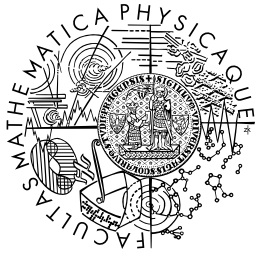

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Tomáš Drozdík

Asynchronous Duet Benchmarking

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Vojtěch Horký, Ph.D.
Study programme: Software Systems
Study branch: System programming

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor, family and friends for their patience and support.

Title: Asynchronous Duet Benchmarking

Author: Tomáš Drozdík

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Ph.D., Department of Distributed and Dependable Systems

Abstract: Accurate regression detection in volatile environments such as the public cloud is difficult. Cloud offers an accessible and scalable infrastructure to run benchmarks, but the traditional benchmarking methods often fail to predict regressions reliably. Duet method acknowledges the variability and runs the workloads in parallel, assuming similar outside impact symmetry. This thesis examines a duet variant that does not synchronize harness iterations which enables broader use of this method. The asynchronous duet method can detect $1 - 5\%$ slowdowns for most of the tested benchmarks in volatile environments while reducing the overall costs by up to 50%. Measurements were obtained by a benchmark automation tool for running and processing benchmarks from multiple suites. This tool can run benchmarks with sequential and both duet methods utilizing containers for portability.

Keywords: benchmarking cloud duet containers

Contents

Introduction **3**

1 Background and Motivation **5**
1.1 Benchmarking in the cloud 6

2 Duet benchmarking **9**
2.1 Synchronous vs. Asynchronous duet 10
2.2 Goals 13

3 Implementation **15**
3.1 Architecture 15
3.1.1 Synchronized duet in containerized environment 16
3.1.2 Run Scheduling 16
3.2 Configuration 17
3.3 Result parsing 18
3.3.1 Notation 18
3.3.2 Overlaps 19

4 Overview of Results Processing **21**
4.1 A/A runs 21
4.2 Data filtering 21
4.3 Accuracy analysis and performance regression tests 22
4.3.1 Relative standard deviation 22
4.3.2 Confidence interval test 22
4.3.3 Mann-Whitney u-test 24
4.4 Minimal detectable slowdown 24

5 Experimental evaluation **27**
5.1 Experiment setup 27
5.1.1 Docker setup 28
5.1.2 Run summary 28

5.2 Measurements 29
5.2.1 RQ1: Runtime savings 29
5.2.2 RQ2: Pair overlaps 30
5.2.3 RQ3: Accuracy improvements 31
5.2.4 RQ4: Minimal detectable slowdowns 34
5.3 Evaluation automation 36

Conclusion **39**

Bibliography **43**

Introduction

Modern software development with rapid development cycles is in dire need of a fast, accessible, scalable, and reliable performance regression detection method. A typical question is if neighboring commits of the same software manifest regression. For instance, authors of a compiler might be concerned if a new optimization increases the performance of a particular benchmark.

The traditional performance measurement method relies on a dedicated bare-metal server with an isolated and carefully set environment that tries to prevent measurement noise. This method is reasonably fast and reliable but lacks accessibility and scalability requirements as the maintenance cost is high.

To address the scalability and accessibility, research over the past decade has focused on offloading performance measurements to the cloud [1, 2, 3]. Cloud offers scalable on-demand infrastructure with pay for what you use principle. However, this research has shown that performance measurement in the cloud is difficult because of the high variability of the shared infrastructure environment.

Even in ideally crafted bare-metal conditions, multiple measurements are necessary to deal with the inherent variability introduced by the code running platform, such as just-in-time compilation, garbage collection, memory mapping, and CPU frequency scaling. Benchmark code itself can have its share of non-determinism that contributes to the overall variability of the measurements. Hence, in addition to benchmark harness running its workload in a loop, practitioners often run the benchmark multiple times. Afterward, analyze individual iteration scores to evaluate the final score.

In the context of compiler programmers, they may run multiple benchmarks on a version with and without the optimization and compare the final scores. We refer to this method as *sequential benchmarking* as it runs different versions after each other.

Bulej et al. [4] introduced a new benchmarking method called *duet benchmarking* designed with volatile environments such as the cloud in mind. The main idea of duet benchmarking is to run both versions in parallel. The distinguishing property of this method is that it does not try to prevent interference. On the contrary, duet benchmarking acknowledges the presence of variability and relies on fair scheduling to make the impact equal on both versions running in parallel. Afterward, the

benchmark scores are statistically evaluated to determine if there is a performance regression between the two versions. The duet benchmarking method has shown improvement ranging from $5.03\times$ on average for Scalabench and DaCapo workloads to $37.4\times$ on average for SPEC CPU 2017 workloads [4]. However, this method relies on synchronized iterations of a benchmark, which requires benchmark harness modification that synchronizes individual iterations between harnesses.

This thesis explores the relaxation of this synchronization requirement which creates a new variant of the duet method we call *asynchronous duet method*. To achieve this, we implement a tool for running benchmarks that supports sequential, synchronous, and asynchronous duet methods. One novel attribute of this tool is that it runs benchmarks in containers to achieve portability and ease of setup.

Using the tool, we run benchmarks from multiple benchmarking suites — Renaissance, Scalabench, DaCapo, and SPEC CPU 2017. The first three are JVM-based while the last one represents natively compiled benchmarks. The goal of this thesis is to assess the viability of the asynchronous duet method for different benchmarks across different environments. Our choice of environments includes bare-metal measurements on a dedicated server, AWS EC2 `t3.medium` instances as cloud representative, and self-hosted shared virtual machine infrastructure.

We examine sequential and duet method differences on overall runtime duration, measurement variability, A/A run detection accuracy, and the factor of minimal detectable slowdowns (MDS) [2].

The thesis is structured as follows. The first chapter provides some background on benchmarking in volatile environments. The second chapter delves into duet benchmarking and poses research questions for this thesis, with the goals summarized at the end of this chapter. The third chapter describes the architecture of our benchmark automation tool capable of running benchmarks from multiple suites using both sequential and duet methods. The fourth chapter summarizes the statistical methods used to evaluate regressions which are later used in the fifth chapter that presents the results. Finally, we conclude the thesis and lay out the possibilities for future work in this area.

Chapter 1

Background and Motivation

Benchmark harness sets up an environment, executes benchmark code in multiple iterations, and then presents some score. The score can be iteration time, throughput, number of processed requests, etcetera. When a benchmark is a small piece of code, also referred to as performance unit tests by Horkỳ et al. [5], it is usually written in some microbenchmark framework. Microbenchmark execution in a public cloud has been thoroughly examined by Laaber, Scheuner, and Leitner [2]. If the benchmark code gets bigger, it is usually called the application benchmark.

Benchmarks do not have the sole purpose of measuring the performance of an application. Inversely they provide a way to measure the capabilities of some hardware. For this reason, benchmark suites aggregate many application workloads, put them under a single harness, and simplify the performance evaluation process. Examples of such suites are from Standard Performance Evaluation Corporation[1] that has multiple suites, for instance, SPEC CPU for natively compiled benchmarks and SPECjbb, which is JVM-based. Other standalone suites used in this thesis are Renaissance [6], DaCapo [7], and Scalabench [8]. All suites mentioned above are JVM-based except for SPEC CPU, which has benchmarks written in C, C++, and Fortran and hence is natively compiled. Apart from testing hardware performance, benchmark suites are good candidates for research in performance measurement methods as they strive to provide a wide range of real-world representative workloads.

The requirement for multiple repetitions causes an inherent trade-off between accuracy and execution time. Execution of a benchmark suite for some software project version to identify newly introduced performance regression can take an overwhelming 3000 machine hours per day as analyzed by Bulej et al. [4] on an open-source GraalVM project [9]. By then, a new version could have come around. This trade-off between measurement execution time and the number of commits between tested versions can be addressed from multiple angles. One is to reduce the number of

[1] `https://www.spec.org`

measurements by identification of commits that might introduce performance regression Oliveira et al. [10]. Other is to reduce the execution time of measurements by parallelization. Effective parallelization involves maintaining multiple dedicated machines, which is difficult and costly. Therefore, much research over the past decade has focused on offloading performance measurements to the cloud [1, 2, 3].

1.1 Benchmarking in the cloud

Cloud offers practically infinite hardware scaling capabilities with on-demand access and a pay-for-what-you-use principle. That makes it a tempting platform for fast and accessible performance measurements. Cloud offers a large scale of options to run your code on from bare metal offerings[2] to running small pieces of code as lambdas[3]. The most commonly examined and accessible method is an Infrastructure-as-a-Service (IaaS) cloud offering where computing resources are acquired and released as a service.

IaaS is typically in form of virtual machines or containers with attached virtual disks. These instances come with different configuration options, such as the number of virtual CPUs, memory size, network throughput, and virtual disk. This gives practitioners some control over what hardware and software platform is used to run their code compared to higher-level offerings such as Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

Studies such as Leitner and Cito [1] and Laaber, Scheuner, and Leitner [2] summarize the viability of measuring software performance on IaaS offerings from different providers. They point out that measuring software performance on the public cloud is a moving target, with providers constantly changing their offerings, hardware, and processes. A common concern is that a given instance type may not run on the same type of hardware and that the same physical hardware might host multiple colocated instances. Hardware heterogeneity, shared and virtualized infrastructure further amplifies the variability of performance measurements in the cloud environment [1] even to the point that practitioners, using state-of-the-art performance measurement and statistical methods, might not be able to detect 1000% slowdown of some measured software as shown by Laaber, Scheuner, and Leitner [2].

