 and 3). Although some of the differences (e.g., Lines 7 and 9) are statistically significant, we consider them small enough to speak of good replicability regarding this experiment. The same applies to switching from running natively to running inside Docker containers. Apparently, the overhead of the virtualization is outweighed by other factors in this experiment.

As expected, swapping the hard drives has a major effect on the results on this benchmark, as the drives are heavily utilized by the database due to the size of the table and the random access pattern. This dependency on the peripherals severely limits the replicability for I/O-heavy experiments. However, we wish to highlight that this is still an improvement to replicating experiments on common PC hardware, as much fewer components are interchangeable. Thus, specifying the execution environment is greatly facilitated.

Summary: *Provided that the peripherals are identical, good replication of macroscopic experiments is possible even for I/O-heavy experiments. Although this can pose a severe limitation to replicability, the limited number of interchangeable components at least facilitates specifying the execution environment of such experiments.*

4.4 Experiment 4: SPECjEnterprise 2010

Our second experiment for assessing the replicability of macroscopic experiments on the Raspberry Pi used SPECjEnterprise 2010,[8] a well-known Java EE benchmark. Similar to our previous experiment, we used one of the Pis as the database server, while the other ran the application server. Deviating from the run rules, we deployed the supplier emulator on the same application server as the actual benchmark application. The test driver was run on the same notebook as before.

Again, we used PostgreSQL 9.6.5 as the underlying RDBMS. Before each run, all tables were dropped, re-created, and loaded with the same data. For the Docker experiments, a new database container was started for each run.

For the application server, we used GlassFish 5.0 (Build 25). Similar to the database, the server was freshly configured and deployed for each run. Due to a connectivity issue, the GlassFish Docker containers had to be run using the host's network stack instead of an own one. Apparently, the application server resolves its own host name locally and transfers the resulting IP address to the client. By default, the host name resolves to a loopback address, and the client fails to connect. The resolution can be corrected by editing the `/etc/hosts` file, however, the application server also tries to bind to the interface with the resolved IP address. This attempt always fails, as the desired ports are also claimed by the Docker daemon to forward them to the container.

[7] https://projects.spring.io/spring-boot/

[8] http://www.spec.org/jEnterprise2010/

Line #	Operation	Configuration	Mean Response Time (in μs)	99% CI Resp. Time (in μs)	σ (in μs)
1	Read customer	Web: D_2 + H_1, DB: D_3 + H_2, native	168,914.4	[167,197.2 ; 170,631.7]	210,817.8
2		Web: D_2 + H_1, DB: D_3 + H_2, Docker	165,467.7	[163,736.3 ; 167,199.1]	212,555.3
3		Web: D_3 + H_1, DB: D_2 + H_2, native	170,284.3	[168,447.3 ; 172,121.3]	225,524.0
4		Web: D_3 + H_1, DB: D_2 + H_2, Docker	179,067.3	[177,265.1 ; 180,869.4]	221,244.0
5		Web: D_2 + H_2, DB: D_3 + H_1, native	283,504.3	[280,914.7 ; 286,094.0]	317,924.2
6		Web: D_2 + H_2, DB: D_3 + H_1, Docker	275,709.3	[272,882.2 ; 278,536.4]	347,072.5
7	Create random customer	Web: D_2 + H_1, DB: D_3 + H_2, native	11,164.6	[11,081.2 ; 11,248.1]	10,241.8
8		Web: D_2 + H_1, DB: D_3 + H_2, Docker	12,368.8	[12,276.1 ; 12,461.4]	11,375.3
9		Web: D_3 + H_1, DB: D_2 + H_2, native	11,832.8	[11,744.1 ; 11,921.6]	10,897.0
10		Web: D_3 + H_1, DB: D_2 + H_2, Docker	14,136.2	[14,025.4 ; 14,247.0]	13,602.9
11		Web: D_2 + H_2, DB: D_3 + H_1, native	12,840.9	[12,743.1 ; 12,938.6]	11,999.0
12		Web: D_2 + H_2, DB: D_3 + H_1, Docker	11,674.5	[11,584.7 ; 11,764.3]	11,024.1
13	Change customer name	Web: D_2 + H_1, DB: D_3 + H_2, native	356,194.7	[354,663.2 ; 357,726.3]	188,019.6
14		Web: D_2 + H_1, DB: D_3 + H_2, Docker	354,837.6	[353,323.9 ; 356,351.3]	185,830.1
15		Web: D_3 + H_1, DB: D_2 + H_2, native	353,910.2	[352,360.9 ; 355,459.6]	190,211.0
16		Web: D_3 + H_1, DB: D_2 + H_2, Docker	381,395.7	[379,798.5 ; 382,993.0]	196,090.5
17		Web: D_2 + H_2, DB: D_3 + H_1, native	551,412.8	[549,424.2 ; 553,401.5]	244,138.3
18		Web: D_2 + H_2, DB: D_3 + H_1, Docker	551,034.6	[548,759.9 ; 553,309.2]	279,250.8

Table 3: Results of the RESTful service experiment

All tests were run with the default configuration, which consists of a 10-minute warmup phase, a measurement phase of 60 minutes, and 5 minutes of cooldown. Table 4 provides information on the response times measured for the five operations performed by the benchmark; the configurations are the same as for the previous experiment (see Section 4.3).

As evident from the table, considerable replicability of the results was achieved only in specific cases. While the response times of the Enterprise Java Beans (EJB)-based operation „Create vehicle" indicate good replicability (Lines 1–6), the response times of the web service (WS)-based variant (Lines 7–12) show substantial differences between the configurations. This also applies to the remaining, web service-based operations. It is particularly remarkable that the response times already differ significantly when swapping the Pi devices, a change which did not have any impact on the previous experiment with RESTful services. As the differences between EJB and web services were much smaller when running the database server on a desktop machine, we assume that the difference is due to different types of database accesses, but are unable to provide an explanation at this point.

Another notable observation is that the response times are considerably lower for the Docker-based Configuration 6 than for the native Configuration 5, while it is the other way around for the other configurations. This may be another occurrence of the disk-related anomaly discussed in Experiment 1, as the respective disk H_1 is used for the database in these configurations.

Besides the unexpected differences in response times, the actual throughput achieved on the Raspberry devices does not meet the requirements of the benchmark. Consequently, all runs are considered as failures by the test driver. We therefore conclude that although running enterprise benchmarks on current Raspberry devices is technically possible, the validity of the results may be questionable. However, this may change with future, more powerful models.

Summary: *Although it is technically possible to run enterprise-oriented benchmarks like SPECjEnterprise on the Raspberry Pi, the results are questionable. The devices are not powerful enough to meet the minimum requirements of the benchmark, although the benchmark is already six years old. Furthermore, the replicability of the results was very limited in our experiments.*

5 FLUCTUATION CAUSE ANALYSIS

While our micro-benchmarking experiments from Section 4.2 and [16] indicate good replicability, even the measures of the baseline show significant deviations (0.2 of 1,6 μs for the base line, factor 3 times for SPASS-meter, factor 25 for Kieker) as well as high maximum values (65 times of the mean for the baseline, 125 times for SPASS-meter, more than 13,160 times for Kieker). The raw data contains massive response time peaks as illustrated for one out of ten experiment runs from [16] in Figure 1.

We may consider these fluctuations as system-immanent, but in the context of evaluating the Raspberry Pi for replicability of experiments, it is worth performing an analysis of potential causes. Moreover, the measurements from [9][9] indicate only some dedicated response time peaks on a server machine rather than a fusillade of peaks as in our Pi experiments. However, the fluctuations that we observed did not exhibit any kind of regular pattern that we could focus on. In order to identify candidates for root causes, we performed a systematic enumeration of potential reasons. Figure 2 illustrates the mind map we obtained from analyzing the system architecture and the involved software stack. For each potential cause, we changed the setup accordingly, re-executed the MooBench experiments for SPASS-meter on D_1 and analyzed the measurements.

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

Line #	Operation	Configuration	Mean Response Time *(in s)*	99% CI Resp. Time *(in s)*	σ *(in s)*
1	Create vehicle (EJB)	Web: D_2 + H_1, DB: D_3 + H_2, native	0.235	[0.228 ; 0.242]	0.031
2		Web: D_2 + H_1, DB: D_3 + H_2, Docker	0.243	[0.236 ; 0.251]	0.030
3		Web: D_3 + H_1, DB: D_2 + H_2, native	0.266	[0.256 ; 0.275]	0.039
4		Web: D_3 + H_1, DB: D_2 + H_2, Docker	0.259	[0.250 ; 0.267]	0.036
5		Web: D_2 + H_2, DB: D_3 + H_1, native	0.293	[0.284 ; 0.303]	0.041
6		Web: D_2 + H_2, DB: D_3 + H_1, Docker	0.310	[0.300 ; 0.319]	0.041
7	Create vehicle (WS)	Web: D_2 + H_1, DB: D_3 + H_2, native	0.447	[0.411 ; 0.484]	0.156
8		Web: D_2 + H_1, DB: D_3 + H_2, Docker	0.718	[0.641 ; 0.795]	0.328
9		Web: D_3 + H_1, DB: D_2 + H_2, native	0.910	[0.807 ; 1.012]	0.436
10		Web: D_3 + H_1, DB: D_2 + H_2, Docker	1.303	[1.190 ; 1.417]	0.483
11		Web: D_2 + H_2, DB: D_3 + H_1, native	1.550	[1.457 ; 1.643]	0.396
12		Web: D_2 + H_2, DB: D_3 + H_1, Docker	0.960	[0.851 ; 1.069]	0.463
13	Purchase	Web: D_2 + H_1, DB: D_3 + H_2, native	0.750	[0.653 ; 0.847]	0.412
14		Web: D_2 + H_1, DB: D_3 + H_2, Docker	1.502	[1.273 ; 1.731]	0.972
15		Web: D_3 + H_1, DB: D_2 + H_2, native	1.991	[1.706 ; 2.276]	1.212
16		Web: D_3 + H_1, DB: D_2 + H_2, Docker	3.177	[2.844 ; 3.510]	1.417
17		Web: D_2 + H_2, DB: D_3 + H_1, native	3.830	[3.555 ; 4.106]	1.173
18		Web: D_2 + H_2, DB: D_3 + H_1, Docker	2.012	[1.708 : 2.315]	1.289
19	Manage	Web: D_2 + H_1, DB: D_3 + H_2, native	0.576	[0.522 ; 0.630]	0.229
20		Web: D_2 + H_1, DB: D_3 + H_2, Docker	0.930	[0.819 ; 1.041]	0.473
21		Web: D_3 + H_1, DB: D_2 + H_2, native	1.139	[1.004 ; 1.275]	0.576
22		Web: D_3 + H_1, DB: D_2 + H_2, Docker	1.661	[1.502 ; 1.819]	0.675
23		Web: D_2 + H_2, DB: D_3 + H_1, native	1.954	[1.817 ; 2.091]	0.582
24		Web: D_2 + H_2, DB: D_3 + H_1, Docker	1.189	[1.037 ; 1.341]	0.648
25	Browse	Web: D_2 + H_1, DB: D_3 + H_2, native	1.194	[1.066 ; 1.321]	0.543
26		Web: D_2 + H_1, DB: D_3 + H_2, Docker	2.231	[1.950 ; 2.513]	1.197
27		Web: D_3 + H_1, DB: D_2 + H_2, native	2.814	[2.451 ; 3.178]	1.546
28		Web: D_3 + H_1, DB: D_2 + H_2, Docker	4.425	[4.004 ; 4.847]	1.792
29		Web: D_2 + H_2, DB: D_3 + H_1, native	5.190	[4.845 ; 5.535]	1.466
30		Web: D_2 + H_2, DB: D_3 + H_1, Docker	2.823	[2.426 ; 3.220]	1.688