With many repetitions and variability across the results, it is necessary to carefully process the results and use statistical methods to find out if a code change introduces a new performance regression. The use of statistical methods has been extensively studied in Bulej, Horký, and Tůma [11] and further analyzed in Laaber, Scheuner, and Leitner [2] specifically in the cloud context. To quantify the variability of the distribution of results from a benchmark Laaber, Scheuner, and Leitner [2] use the

[2]AWS Outposts `https://aws.amazon.com/outposts/`

[3]AWS Lambda `https://aws.amazon.com/lambda/`

coefficient of variance, also called mean standard deviation. To compare results of two software versions Bulej, Horký, and Tůma [11] use both methods of overlapping confidence intervals computed by bootstrap procedure and hypothesis testing with Wilcoxon rank-sum test and its variation Mann-Whiteney U-test as well as Welch's t-test. Bulej, Horký, and Tůma [11] also raise concerns about the validity of these methods when used for performance measurement data. Concerns stem from the fact that performance measurements don't satisfy many common assumptions that statistical methods make, such as random independent and identically distributed samples and asymptotic normality of the data. Moreover, performance data also tend to exhibit long-range dependencies between samples, have unknown distributions with long tails, and are not stationary in general, which further complicates statistical analysis [11]. So far, there does not seem to be any "silver bullet" statistical method that would be a good fit for performance data, and thus, these techniques need to be used with caution as reasoned in multiple studies [1, 2, 11].

Unfavorable conditions for benchmarking in the cloud spiked interest in research that focuses on performance measurement methods that reduce measurement variability, increase accuracy and reduce false positives of regression detection in a cloud environment. One such method and current best practice [2] is called *Randomized Multiple Interleaved Trials* (RMIT) introduced by Abedi and Brecht [3]. Trial refers to a single run of benchmark, with multiple inside iterations, on a particular version of the software it measures the performance of. To compare performance between multiple versions of some software, one has to run the same benchmark with each alternative. RMIT states that these runs of these alternatives should be run (1) multiple times as discussed above and (2) repetitions of all alternatives should be randomized to minimize both partial and periodic outside interference.

Another method that aims to reduce variance is duet benchmarking, first introduced in Bulej, Horkỳ, and Tůma [12] and further expanded on in Bulej et al. [4]. The next chapter examines the duet method and its new asynchronous variant in detail and poses research goals for this thesis.

Chapter 2

Duet benchmarking

Performance comparison of two different software versions is referred to as A/B benchmarking. For example, compiler authors want to test if their optimization improved the performance of some benchmark, or web developers want to try if a recent commit caused regression in their internal benchmark. In both cases, the benchmark code is the same, and it runs with two slightly different workloads, versions A and B, that are similar in nature.

Figure 2.1 shows a comparison between three benchmarking methods analyzed in this thesis (1) sequential, (2) synchronous duet, and (3) asynchronous duet. The sequential method is the simplest, where each version is run independently of the other. On the other hand, the synchronous duet runs A and B in parallel, and it has to synchronize the starts of individual iterations. Asynchronous duet runs in parallel as well but does not require synchronized iterations.

The example in figure 2.1 has the durations of iterations equal for all methods, which illustrates one immediate benefit of running duet benchmarks — overall runtime reduction. In an ideal case, the duet method could save up to 50% of execution run time and hence halve execution costs. As this speedup is crucial for time and monetary reasons, we will investigate it thoroughly in this thesis. Therefore, we would like to answer the following research question.

> **RQ1:** *What are the runtime savings of the asynchronous duet method compared to the sequential method?*

However, to achieve such speedup, workloads running in parallel must not interfere with each other. To address mutual workload interference, Bulej et al. [4] set up measurements in such a way that each benchmark was restricted to a single dedicated virtual core. Even then, it may be the case that the two virtual cores map to the same hardware threads of the same physical core. Such virtual cores would then compete for the single shared physical core. Here Bulej, Horkỳ, and Tůma [12] rely on workload symmetry.

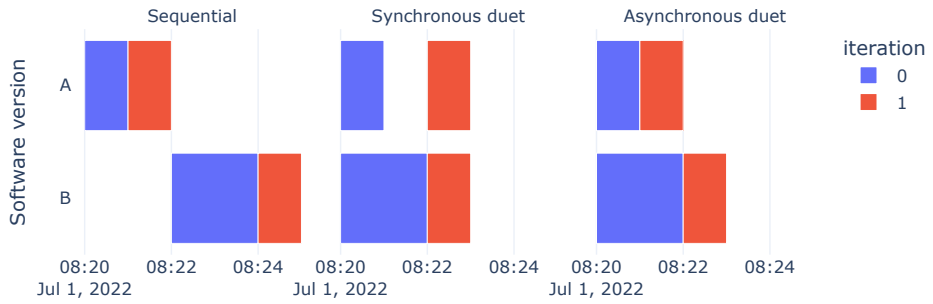

Figure 2.1 Comparison of different benchmarking methods, sequential, synchronous duet, and asynchronous duet, when comparing two software versions, A and B. Benchmark runs two iterations with the same durations for all methods and versions. Note that version B introduced a regression where the first iteration takes twice as long. The x-axes is an example timeline of measurements.

Workload symmetry states that if a cloud were to exhibit systematic performance differences between the two virtual cores running duet workloads, it would likely exhibit similar unwarranted performance differences in other common concurrent workloads, such behavior would be likely considered a bug and remedied. In other words, duet benchmarking relies on fair CPU scheduling such as Completely Fair Scheduler [13] in Linux to work out mutual interference. Bulej, Horkỳ, and Tůma [12] also note that this works only for similar workloads, which can be expected of neighboring commits or versions of the same software.

2.1 Synchronous vs. Asynchronous duet

Figure 2.1 also note that synchronized duet methods are not always running in parallel. Once an iteration finishes it has to wait for its pair iteration to finish, until then, this pair iteration runs alone. It might seem that an asynchronous duet solves this issue because iterations are not blocked on the barrier, so the only iteration running alone will be the last iteration of either A or B, whichever finishes last.

However, the asynchronous duet might be worse off in terms of equal impact symmetry on parallel workloads due to the ways benchmark harnesses and benchmarks themselves work. Algorithm 1 is a generic example of what an inner loop of a benchmark harness might look like. Pre-iteration and post-iteration functions might do some environment checks like running garbage collection and recording or validating results. Therefore, it is natural that these additional steps create gaps between

measured code and that impacts asynchronous duet symmetry.

Algorithm 1 Generic workings of the benchmark harness which executes a benchmark. Note that not all harnesses follow this structure — some functions might be effectively empty. Specifically for synchronous duet Bulej et al. [4] had to modify *PreIteration* to wait on tabarrier.

```
function RunHarness(b)
    Setup(b)
    for i ∈ 1 ... I do
        PreIteration
        start ← clock.now
        RunBenchmark(()b)
        end ← clock.now
        PostIteration(b, i, start, end)
    end for
    ValidateResults(b)
    Teardown(b)
end function
```

Figure 2.2 shows what implications can the absence of synchronization have on workload overlaps. With enough iterations, harness specifics and potentially different A/B performance workloads will inevitably diverge.

Even if workloads are partially overlapping, the start of a workload might do different things, for example, be CPU bound while the end might be I/O bound. Fair CPU scheduling should, in theory, allocate the same amount of processor time for both workloads, essentially synchronizing them. However, in practice, due to the inherent variability of these systems, workloads will diverge.

For example, figure 2.3 demonstrates a case when this happens. Even workloads with the same performance might get to a state of equilibrium where parts of workloads with different bound nature overlap each other. This kind of symmetry can further hinder the impact symmetry argument for the synchronous duet method. Therefore, our next research question examines the way duet pairs overlap.

RQ2 *How do workloads overlap in the asynchronous duet?*

For the asynchronous duet method to be useful, runtime reduction and overlapping workloads are not enough. The asynchronous duet method has to show accuracy in regression detection on par with the synchronized duet method:

RQ3: *What is the accuracy of regression detection of asynchronous duet compared to synchronous duet and sequential execution across different environments?*

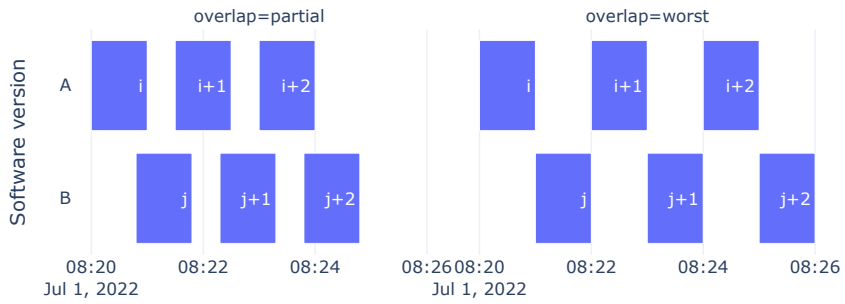

Figure 2.2 Tailored examples of how lack of synchronization and time in between workload execution might affect duet overlap. This is an example timeline of iterations of an *asynchronous duet* measurement from an iteration i and j for versions A and B, respectively. Focus on how versions A and B overlap — just partially or even not at all.

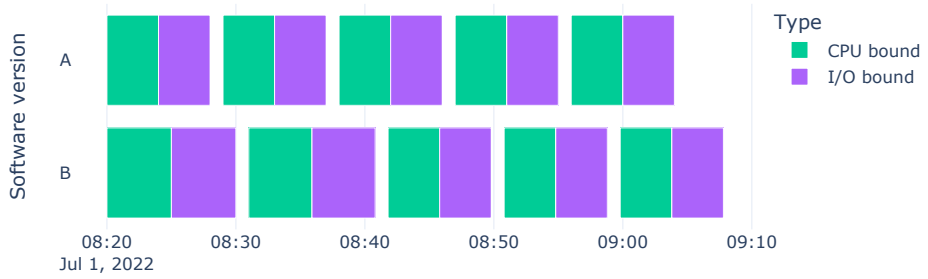

Figure 2.3 Tailored examples of how different nature of a workload might compromise the duet's impact symmetry. Initially, A and B overlap on both CPU and I/O bound parts, slowing down B for the first two runs. Then, CPU and I/O bound parts of A and B workloads overlap less end B turns out to be as fast as A. Notice that A and B enter a state of equilibrium where each is bound on different parts of the underlying hardware.

Additionally, using a similar approach to Laaber, Scheuner, and Leitner [2] asynchronous and synchronous duet methods can be compared with sequential measurements in terms of *minimal detectable slowdowns* section 4.4:

> **RQ4:** *What are the minimal detectable slowdowns with the sequential, synchronous duet, and asynchronous duet measurements that we can detect with* 95% *confidence?*

2.2 Goals

To summarize, these are the areas of focus for this thesis:

Accessibility Asynchronous duet relaxes the synchronized iteration requirement on benchmark harnesses imposed by the synchronized duet method. Our goal is to create a generic benchmark super-harness for running benchmarks from various suites using sequential and duet methods and automate the result processing.

Runtime and cost reduction Duet methods have the potential to drastically reduce, up to 50%, the execution time of benchmarks (RQ1).

Accuracy The synchronized duet method can reduce the variance of benchmark results measured in a volatile environment, such as a public cloud [4]. The question is if the asynchronous duet can do the same (RQ2, RQ3, RQ4).

Chapter 3

Implementation

We implemented a super-harness that takes an existing benchmark harness as an input and executes its benchmarks in the asynchronous duet style.

To keep a configuration portable and dependencies minimal, we have decided to use the container technology, namely Docker [14] or Podman [15]. The tool is named `duetbench`, and it is part of open source python package `duet` [16] that also implements other complementary tools described in section 3.3. Documentation is in the form of GitHub wiki [17].

We want to compare asynchronous duet with Bulej et al. [4], hence our super-harness supports synchronized duet and sequential method in a containerized environment.

3.1 Architecture

Figure 3.1 depicts architecture of `duetbench` script. `duetbench` is a process on the host machine that spawns subprocesses that run containers — one for each version A/B. Since initialization of a container might take longer `duetbench` waits for both containers to start and then uses `docker exec`[1] to run the command that executes the benchmark harness loop from algorithm 1. Once both benchmarks finish, copy the raw result of those benchmarks from their respective containers and store them in a predefined directory structure that preserves super-harness metadata and raw results [17]. Afterward, stop and clean up both containers. We refer to this process as a *run* and it is analogous to a *trial* from Laaber, Scheuner, and Leitner [2] and Abedi and Brecht [3] for sequential runs. For duet methods, one *run* executes 2 benchmarks in parallel.

If there are more benchmarks to execute, pick the next one (according to a particular scheduling strategy, see section 3.1.2) and do the same process. In the case of

[1] `https://docs.docker.com/engine/reference/commandline/exec/`

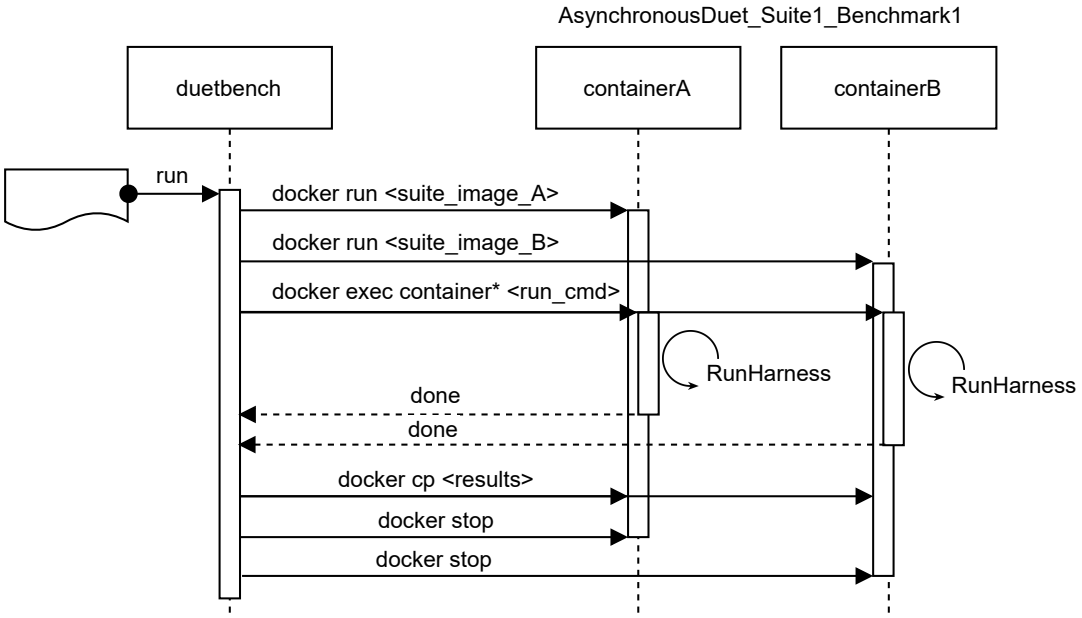

Figure 3.1 UML sequence diagram of how `duetbench` executes a single A/B asynchronous duet benchmark. All the parameters in angled brackets are specified in the configuration file `duet.yml` described in section 3.2. Note that the docker commands are simplified for brevity.

the sequential method, the process is very much the same omit the second container entirely and run them separately.

3.1.1 Synchronized duet in containerized environment

Bulej et al. [4] uses a shared-memory barrier to synchronize benchmark iterations. We ported this solution into the containers to mimic the original behavior.

The barrier uses the pthread API that is not available on all systems, such as macOS. Therefore, `duetbench` starts a third container to initialize and later clean up the barrier, hence bypassing the missing functionality on the host system.

Docker environments are separated by design. To facilitate this feature `duetbench` mounts host directory `-v /dev/shm:/dev/shm` and shares inter-process communication namespace `--ipc host`[2].

3.1.2 Run Scheduling

`duetbench` supports two options for *run scheduling*:

[2] `https://docs.docker.com/engine/reference/run/`

Sequential where benchmarks are run in order of definition from a configuration file.

Randomized Interleaving Trials where it randomly interleaves the runs from a configuration file. Hence, it can mix different types of measurement methods and benchmarks. With multiple repetitions of a given measurement method, it is essentially RMIT [3].

3.2 Configuration

Listing 1 shows section of `duetbench` YAML configuration file that describes an A/B asynchronous duet and sequential run. The goal of the design was to make configuration generic and straightforward to use so that many harnesses and benchmark suites can be adapted to use with `duetbench`. Complete documentation of the `duetbench` configuration, including a description of how to run synchronized duets (not present in the listing 1) is on the wiki [17].

Freedom in run command allows tweaking the harness to suit the asynchronous duet better. For example, some harnesses support skips of validation or various garbage collector options. Detailed configuration of experiments is in section 5.1.

Listing 1 Example part of YAML configuration file for `duetbench` that runs `avrora` benchmark from the DaCapo suite. In this case, both A and B versions are packaged in a single container image as Java JAR archives. Run command specifies how to invoke the DaCapo harness — 50 iterations, results in `results.csv` and run only `avrora` benchmark. All the result files or directories need to be specified in `results` array field. Note the correspondence between user input fields from this configuration and parameters in angled brackets from figure 3.1. Furthermore, users can specify the number of repetitions for both asynchronous duet and sequential measurements, as well as the scheduling strategy for those runs.

```
avrora:
  image: dacapo
  duet_repetitions: 2
  sequential_repetitions: 2
  schedule: randomized_interleaving_trials
  A:
    run: java -jar dacapo-A.jar -n 50 -o results.csv avrora
  B:
    run: java -jar dacapo-B.jar -n 50 -o results.csv avrora
  results:
    - results.csv
```

3.3 Result parsing

Once a run finishes, results are copied in their raw format to a dedicated result directory on the host machine. Results are then processed using `duetprocess` script [17] for further analysis. `duetprocess` takes an output directory of `duetbench` as input and produces a single CSV result file.

The minimal schema of a result data point looks like this:

- Directly parsed by `duetprocess`:
  - suite,
  - benchmark,
  - run — repetition id, not necessary in order of execution,
  - pair,
  - pair order — which pair was started first,
  - iteration,
  - *artifacts* — `duetbench` provides an option to run some additional commands such as `lscpu` or `cat /proc/meminfo` to obtain information about host environment.
- Specific per benchmark suite parsers:
  - iteration start absolute timestamp,
  - iteration end absolute timestamp.

Since different benchmark suites have widely different results formats, users have to write a parser plugin — python function. The parser must obtain iteration start and end absolute timestamps from benchmark results. Additionally, since users need to write a parser for a given benchmark suite, they might add any data that the benchmark produces as an addition to the schema mentioned above.

Some benchmark suites don't track absolute timestamps of iterations and thus need some modification. However, without absolute timestamps, one could not evaluate how iterations overlap (*RQ2*).

3.3.1 Notation

To describe parsed results more formally and describe terminology from the `duet` package, we use the following terms:

Environment $e \in E$ classifies host hardware and software configuration — where `duetbench` runs. The environment is deduced from artifacts in the above schema. The set of environments E used in our experiments is described in section 5.1.

Type $t \in T$ is the measurement method type where $T = \{seqn, sduet, aduet\}$ for sequential, synchronous duet and asynchronous duet respectively.