Table 4: Results of the SPECjEnterprise experiment

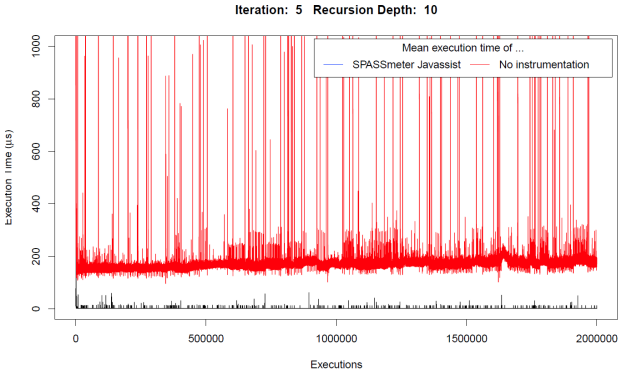

Figure 1: Response time fluctuations observed in [16].

We focused on SPASS-meter, assuming that identified causes will finally also improve the Kieker results.

We discuss now in separate sections the cause categories shown in Figure 2 starting with the 'hardware' category, and then follow a clockwise order. Within each category, we discuss the causes shown in a top-down fashion. We base our discussion on previous experiments from [16][10], but also on the new experiments. For pragmatic reasons, we performed the experiments in a different sequence, focusing first on those experiments that we considered most likely for explaining the peaks. Table 5 details the experiment sequence, the respective (incremental) base cases, descriptive statistics for the baseline and and the SPASS-meter runs, both also indicating the number of peaks. For illustrating our discussion, we count a value as a peak if it is larger than than 5 times the mean value. For the whole data set underlying Figure 1 we identified 1,155 such peaks.

5.1 Hardware

The hardware of the different Raspberry types is rather standardized as detailed in Section 2.1, i.e., the configuration spectrum for a Raspberry Pi is rather restricted compared with desktop, laptop or server machines. This restricted configuration space eases the identification of variation causes.

[10]https://doi.org/10.5281/zenodo.1003075

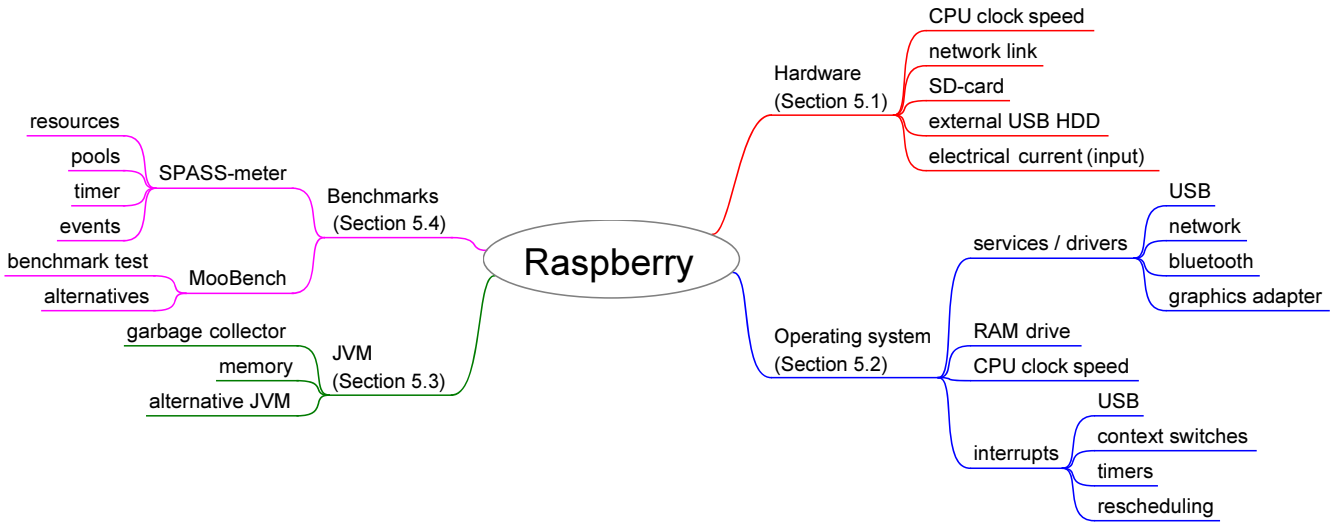

Figure 2: Cause-tree for response time fluctuations.