Pair $p \in P$ is version of tested software in case of A/B runs $P = \{A, B\}$

Benchmark $b \in B$ is a benchmark from a benchmark suite, formally $B = \{(suite, benchmark) \mid benchmark \in suite\}$

Run in environment $e \in E$ of type $t \in T$ running benchmark $b \in B$ and pair $p \in P$ is denoted as $R^{e,t,b,p}$. It is a unit with which `duetbench` works and can run multiple times.

Iterations of a run $R^{e,t,b,p}$ with $iters$ iterations is an ordered set $(i_1, i_2, \dots i_{iters})$ of scores of particular benchmark — execution time in nanoseconds.

To distinguish different measurement methods for some benchmark $b \in B$ in an environment $e \in E$ denote a set of all sequential runs as

$$M^{e,b}_{seqn} = \{(R^{e,seqn,b,A}_r,\ R^{e,seqn,b,B}_r) \mid \forall r \in \{1 \dots runs\}\},$$

set of all synchronized duet measurements, which are naturally paired on the level of pairs and iterations as

$$\begin{aligned} M^{e,b}_{sduet} &= \{(R^{e,sduet,b,A}_r,\ R^{e,sduet,b,B}_r) \mid \forall r \in \{1 \dots runs\}\} \\ &= \{(i^A_j,\ i^B_j)_r \mid \forall r \in \{1 \dots runs\},\ \forall j \in \{1 \dots iters\}\}, \end{aligned}$$

and finally set of all asynchronous duet measurements, naturally paired on the level of pairs as

$$M^{e,b}_{aduet} = \{(R^{e,aduet,b,A}_r,\ R^{e,aduet,b,B}_r) \mid \forall r \in \{1 \dots runs\}\}.$$

Finally, refer to M^e as the union of all runs in an environment $e \in E$

$$M^e = \{M^{e,b}_{seqn} \cup M^{e,b}_{sduet} \cup M^{e,b}_{aduet} \mid b \in B\}.$$

3.3.2 Overlaps

To address *RQ2* `duetprocess` provides a method to compute overlapping iterations of a given asynchronous duet run. Let $(R^A, R^B) \in M^{e,b}_{aduet}$. Iterations of respective runs are $R^A = (i^A_1 \dots i^A_{iters})$ and $R^B = (i^B_1 \dots i^B_{iters})$. Iteration overlap definition uses the following notation:

$$\begin{aligned} start(i) &= \text{start of iteration } i, \\ end(i) &= \text{end of iteration } i, \\ overlap\_start(i^A, i^B) &= max(start(i^A), start(i^B)), \\ overlap\_end(i^A, i^B) &= min(end(i^A), end(i^B)). \end{aligned}$$

Then set of overlaps of these two runs O_{R^A,R^B} can be expressed as

$$O_{R^A,R^B} = \{(i_j^A, i_k^B) \mid overlap\_start(i_j^A, i_k^B) < overlap\_end(i_j^A, i_k^B), \\ \forall j \in \{1 \dots iters\}, \forall k \in \{1 \dots iters\}\}.$$

However, not all overlaps are equally important. As shown in figure 2.2 and *RQ2* introduction, some overlaps might be tiny and cover the start of one iteration with the end of another. The following notations are used to express the quality of an overlap:

$$overlap\_time(i^A, i^B) = overlap\_end(i^A, i^B) - overlap\_start(i^A, i^B),$$
$$overlap\_rate^A(i^A, i^B) = \frac{overlap\_time(i^A, i^B)}{i^A},$$
$$overlap\_rate^B(i^A, i^B) = \frac{overlap\_time(i^A, i^B)}{i^B},$$
$$overlap\_rate(i^A, i^B) = min(overlap\_rate^A(i^A, i^B), overlap\_rate^B(i^A, i^B)).$$

Denote a set of all asynchronous duet overlaps for a benchmark $b \in B$ in an environment $e \in E$ as

$$O^{e,b} = \{O_{R^A,R^B} \mid (R^A, R^B) \in M_{aduet}^{e,b}\}. \tag{3.1}$$

Chapter 4

Overview of Results Processing

With multiple repetitions and inherent variance of measurements, statistical analysis is used to determine performance regression in an A/B run. This chapter describes the statistical methods used to determine and compare measurement variance and regression detection of an A/B run using different benchmarking methods. In particular, it adjusts the confidence interval test from Bulej et al. [4] to work with the asynchronous duet.

4.1 A/A runs

Finding a software project that exhibits a certain slowdown is difficult since regression is often tied to the hardware running the benchmark. Instead, research into performance measurements employs A/A measurement technique [.] A/A measurement runs the same version, and later labels run with A and B labels — as if there were two different versions. Since the code is the same, the performance should be the same as well. The usability of a benchmarking method can thus be a result of its ability to detect A/A measurements correctly. In the context of duet runs, A/B labeling is performed on concurrently running duet pairs.

4.2 Data filtering

In our experiments, we removed the first half of all iterations per run of JVM-based benchmarks to avoid warm-up, same as in Bulej et al. [4]. We kept all measurements of SPEC CPU benchmarks since it is natively compiled.

4.3 Accuracy analysis and performance regression tests

We use various statistical methods to address the variance of different benchmarking methods, running benchmarks in containers across different environments (*RQ3*). First, we compare the raw variance of measurements after data filtering using relative standard deviation. Then we compute the confidence intervals of paired iteration score difference similar to Bulej et al. [4], but it has to be adjusted for the asynchronous duet. That is the basis for the confidence interval test, and it allows us to compare the relative confidence interval width across different methods. Another regression detection method is Mann-Whitney u-test that compares iteration score distributions between pairs. Last we use both regression tests to explore the minimal detectable slowdowns [2] of sequential and duet methods.

4.3.1 Relative standard deviation

Relative standard deviation, also known as the coefficient of variation (CV), is one way to determine the variance of iteration times. For measurement type $t \in T$, benchmark $b \in B$, runs $R^{t,b} \in M^e$:

$$CV^{t,b} = \frac{std(R^{t,b})}{mean(R^{t,b})}.$$

The CV measures how the results of different benchmarks vary relative to each other using different benchmarking methods. However, the variance of duet methods cannot be determined based on CV.

For instance, higher CV for given benchmark duet measurements compared to its sequential measurements doesn't imply worse predictability of duet methods because of the impact symmetry argument. Overall variance might be higher, but the pairwise comparison could be more telling. On the other hand, lower CV for duet methods suggests better robustness of duet methods.

4.3.2 Confidence interval test

To determine if pairs manifest performance change, use the method of confidence intervals from Bulej et al. [4]. This method also compares variance across benchmarking methods based on relative confidence interval width. Here, we describe the method and extend its use for the asynchronous duet.

First, runs and their iterations need to be paired.

Sequential runs don't have any natural pairing but are randomly paired by the nature of A/A runs labeling. We decided to pair their iterations sequentially

$$is\_pair(i_i^A, i_j^B) = True \iff i = j.$$

Synchronous duet runs are naturally paired - sequentially by iterations; thus they have the same is_pair predicate.

Asynchronous duet runs can't be paired sequentially. Considering A/B measurements, non-synchronized iterations will inevitably diverge, hence sequential iteration pairing wouldn't support impact symmetry. Instead, we decided to pair iterations i^A and i^B based on the minimum overlap ratio $m \in (0, 1)$ and thus define a set of filtered overlaps as

$$O_m^{e,b} = \{(i^A, i^B) \mid (i^A, i^B) \in O^{e,b}, overlap\_rate(i^A, i^B) > m\}, \tag{4.1}$$

where $O^{e,b}$ is set of overlaps from equation (3.1). The smaller the m, the fewer pairs of iterations there will be and vice versa. Formally, denote the predicate for asynchronous duet pairing with minimum overlap ratio parameter m as

$$is\_pair(i^A, i^B) = True \iff (i^A, i^B) \in O_m^{e,b}.$$

Analysis of how many overlaps there are for different values of m is in section 5.2.2.

Each pairing method returns a set of paired iterations for an experiment $M_t^{e,b}$,

$$Pairs(M_t^{e,b}) = \{(i^A, i^B) \mid is\_pair(i^A, i^B),\ \forall\, i^A, i^B \in M_t^{e,b}\}.$$

Second, paired iteration times are subtracted to determine the difference between all pairs which yields random sample X with distribution F_X

$$X = \{i^A - i^B \mid (i^A, i^B)\ \in\ Pairs(M_t^{e,b})\}.$$

Furthermore, Bulej, Horký, and Tůma [11] recommend using *run means* sampling instead of *runs merged* sampling. The runs merged method uses all paired iterations from all runs as a sample. On the other hand, run means aggregates paired iterations in a paired run using statistical methods such as arithmetic mean and uses those as samples. Given r runs each with i iterations *runs merged* results in $r * i$ observations and *run means* results in r observations.

We decided to use *run means* sampling because we have the same amount of runs for each benchmark but not identical iteration counts across suites.

These observations are used to create 95% confidence interval using non-parametric bootstrap with 9999 draws with replacement and arithmetic mean as

statistics.[1] For A/A runs, we expect distribution F_X to be centered around 0. Therefore, if the confidence interval encompasses 0, we consider the workloads equal, otherwise report a difference.

The confidence computation in Bulej et al. [4] differs for synchronous duet because it uses the difference of paired iteration durations together with arithmetic mean instead of iteration duration ratio with geometric mean. This change simplifies the comparison process because the samples of different benchmarking methods have the same units.