- The CPU of a Pi allows for changing *clock speeds*, in particular to save energy. On the Raspberry Pi platform, the Raspbian operating system takes active control over the CPU clock speed as we will detail in Section 5.2.
- The *network link* used to control the experiments was active during the experiments and may have caused superfluous interrupts. However, benchmark runs[10] with disconnected network link, background execution of the benchmarks or even an operating network connection during foreground execution showed similar response times and deviations.
- The operating system of a Raspberry device is typically installed on an *exchangeable SD-card*. The Pi sets we obtained contained class-4 SD cards supporting a minimum sequential write speed of 4MByte/s[11]. Previous experiments[10] were also run with a class-10 SD card. For SPASS-meter, the faster card led to an increase of the average response time of 5% as well as an increase of factor 7 of the response time and similar deviations. In contrast, for the I/O intensive Kieker benchmarks, the average response time dropped by 50%, the deviation by factor 2 and the maximum response time by factor 2.6. As a result, a faster SD card can lead to improvements for response time, but may not significantly influence response time peaks (similar to Table 5, Id 1).
- Instead of running the benchmarks on an SD card, we considered a potentially faster *external USB hard disk*. Although Raspberry 3 devices ship only with USB 2.0 ports, previous results [16] show that an external USB hard disk can lead to significant speedup for I/O intensive benchmarks, e.g., for Kieker around factor 4.5, but also to a slowdown, e.g., for SPASS-meter by roughly 5%. In case of speedups, deviation and maximum response time dropped, e.g., for Kieker by around 95%, but the response time peaks did not disappear (similar to Table 5, Id 1).
- The Raspberry Pi needs at least 700 mA of *electrical current*[12]. Power adapters just fulfilling this specification may affect stability and performance if additional USB devices are connected. We experienced this when replacing the shipped power adapters (2.5 A) with a 2.0 A adapter. For example, in case of the SPEC benchmark in Section 4.4 the results differed significantly. However, we can exclude this cause as the SPASS-meter experiments were conducted with the shipped adapters.

Although the storage device may significantly impact the performance, in particular for I/O intensive benchmarks, the hardware category did not lead to a clear cause for the response time peaks.

5.2 Operating system

Nowadays, an operating system consists of several layers including kernel, drivers and services, whereby each of these layers may cause fluctuations in a benchmark experiment.

- *System services* may allocate resources that cause fluctuations in the measurements. Therefore, unneeded services like window system, virus scanner or automated updates should be disabled. Such services are not included in the Raspbian versions we used for our experiments. For identifying further problematic services, we analyzed the running processes and

[11] https://www.sdcard.org/developers/overview/speed_class/

[12] https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md

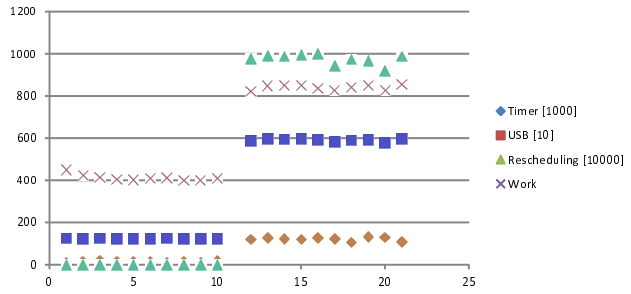

Figure 3: Interrupts during MooBench executions for baseline (left) and SPASS-meter (right).

disabled in subsequent experiments services, such as bluetooth, service discovery (avahi-daemon), extended keyboard handling (triggerhappy), regular task scheduling (cron), or the network service. Table 5, Id 6 is a representative example illustrating that this did not lead to significant changes.