Variability comparison using CI

Another use of these confidence intervals is to compute their relative width to compare the variability of different measurement methods similar to Bulej et al. [4].

$$CI_{relative\_width}(X) = \frac{CI_{width}(X)}{mean(X)}. \tag{4.2}$$

4.3.3 Mann-Whitney u-test

The next method to evaluate regression in an A/B run is to use hypothesis testing. One such test used for the purpose of A/B performance evaluation is Mann-Whitney u-test [11, 2]. It is a non-parametric test of the null hypothesis that two samples come from the same distribution. One sample is the iteration durations of pair A, and the other is of pair B.

We used u-test implementation from `scipy` python package [2] in its two-sided variant and decided to reject the null hypothesis if p-value is less than 5%. Hence, if p-value is below 5% for a given benchmark, test method, and environment reject the null hypothesis — sample distributions are distinct enough, thus we report a performance change.

4.4 Minimal detectable slowdown

Laaber, Scheuner, and Leitner [2] explore the minimal detectable slowdowns (MDS) for a sequential micro-benchmarking method to determine which micro-benchmarks are viable for cloud benchmarking. We use this method to determine the viability of duet methods compared to sequential methods in the cloud environment.

[1] We used the bootstrap method from python `scipy` package `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bootstrap.html` that uses a bias-corrected and accelerated method to compute the CI by default.

[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html`

First, A/A test results are adjusted to emulate a scenario where pair B is s times wlog slower — emulating A/B test results where version B has a performance regression. For all iterations running version B, iteration runtime is multiplied by s. Formally we adjust the experiment M^e as

$$M^e_{slowed} = \{(R^A, i^B * s \mid \forall i^B \; in R^B) \mid \forall (R^A, R^B) \in M^e\}.$$

Increased iteration times of pair B affect the overlap of A and B during the asynchronous duet. Therefore, recompute each B's iteration's start and end timestamps, assuming the gaps between iterations remain unchanged.

Then the altered results can be compared as if they were A/B runs using the confidence interval test or Mann-Whitney u-test while the slowdown is known.

Chapter 5

Experimental evaluation

We compare the benchmarking methods on multiple experiments in multiple environments. All experiments were run on super-harness described in chapter 3. This super harness executes A/A benchmarks from multiple benchmark suites using sequential, synchronous, and asynchronous duet benchmarking methods.

5.1 Experiment setup

Benchmark suites used in the experiments are Renaissance, Scalabench, DaCapo, and SPEC CPU 2017. Latter three are also part of Bulej et al. [4], and two of those — Scalabench and DaCapo are representatives of synchronous duet method as their harnesses were adapted to work with `duetbench`.

Renaissance benchmark suite was selected as a modern JVM suite with a focus on parallel workloads among others[6]. We used version 0.13 and had to exclude `neo4j-analytics` benchmark as it failed to execute on our configuration. That leaves 24 benchmarks in total from the Renaissance suite.

Scalabench ($v.9.12$) and DaCapo ($v.9.12$) are both JVM benchmarks that have significant overlap in benchmarks but are the only ones that have their runtime altered to run synchronous duet. For DaCapo and Scalabench we excluded `batik`, `eclipse`, `tomcat`, `actors`, and `jython` (the last two are only in Scalabench). That account for 10 and 21 benchmarks in total for DaCapo and Scalabench suites, respectively. Both suites were run with `--novalidation` parameter to tighten the workload loop.

All JVM-based benchmarks were run in OpenJDK 11.0.11 JVM with fixed heap size and disabled garbage collection ergonomics; other virtual machine settings were left at their defaults.

SPEC CPU 2017 is the sole non-JVM benchmark — representative of natively compiled C, C++, and Fortran benchmarks. We run its rate workloads in peak tuning us-

ing the `onlyrun` action[1] that skips additional non-workload steps. SPEC CPU was not intended to run in a container, and hence we faced multiple issues setting it up. From build and training errors in installation step (`503.bwaves_r`, `510.parest_r`, `521.wrf_r`, `527.cam4_r`) to runtime memory requirements (`502.gcc_r`, `523.xalancbmk_r`, `526.blender_r`). Overall we run 16 benchmarks from SPEC CPU 2017.

5.1.1 Docker setup

The Docker image for Renaissance suite is based on the official Renaissance docker image [2] based on OpenJDK 11, Scalabench and DaCapo images are built from `openjdk:11.0.8-jdk`[3] image, but it is adjusted to run synchronization agent described in section 3.1.1. The Docker image for SPEC CPU is based on `ubuntu` image, and the whole suite installation is done in the container build step using a configuration with `gcc`; the full configuration is available in the attached source code.

Docker containers run with no additional options than mentioned in chapter 3. In particular, there are no reservations on CPU or memory. There are multiple reasons for this (1) benchmarks themselves check the number of available CPUs and set their parallelism accordingly, but that information is returned by the OS and thus would be the same for both duet pairs regardless of docker allocations, (2) the CPU reservations would rule out use of smaller cloud instances, (3) impact symmetry with completely fair scheduler should allocate equal resources to both benchmarks, hence simplifying setup.

All the measurements used Podman containers.

5.1.2 Run summary

One of the goals of the duet methods is to enable benchmarking in varying environments:

cloud environments is represented by AWS EC2 `t3.medium` instances,

bare-metal environment consist of dedicated bare-metal servers,

shared-VM environment is self-hosted virtualized environment with no special performance measurement treatment that is actively used by other students.

AWS `t3.medium` instance was chosen because the `t2` instances were not powerful enough to run two SPEC CPU benchmarks in docker side by side and `t3.medium` turned

[1] Disclaimer: SPEC CPU prohibits the publication of results without verification, but for our use case, verification could affect our results negatively, and we don't report SPEC CPU scores.

[2] `https://hub.docker.com/r/renaissancebench/buildenv`

[3] `https://hub.docker.com/_/openjdk`

out to be the cheapest alternative able to do so. The measurements were parallelized across multiple instances in each environment except for the shared-VM environment, as there is only one instance. Table 5.1 summarizes the configuration of selected platforms and the timeframe of the measurements.

Environment	Instances	CPU	vCPUs	Memory	Measurements
AWS t3.medium	10	Intel(R) Xeon(R) Platinum 8259CL Intel(R) Xeon(R) Platinum 8175M	2	4.0 GiB	Jul 12 – Jul 19 2022
Bare metal	8	Intel(R) Xeon(R) CPU E3-1230 v6	4	32 GiB	Jun 17 – Jun 22 2022 Dec 8 – Dec 10 2022
Teaching	1	Intel(R) Xeon(R) CPU E5-2620 v4	4	16 GiB	Dec 3 – Dec 20 2022

Table 5.1 Summary of selected platforms.

We ran 20 runs of each benchmark using Randomized Multiple Interleaving Trials method [3] in each environment with every available measurement type. The only exception is the shared-VM environment, where we decided not to run SPEC CPU due to disk and runtime requirements. That sums up to more than 100 days of machine time executing 61 unique benchmarks.

Each benchmark run has been executed for a given amount of iterations or until it reaches a timeout. Timeout was 5 and 30 minutes for JVM-based and SPEC CPU suites, respectively. The iteration limit for Renaissance was set to default which is supposed to be reasonable for that benchmark. Scalabench and DaCapo had a hard limit of 50 iterations, while SPEC CPU had a limit of 10. Deep dive into runtime is in the next section.

5.2 Measurements

The following sections provide answers to research questions proposed in chapter 2. Often, due to a considerable number of benchmarks, we aggregate the results by suites or measurement methods. For detailed, per benchmark, results see section 5.3. All data in the following figures are subject to data preprocessing described in section 4.2 unless stated otherwise.

5.2.1 RQ1: Runtime savings

The first research question regards the duet methods' overall runtime reduction compared to the sequential method. The presumption is that since duets run concurrently, they can be up to twice as fast as the sequential method.

Runtime is computed on raw measurements from the start of the particular run to its end. For the sequential method, add the duration of pairs A and B since pairs

don't run in parallel; for duet methods, take a sooner start and a later end of a run to compute the duration

$$\begin{aligned} runtime^{e,b}_{seqn} &= \{end(R^A) - start(R^A) + end(R^B) - start(R^B) \mid \\ &\quad (R^A, R^B) \in M^{e,b}_{seqn}\}, \\ runtime^{e,b}_{aduet} &= \{min(start(R^A), start(R^B)) - max(end(R^A), end(R^B)) \mid \\ &\quad (R^A, R^B) \in M^{e,b}_{aduet}\}. \end{aligned}$$

Then runtime speedup for particular benchmark $b \in B$ in an environment $e \in E$:

$$runtime\_speedup^{e,b} = \frac{mean(runtime^{e,b}_{seqn})}{mean(runtime^{e,b}_{aduet})}. \tag{5.1}$$

Figure 5.1 shows the runtime speedup of asynchronous duet run. As it turns out, this heavily depends on the environment and the type of benchmark. Some benchmarks are almost twice as fast, especially from the SPEC CPU suite in stable environments. However, there is significant variability, and even in the bare-metal environment, some benchmarks show no speedup. Duet in the cloud environment is even slower for some benchmarks. This is likely the cause of vCPU discrepancy as `t3.medium` has only 2 vCPUs while other environments have 4.

Overall, the median speedup is 5%, 73%, and 53% for cloud, bare-metal, and shared-VM environments, respectively. These results show that using duet methods can significantly reduce the machine time required to run benchmarks.