- To reduce the impact of I/O operations during the benchmarks, we created a *RAM drive* with a capacity 100 MBytes so that the JVM could still operate with a 512 MByte heap as described in [16]. However, the RAM drive was too small to store all benchmark results. Therefore, we modified the benchmark script so that the results were moved from the RAM drive to the SD card after completing an individual benchmark step. As shown in Table 5, Id 5, this did not significantly change the results.
- *Swapping* memory pages from/to CPU caches or storage devices may cause response time fluctuations. We disabled swapping for a benchmark run (Table 5, Id 13), but without significant effect on the response time results.
- The Raspbian versions that we used in our experiments adjust the *CPU clock speed* dynamically to the system load. The default mode is `ondemand`, i.e., for a Pi 3, the operating system switches the CPU clock speed between minimum (600 MHz) and maximum (1.2 GHz) clock speed. Such abrupt frequency changes may cause response time fluctuations. In our experiments, we fixed the CPU frequency either to `powersave` mode (600 MHz) or `performance` mode (1.2 GHz). While the `powersave` mode increased the response time by a factor of 2 and caused an increase of the standard deviation as well as more response time peaks (Table 5, Id 12), the `performance` mode did not significantly change the results (Table 5, Id 11).
- Hardware and software can cause *interrupts* that suspend normal program execution. A comparison of the system interrupt table before and after a benchmark execution indicated a high number of timer, USB (representing correlated SD-card and direct memory access) and rescheduling interrupts. Figure 3 illustrates the aggregated results for all CPU cores running the baseline and SPASS-meter. The baseline produced fewer interrupts than the SPASS-meter benchmark. This is reasonable as SPASS-meter applies scheduled execution of some probe collections. While we analyze modifications to SPASS-meter in this regard in Section 5.4, we focus here on the rescheduling interrupts To analyze the effects, we ran the experiments while pinning the benchmarks to specific CPU cores. Utilizing only one core increased the timer and work interrupts by a factor of 2 and avoided more than 98% of the rescheduling interrupts, but also caused a significant performance drop and more response time peaks (Table 5, Id 9). Running the benchmark on two cores reduced the timer interrupts by 37% and led to a similar performance as utilizing all cores (Table 5, Id 10).

Despite some effort and applying typical benchmark preparations such as disabling system services, we did not find a clear root cause for the peaks in the operating system category.

5.3 Java Virtual Machine

The next layer that can influence Java benchmark results is the JVM itself. As described in Section 3, we used an Oracle JVM for ARM in our experiments.

- By default, the Oracle JVM for ARM utilizes a sequential *garbage collector*, while the JVM for Intel processors relies on parallel garbage collection. We forced parallel garbage collection through a command line switch during the benchmark experiments, but this increased the mean response time by 15% as shown in (Table 5, Id 3).
- The fluctuations could be caused by properties of the specific JVM implementation. However, the alternative OpenJDK JVM for ARM does not provide a just-in-time compiler and was, thus, in our trials by orders of magnitude slower, making direct comparisons unfeasible.

Although the JVM or the JVM settings could be a reason for the fluctuations, we were not able to identify a clear root cause.

5.4 Benchmarks

The final layer is the program running within a JVM, in our case MooBench, SPASS-meter, and Kieker. Regarding SPASS-meter, we identified four different potential causes:

- In the original MooBench setup, information on all supported resources is collected. In particular, monitoring the memory usage is a resource-consuming task [10] that stresses the internal event-processing. In this experiment (Table 5, Id 4), we changed the monitoring scope to observe response time as the only resource. This improved the average response time for SPASS-meter by 11% (we classify the change of the average response time of the baseline as an outlier) and reduced the extreme peaks by factor 2.
- As discussed in [9], the initialization of internal object pools for instance reuse may have significant impact on the performance. We re-visited (and adjusted) the object pools of SPASS-meter, which caused only a minor improvement of the mean response time (Table 5, Id 2), while also increasing the maximum (peak) response time and the number of peaks.
- SPASS-meter uses a timer to regularly pull process and system-level resource consumptions. We disabled this timer, which is not relevant for the benchmarking results here. Reconsidering the interrupts discussed Section 5.2, we recorded roughly the same number of USB/SD card interrupts and

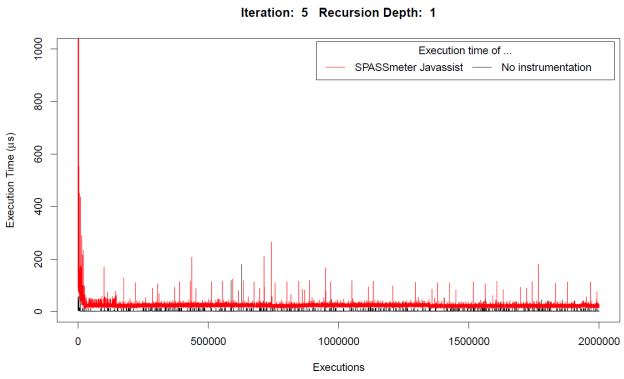

Figure 4: Response time with recursion-depth 1 instead.

work interrupts, while the number of timer interrupts increased by 14% and the number of context switches decreased by 17%. As indicated in (Table 5, Id 7), the mean response time slightly improved by 2.7% and the number of peaks dropped by 39% for most of the following experiments.

- SPASS-meter uses a producer-consumer pattern to asynchronously process collected probe information. For experiments, synchronous event processing can be used [10], which may increase the response time but also reduce threading effects in the timer interrupts. Using this mode, mean and median response time did not change significantly (Table 5, Id 8), the standard deviation increased by factor 4 and the maximum response time by factor 31. As expected, the number of timer interrupts decreased, while the amount of rescheduling and work interrupts did not change.

MooBench itself could also be a cause for the fluctuations. In particular, the parts of the benchmark running during the test could influence the results. We therefore changed the recursion depth in the benchmark from 10 to 1. Although this did not affect the baseline measures (Table 5, Id 14), it did reduce the number of peaks by 67%. Even if smaller peaks remained, the huge peaks disappeared as illustrated by the response time graph in Figure 4. Moreover, the average response time, standard deviation as well as minimum and maximum response time improved significantly.

One important observation is that the *baseline*, i.e., the execution of the benchmark test case without any monitoring, contains a high number of (relative) peaks. This fact remained irrespective of all experiments that we conducted.

5.5 Summary

We identified the recursive benchmark test as a trigger for the massive response time peaks we observed. However, the underlying reason is still unclear. In comparison, the results in [9] (Intel Core i5-2500, 3,3 GHz, 6MB cache, kernel 3.2) only contained few solitary peaks using the same MooBench and SPASS-meter versions without changing the recursion depth. We can imagine that the peaks are caused due to different CPUs/caches, operating systems/kernels or JVMs. As mentioned in Section 5.3, we observed similar fluctuations in the laptop trial (CPU i7-4500U, 1,8 GHz, 4MB cache, kernel 4.8). Although the cache sizes of the Pi are much smaller (cf. Section 2), we do not believe the CPU/cache to be the reason, as the cache sizes of the non-Pi machines are of roughly the same size. However, the Linux kernel versions and the JDK versions differ between the setup used in [9] (JDK 1.7) and our experiments (all kernel 4.x and JDK 1.8). Therefore, it seems more probable that either the kernel or the JVM apply different scheduling/optimization strategies. Confirming this hypothesis would require more cross-platform experiments, which are out of scope of this paper. Furthermore, we identified some optimizations opportunities regarding the application of SPASS-meter (focusing on the relevant resources to be monitored) as well as its implementation (avoiding unused timers, better initialization of shared instance pools). We also identified potential issues of a benchmark setup that can impact the results such as using the 'wrong' garbage collector, setting the CPU to a fixed frequency, or trying to pin the benchmark to less CPU cores than needed.

6 RELATED WORK

Replicability and reproducibility are well-known problems in empirical software research. However, in particular computational replicability is known to be only episodically aimed at in experimental computer science [3]. A major reason for this are that reproducing experiments from scratch is time-consuming, error-prone, and sometimes just infeasible, typically due to insufficient documentation of the experiment, an experiment setup not running on the target environment, missing libraries, different library versions, or the inability to install the required dependencies [2, 3, 6, 9]. Even in standardized high-performance environments, replicability is difficult to achieve [14].

Several approaches for achieving replicability are discussed in the literature. Similar to our approach, Tso et al. employ Raspberry Pi devices to create an affordable, replicable environments for distributed computing, called the Glasgow PiCloud [22]. This environment consists of about 50 devices, which are used to build a scale model of a data center. The PiCloud also makes use of container-based virtualization. However, the containers are used as a replacement for virtual machines, which are not feasible on the Raspberry Pi due to the limited resources and the lack of hardware support, not for replicating experiments. A similar setup with more than 300 devices is described by Abrahamsson et al. [1].

Instead of replicating performance experiments locally, experiments may also be run in Cloud environments. Although most Cloud providers offer standardized instance types, these types are often not clearly and sufficiently specified [12], and may differ significantly in performance. Furthermore, the provider may move virtual machines to different hosts or even change the underlying hardware or the type specification at its own discretion, posing a threat to replicability.

De Oliveira et al. present an infrastructure called DataMill [7], which allows to run experiments on a pool of different worker machines provided by the DataMill community. This infrastructure aims at producing robust and replicable results by running the experiments on multiple devices with slightly different specifications, thus creating a results less dependent on the specifics of a particular setup. Furthermore, this infrastructure allows researchers to explore how particular changes to the environment (e.g., compiler

Id	Experiment (base)	Baseline						SPASS-meter					
		mean	σ	min	max	95% CI	peaks	mean	σ	min	max	95% CI	peaks
1	from [16]	1.6	0.2	1.5	105.2	[1.6;1.6]	1,667	164.8	44.1	91.9	19,228.7	[164.8;164.8]	1,155
2	object pools (1)	1.6	0.3	1.5	352.2	[1.6;1.6]	1,864	152.3	142.5	89.8	370,604.0	[152.3;152.4]	**818**
3	parallel GC (2)	1.6	0.2	1.5	107.5	[1.6;1.6]	1,632	**194.4**	56.7	110.1	27,715.9	[195.4;194.5]	**6,901**
4	time resources (2)	1.6	0.3	1.5	358.4	[1.6;1.6]	1,729	**146.3**	34.9	88.5	13,034.8	[146.2;146.3]	406
5	ramdrive (2)	1.6	0.2	1.5	132.9	[1.6;1.6]	1,685	146.8	40.0	90.6	19,453.1	[146.7;146.7]	534
6	services (5)	1.6	0.3	1.5	207.2	[1.6;1.6]	1,774	150.5	39.2	91.0	24,952.0	[150.4;150.5]	528
7	SPASS timer(6)	1.6	0.3	1.5	545.9	[1.6;1.6]	1,695	146.1	36.8	91.5	10,972.5	[146.2;146.3]	321
8	SPASS events (7)	1.6	0.2	1.5	108.6	[1.6;1.6]	1,773	146.7	157.7	86.7	349,777.3	[146.6;146.7]	333
9	one CPU core (6)	1.6	0.3	1.5	312.5	[1.6;1.6]	**2,223**	**492.8**	**427.1**	86.0	13,560.1	[492.6;493.1]	**37,360**
10	two CPU cores (6)	1.6	0.3	1.5	616.2	[1.6;1.6]	1,818	147.4	46.7	89.7	54,325.9	[147.3;147.4]	348
11	max CPU clock (6)	1.6	0.2	1.5	98.2	[1.6;1.6]	1,628	148.1	41.0	108.5	12,913.6	[148.1;148.2]	359
12	min CPU clock (6)	**3.1**	**0.5**	**3.0**	**945.6**	**[3.1;3.1]**	**3,116**	**294.8**	**80.5**	**177.0**	**120,450.8**	**[294.7;294.8]**	**752**
13	no swapping (6)	1.6	0.3	1.5	185.1	[1.6;1.6]	1,704	147.4	40.6	88.9	13,771.2	[147.4;147.4]	388
14	no recursion (6)	1.4	0.2	1.3	191.6	[1.4;1.4]	1,534	17.6	1.8	11.35	3,361.3	[17.6;17.6]	53