5.2.2 RQ2: Pair overlaps

The second research question aims to understand the nature of asynchronous duet iteration overlaps. However, as stated in section 3.3.2, not all overlaps are the same. It is essential for the CI computation from 4.3.2, where the minimum overlap ratio is used for pairing asynchronous duet iterations. Formally, define the relative overlap time for benchmark $b \in B$ in environment $e \in E$ and minimum overlap ratio m as

$$relative\_overlap\_time^{e,b}_{m} = \frac{\sum\limits_{(i^A, i^B) \in O^{e,b}_m} 2 * overlap\_time(i^A, i^B)}{\sum\limits_{(R^A, R^B) \in M^{e,b}_{aduet}} \left(\sum\limits_{i \in R^A} i + \sum\limits_{i \in R^B} i \right)}. \tag{5.2}$$

The overlap time in the nominator computes the sum of filtered overlaps from equation (4.1). This sum is doubled because the overlap is a symmetric relation, and the denominator accounts for the runtime of both pairs; this way, the $relative\_overlap\_time^{e,b}_{m} \in [0, 1]$ and represents a percentage of overlapping workload runtime, not just overlapping harness.

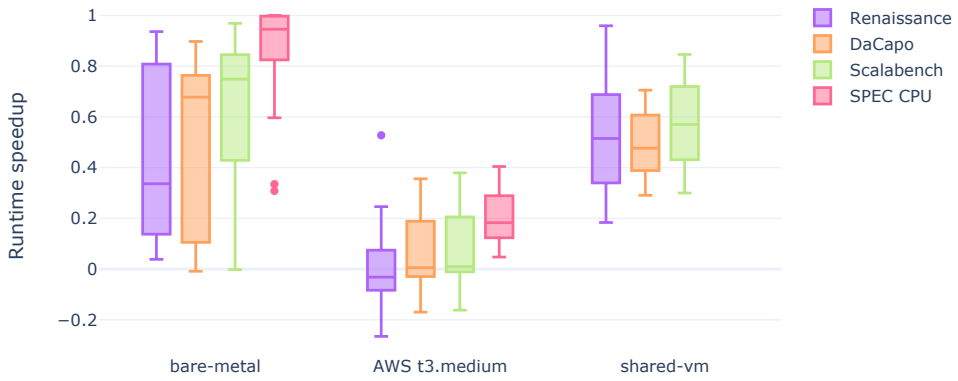

Figure 5.1 Runtime speedup of asynchronous duet method across different environments and suites. This is not subject to data preprocessing from section 4.2 as it captures raw measurements time.

Figure 5.2 shows how much does minimum overlap ratio affect the relative overlap time. SPEC CPU has very good overlap coverage, which suggests a low variance of iteration durations. Other suites show a steady decline as the minimum overlap ratio reaches 90% overlap time is less than 20% of total iteration time, hindering the runtime improvements from section 5.2.1. However, the minimum overlap ratio of 30% or 50% retains above 80% or 60% of iteration runtime, respectively, which seems like a good fit for the minimum overlap ratio parameter.

5.2.3 RQ3: Accuracy improvements

The third research question concerns the accuracy of regression detection using different benchmarking methods across different environments. The accuracy is inherently dependent on the variance of measurements. Therefore, the first method of comparison uses relative standard deviation CV as described in section 4.3.1.

Figure 5.3 shows CV of different benchmarking methods across different environments for all benchmarks. It shows that duet methods have comparable variance, and even though the sequential method usually has smaller CV in all environments, there is no significant difference between methods or environments. SPEC CPU has the smallest CV, which is expected since it is the only natively compiled suite with longer runtime. A closer look shows common benchmark outliers, namely `luindex` in the shared-VM environment in both Scalabench and DaCapo suites, `fop` from DaCapo in shared-VM and cloud environments, and the `dotty` from Renaissance suite in the

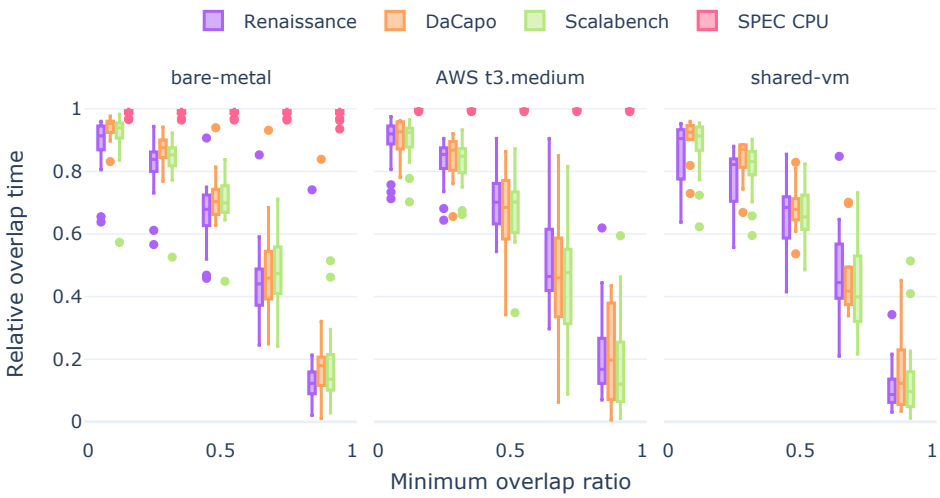

Figure 5.2 Box plot of benchmark relative overlap time from equation (5.2) based on minimum overlap ratio.

cloud environment which has maximum CV of 0.8.

Next, we can determine the accuracy of each method using A/A detection correctness of CI test from section 4.3.2. As shown in section 5.2.2 the value of the minimum overlap ratio can significantly influence overall overlap time. Figure 5.4 shows success of CI test in determining A/A run based on minimum overlap ratio. The first important takeaway is that the CI test has reasonably good A/A detection capabilities across all environments. However, steep rises or declines imply that multiple benchmarks in the given suite were on the edge of CI straddling 0, which is expected as higher values of minimum overlap ratio decrease the overlap runtime significantly 5.2.

Scalabench in the shared-VM environment comes the off worst with only 70% A/A detection rate closely followed by SPEC CPU with only 81%. The low correctness of the CI test for the SPEC CPU suite is likely a result of its low variance and small relative CI width, as shown in figure 5.5. Based on these findings, we set the minimum overlap ratio to 40% in the following CI computations.

Figure 5.5 shows relative CI width 4.3.2. The width of CI compares iteration variance across environments and the variance of CI computation in particular. It differentiates the shared-VM environment despite the fact that all environments had low CV 5.3. Surprisingly, the CI width for the cloud environment looks very similar to bare metal measurements.

The overall results of the CI test are in Figure 5.6. CI test achieves the best results in the cloud environment, but the overall correctness of A/A detection straddles 80%. One particular thing to point out is that the sequential method is not worse than duet

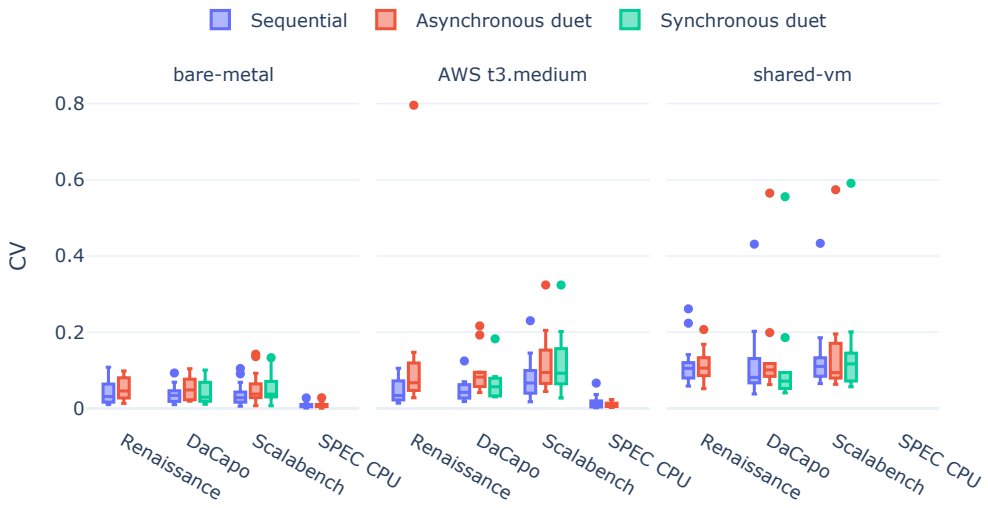

Figure 5.3 Box plot of CVs of benchmarks from respective suites.

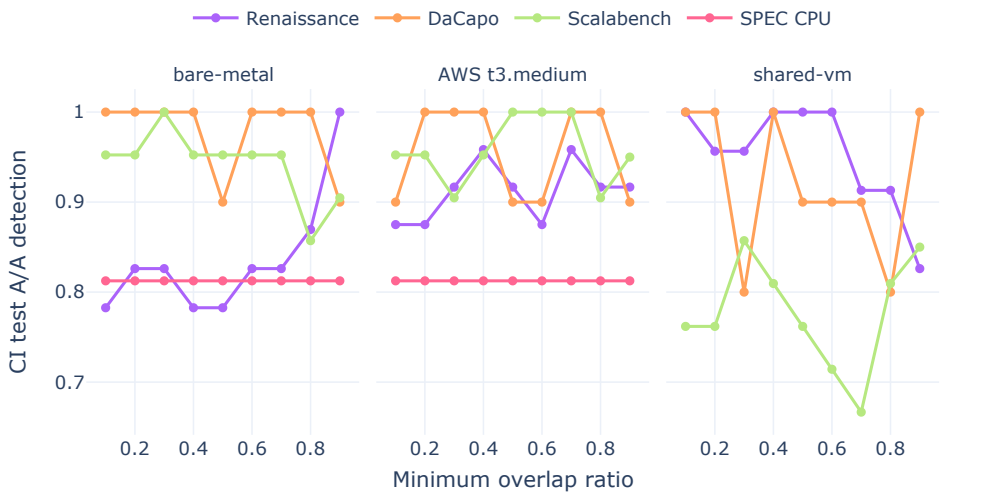

Figure 5.4 The effect of the CI test parameter minimum overlap ratio on correct A/A run detection for the asynchronous duet. The y-axis shows the proportion of benchmarks that the CI test considers equal in performance — does not report a regression. Note that each suite has a different amount of benchmarks. For example, a 20% change in DaCapo A/A detection rate means that the CI test changed its verdict for two benchmarks. Benchmark count for each suite is in section 5.1.

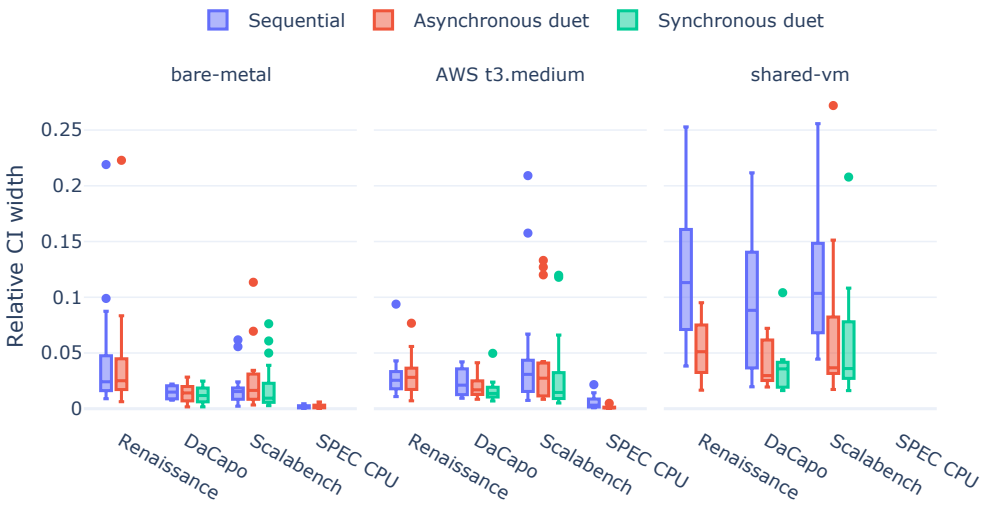

Figure 5.5 Box plot of benchmark relative CI width from equation (4.2). CI for asynchronous duet uses the minimum overlap ratio of 40%.

methods and sometimes even outperforms duet methods.

Figure 5.7 shows success of u-test described in section 4.3.3 in determining A/A run. The u-test seems to be doing much worse compared to the CI test in the bare-metal environment in particular. However, it has a 100% success rate on the SPEC CPU suite in the cloud environment for both sequential and asynchronous duet methods.

5.2.4 RQ4: Minimal detectable slowdowns

The last research question regards minimal detectable slowdowns (MDS). First, the results have to be adjusted as described in section 4.4. Then evaluate regression detection capabilities using CI test and u-test, with no slowdown, the values are the same as in figure 5.6 and figure 5.7 for CI test and u-test respectively. The bigger the slowdown, the sooner the A/A match should drop to 0% — no benchmarks were considered equal in performance.

Figure 5.8 shows MDS using the CI-test. The SPEC CPU has MDS of 1% slowdown for both benchmarking methods and in both environments, but its initial A/A detection is only at approximately 80%. The rest of the benchmarks have somewhat similar trends of MDS, with the sequential method sometimes lacking slightly behind duet methods. Overall almost all benchmarks using duet methods have MDS of 5% in bare-metal and cloud environments. However, the sequential method in the shared-VM environment performs significantly worse than duet methods as it fails to

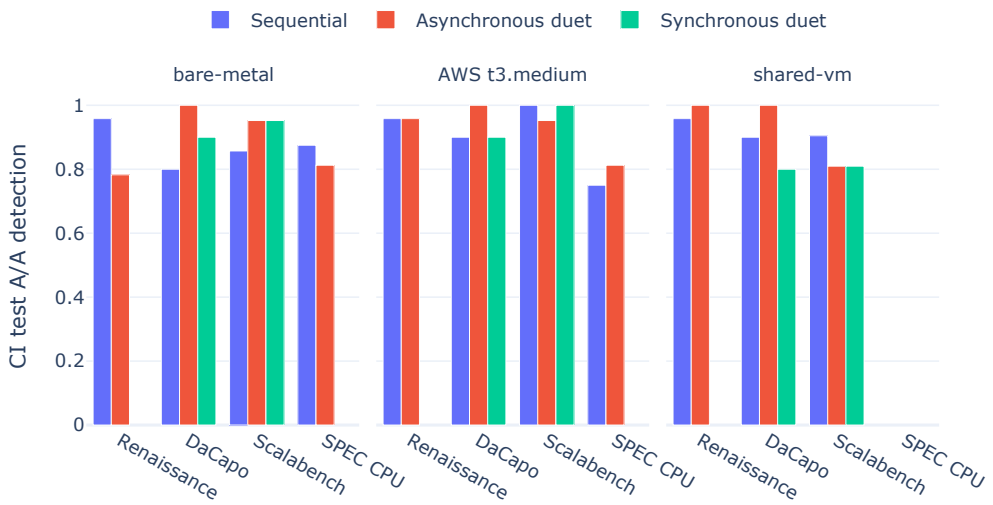

Figure 5.6 The success of the CI test in determining A/A run. The y-axis shows the proportion of benchmarks that the CI test considers equal in performance — does not report a regression.

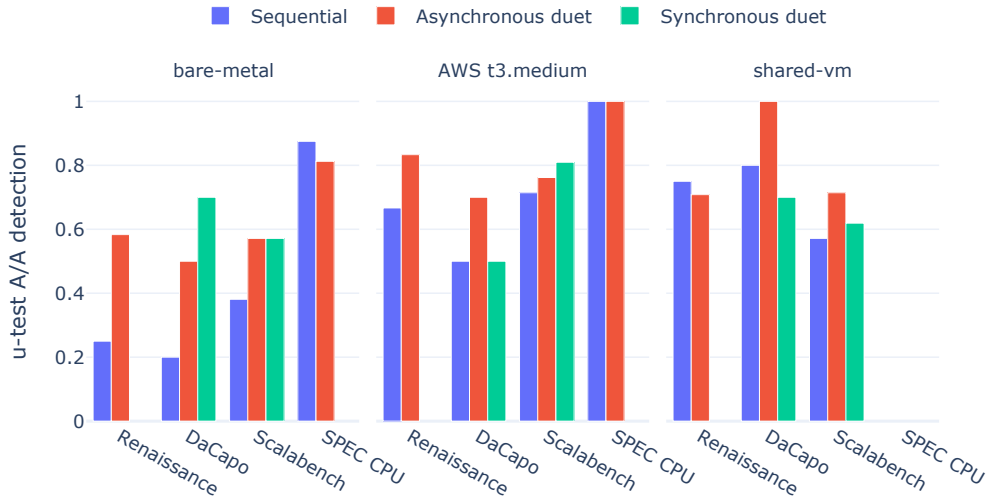

Figure 5.7 The success of u-test in determining A/A run. The y-axis shows the proportion of benchmarks that the u-test considers equal in performance — does not report a regression.

detect even some 15% regressions likely because of the biggest relative CI width in this environment as shown by figure 5.5.

Since u-test showed a reasonable A/A detection rate only for the SPEC CPU suite, we showcase MDS for this suite only in figure 5.9. Similar to the CI test, u-test can detect $1-2\%$ slowdown for the whole SPEC CPU suite both in the cloud and bare-metal environment.

5.3 Evaluation automation

To streamline the duet evaluation process `duet` package provides a set of interactive Jupyter notebooks that give a more detailed view of the results. Specifically, they visualize data behind each research question separately per benchmark.

One example from these notebooks is the figure 5.10 which shows the confidence intervals from CI test section 4.3.2 for SPEC CPU benchmarks. It clearly shows that some benchmarks are much more variable than others, particularly in the cloud environment.