Table 5: Summary of selected case experiments on response times in μs. Notable changes are shown in bold font.

switches) affect their experiments. A similar goal is pursued by the PerfDiff framework by Zhuang et al. [24].

As previously mentioned, replication of performance experiments also requires replicating the surrounding software environment. We used Docker containers for this purpose, which is recommended by several authors [2, 5]. Chirigati et al. present ReproZip [3, 4], a tool which facilitates creating container images by tracking the accessed files during an experiment by monitoring system calls, and automatically adding them to the image.

Another approach to replicating the software environment is to provide fully configured virtual machines, as suggested by [11]. However, virtual machine images can be very large, and since the entire operating system is included in the image, licensing issues may occur. A third approach relies on using configuration management tools able to automatically set up a machine according to pre-defined rules, such as Ansible,[13] Chef,[14] or Puppet[15] [15].

In order to identify potential root causes for the fluctuations in our previous experiments, we furthermore performed a root cause analysis. Typically, a root cause analysis consists of steps like data collection, causal factor charting, root cause identification and recommendation generation [20]. In our case, performing a complete data collection was not feasible, so we opted for an incremental analysis with interleaved factor charting and progressing based on excluded root causes. Of course, an automated approach to root cause detection would be highly desirable, in particular to reduce the manual effort. Existing automated approaches typically focus on one specific layer of the software stack such regression testing [13], web applications and related services [17], or single programs that can be instrumented to obtain the calling context tree [24]. However, in our situation, we applied an incremental manual process as in statistical debugging [21] or in [9], but here considering a wide range of potential causes across multiple layers of the involved hardware and software stack.

[13] https://www.ansible.com/
[14] http://www.chef.io/
[15] http://www.puppet.com/

7 CONCLUSIONS AND FUTURE WORK

In this section, we conclude the paper, present lessons learned from our experiments, and point out directions for future work.

7.1 Conclusions

In this paper, we have presented results and experiences from different experiments to evaluate to what extent the Raspberry Pi and Docker can be used as a platform for replicable performance experiments. Furthermore, we presented a systematic root cause analysis to identify potential sources for variance. Below, we present the answers to the research questions presented in the introduction.

RQ 1: We conclude from the experimental results that the Raspberry Pi appears to be well suited for replicating microbenchmarks, in particular benchmarks that are not very I/O-intensive. Replicating macroscopic experiments may work as well, but depends on the availability of comparable peripherals such as storage devices. The platform is less suited for enterprise-oriented benchmarks, as it may lack the sheer processing power or memory capacity to meet their requirements.

RQ 2: Docker has proven to be a valuable tool for packaging experiments in a replicable way. However, this comes at the cost of slightly increased variance in the results, and a potential performance impact. Furthermore, the virtualization can be a source of additional complexity, such as the connectivity issue observed in Experiment 4.

RQ 3: Despite considerable effort, we identified triggers for the fluctuations observed in the experiments, but, in the end, we were unable to pinpoint root causes. However, our results do not indicate any systematic flaw of the platform itself.

In conclusion, we think that Docker on the Raspberry Pi is indeed a viable option for building replicable performance microbenchmarks.

7.2 Threats to Validity

We see the the greatest threats to the validity of our results in the selection of the experiments and the small number of devices

that were available to use. Furthermore, most of our experiments were run on the Java Virtual Machine, so that the results may not be transferable to experiments running in other environments. As discussed in the Future Work section below, we intend to run additional experiments to further increase the validity of our results.

7.3 Lessons Learned

During our experiments, we learned several lessons about running performance experiments with the Raspberry Pi and Docker, which we summarize below:

- *Docker facilitates running benchmarks and fosters experimentation*, especially due to the fact that containers can be easily (re-)created in a defined state.
- *I/O-heavy experiments should be executed only on hard disks.* We broke two SD cards during our experiments due to high write counts.
- *As soon as peripherals are involved, power consumption is an issue.* Common USB power supplies, such as the ones shipped with mobile phones or tablet computers, provide too little electrical current for a Raspberry Pi and a USB hard drive under heavy load.
- *Container networking can be tricky*, as seen in the SPECjEnterprise experiment.
- *Merging and analyzing* experiment results created at different geographical locations as in our case worked pretty well, also in particular to agreements on using the same formats, naming conventions and tools.
- *Legal issues may prevent publication of container images.* Some software components can be used free of charge, but limitations may apply regarding redistribution. For example, it is currently unclear whether distributing Oracle's JDK in a Docker container is compliant with the underlying license.[16]

7.4 Future Work and Directions

In our future work, we intend to extend our analysis to locate potential root causes for the performance fluctuations. We also plan to further evaluate the viability of the Raspberry Pi as well as other single-board computers for additional benchmarks. As we expect the next generation of Raspberry Pi to be equipped with more memory and computing power, executing more demanding benchmarks might become possible in the future. We furthermore intend to conduct experiments on a larger number of Pi devices to reduce the influence of potential device-specific deviations.

Moreover, we envision that the results of different researchers in the direction of replicable performance experiments could foster a community practice, including best practices and default experiment workflows, but also accepted technical means, such as Docker, standardized hardware, or even hardware-benchmark combinations specified and endorsed by benchmark organizations. Further, a public experiment repository containing reference Docker experiment images, but also standardized installation images for the operating system to avoid uncontrolled changes to the host system would be desirable. First steps towards such a community practice are visible as numerous conferences and journals encourage researchers to also submit artifacts, including Docker images.

[16] see http://blog.takipi.com/running-java-on-docker-youre-breaking-the-law/

Future steps might include public experiment repositories or even an accessible science (Pi) cloud. This would facilitate the sharing of experiments between researchers and pave the way for artifact and cross-validation tracks or new publication models, such as, for instance, proposed in [3].

REFERENCES

[1] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni. 2013. Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment. In *Intl. Conference on Cloud Computing Technology and Science*.

[2] C. Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (2015).

[3] F. Chirigati, R. Capone, R. Rampin, J. Freire, and D. Shasha. 2016. A Collaborative Approach to Computational Reproducibility. *Inf. Syst.* 59, C (July 2016), 95–97.

[4] F. Chirigati, D. Shasha, and J. Freire. 2013. Packing Experiments for Sharing and Publication. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[5] J. Cito and H. C. Gall. 2016. Using Docker Containers to Improve Reproducibility in Software Engineering Research. In *Intl. Conference on Software Engineering Companion*. 906–907.

[6] A. Davison. 2012. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science & Engineering* 14 (2012), 48–56.

[7] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. 2013. DataMill: Rigorous Performance Evaluation Made Easy. In *Intl. Conference on Performance Engineering*. 137–148.

[8] Docker, Inc. 2017. Dockerfile reference. (2017). https://docs.docker.com/engine/reference/builder/.

[9] H. Eichelberger, A. Sass, and K. Schmid. 2016. From Reproducibility Problems to Improvements: A Journey. In *Symposium on Software Performance*.

[10] H. Eichelberger and K. Schmid. 2014. Flexible Resource Monitoring of Java Programs. *Journal of Systems and Software* 93 (2014).

[11] I. P. Gent and L. Kotthoff. 2014. Recomputation.Org: Experiences of Its First Year and Lessons Learned. In *Proc. of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*.

[12] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. 2010. Case Study for Running HPC Applications in Public Clouds. In *Intl. Symposium on High Performance Distributed Computing*. 395–401.

[13] C. Heger, J. Happe, and R. Farahbod. 2013. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Intl. Conference on Performance Engineering*. 27–38.

[14] T. Hoefler and R. Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve ways to tell the masses when reporting performance results. In *Intl. Conference on Supercomputing*.

[15] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. 2017. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Intl. Parallel and Distributed Processing Symposium Workshops*.

[16] H. Knoche and H. Eichelberger. 2017. The Raspberry Pi: A Platform for Replicable Performance Benchmarks?. In *Symposium on Software Performance*. accepted, available on request.

[17] J. P. Magalhães and L. M. Silva. 2011. Root-cause Analysis of Performance Anomalies in Web-based Applications. In *Symposium on Applied Computing*. 209–216.

[18] R. D. Peng. 2011. Reproducible Research in Computational Science. *Science* 334, 6060 (2011).

[19] M. Richardson. 2016. Docker comes to Raspberry Pi. (2016). https://www.raspberrypi.org/blog/docker-comes-to-raspberry-pi.

[20] J.J. Rooney and L.N.V. Heuvel. 2004. Root cause analysis for beginners. *Quality Progress* 37 (2004), 45–53.

[21] L. Song and S. Lu. 2014. Statistical Debugging for Real-world Performance Problems. *SIGPLAN Not.* 49, 10 (2014), 561–578.

[22] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros. 2013. The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures. In *Intl. Conference on Distributed Computing Systems Workshops*.

[23] J. Waller, N. C. Ehmke, and W. Hasselbring. 2015. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *Software Engineering Notes* 40, 2 (2015).

[24] C. Zhuang, S. Kim, M. Serrano, and J.-D. Choi. 2008. PerfDiff: A Framework for Performance Difference Analysis in a Virtual Machine Environment. In *Intl. Symposium on Code Generation and Optimization*. 4–13.