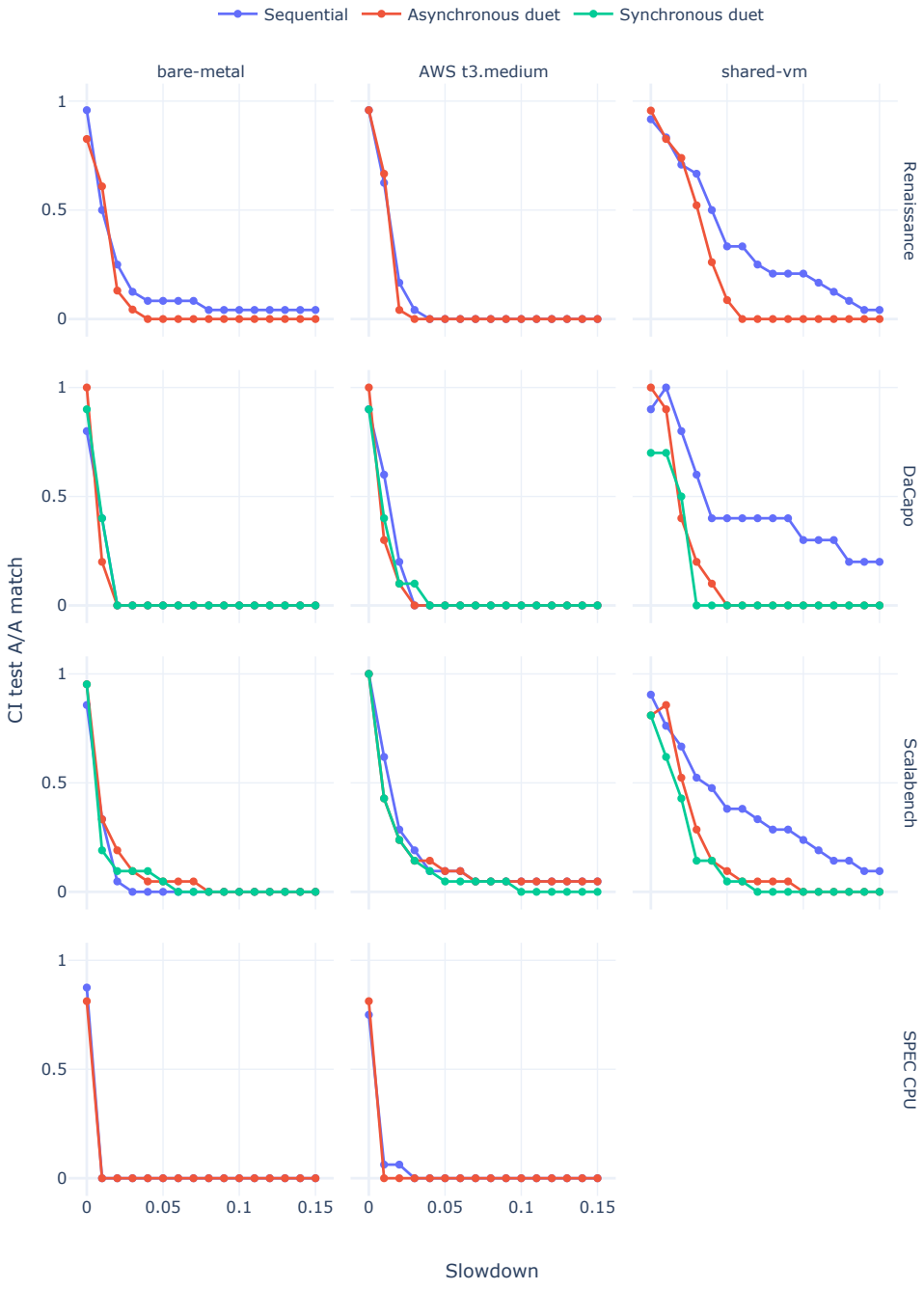

Figure 5.8 Facet plot of minimal detectable slowdowns using the CI test. The x-axis has the artificially applied slowdown on one pair, starting from no slowdown (A/A run) to 15% slowdown. The y-axis shows the proportion of benchmarks from a given suite that the CI test considers equal in performance despite the applied slowdown.

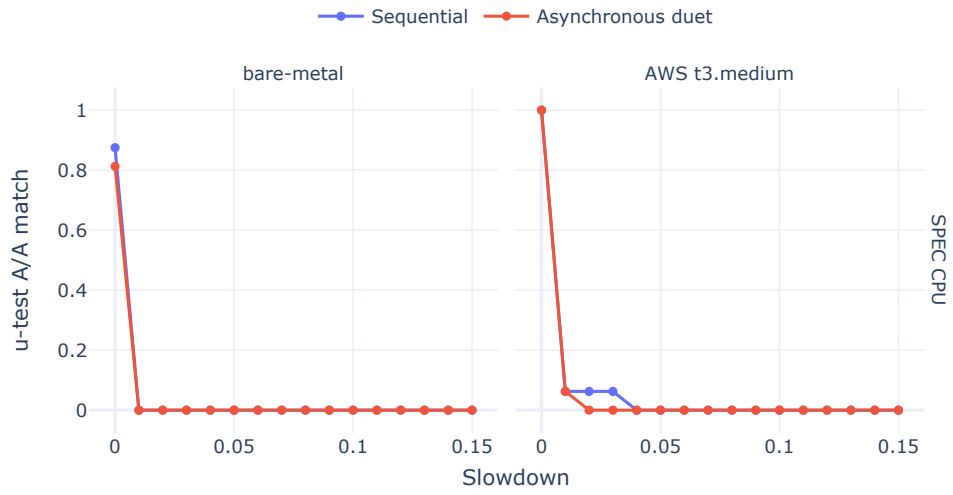

Figure 5.9 The minimal detectable slowdown of SPEC CPU suite using u-test. Note that u-test has 100% accuracy in the cloud environment, and it can detect a 2% slowdown for all SPEC CPU benchmarks using asynchronous duet.

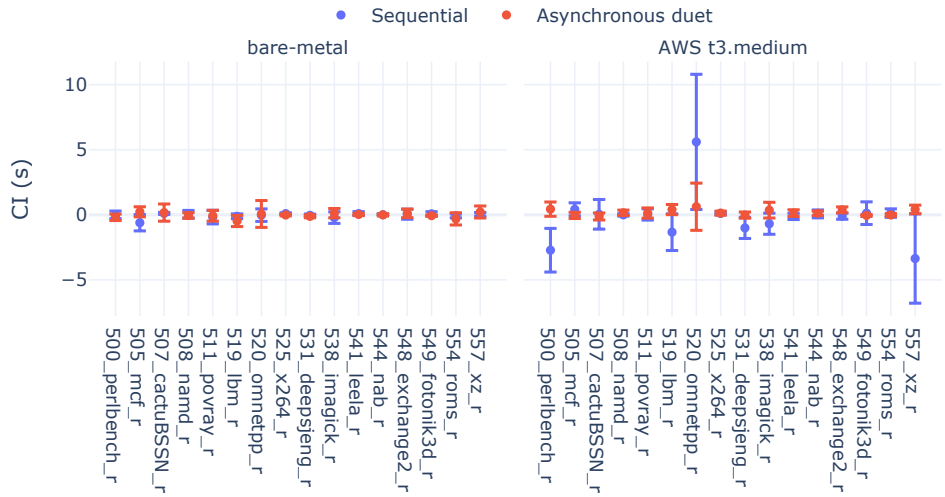

Figure 5.10 Confidence intervals used in the CI test for the SPEC CPU suite. Benchmarks are considered equal in performance if the CI contains the point 0; otherwise, the test reports a regression. The y-axis shows the bootstrapped 95% confidence interval of the differences between paired A/B iteration times in seconds as described in section 4.3.2.

Conclusion

Our main goal was to evaluate the behavior of a new duet method variant that does not synchronize iterations. In particular, we wanted to answer questions from section 2.2.

First of which was better accessibility of duet methods. We achieved that by creating a benchmark super-harness tool that can run benchmarks from multiple suites in a portable containerized way using both sequential and duet methods. This tool was used to run measurements in cloud, bare-metal, and shared virtual machine environments.

The second goal focused on the runtime reduction potential of duet methods. Section 5.2.1 shows that most of the natively compiled and longer running benchmarks from SPEC CPU suite ran $80-100\%$ faster when using the asynchronous duet method compared to using the sequential method reducing the costs by up to 50%. However, results varied across benchmark suites and environments. Cloud environment, in particular, did not do well in runtime speedups and even manifested a slowdown for some benchmarks. Although this was likely caused by cloud instances having two vCPUs while other environments had 4. Overall average speedups were 8%, 60%, and 54% for cloud, bare-metal and shared-VM environments respectively.

The third and last goal focused on the accuracy of the asynchronous duet method. Initially, we looked at how asynchronous iterations overlap in section 5.2.2. Overlap quality was determined by the proportion of overlap duration and time of overlapping iterations being above a certain threshold — minimum overlap ratio. It turned out that SPEC CPU benchmarks overlapped almost all the time as if the iterations were synchronized. JVM-based suites with shorter and more variable benchmarks create overlaps of various sizes.

Section 5.2.3 looked at iteration variability of different methods and A/A run detection capabilities. The overall variability of the benchmark scores across different environments was captured by the coefficient of variability and the relative width of confidence intervals from the confidence intervals test. The coefficient of variability turned out to be below 0.2 for almost all tested benchmarks in all environments, with SPEC CPU being the least variable, as expected. We did not observe any significant difference between cloud and bare metal. However, the width of confidence inter-

vals differentiates the shared virtual machine environment as the most volatile, with the sequential method having the most extensive variability. Again the cloud and bare-metal environments look very similar when compared by confidence interval width.

A/A detection accuracy used an adjusted confidence interval test and u-test. The confidence interval test for an asynchronous duet was parametrized by the minimum overlap ratio pairing of 40% based on empirical observations. The confidence interval test was able to successfully detect 90% of all JVM-based benchmarks and 80% of SPEC CPU benchmarks. The relatively low success rate of detecting SPEC CPU A/A runs is likely due to its low variability hence relatively low confidence interval width that might not include point 0. To accommodate this, the confidence interval test can be extended to consider the error by which the confidence interval missed 0 and account for some leeway.

Mann-Whitney u-test was much more sensitive to difference; hence it has much worse A/A accuracy overall with only 62% correctly detected JVM-based benchmarks. However, the sensitivity proved useful for SPEC CPU benchmarks which had 100% A/A detection in the cloud environment and 84% in bare-metal. Method comparison with u-test rules asynchronous duet as the best with 74% followed by the synchronous duet with 65% and sequential with 60% A/A detection accuracy.

Last, we examined the minimum detectable slowdowns (MDS) factor using artificially slowed down A/A runs in section 5.2.4. Duet methods outperformed sequential method with the ability to detect $3-5\%$ slowdown for JVM-based benchmarks and even 1% regression for SPEC CPU benchmarks, while the sequential method had issues detecting even 10% regressions in our shared virtual machine environment. However, the results were similar in the cloud and bare-metal environments.

Overall, removing the synchronization requirement does not hurt synchronous duet accuracy improvements. With possibilities for significant runtime and cost reduction while maintaining high regression detection capabilities, we believe the asynchronous duet method is an attractive novel benchmarking approach that is usable even in variable environments. Additionally, the `duet` python package provides a generic way of running and processing benchmarks in the containerized environment using sequential and duet methods.

Future work

One thing we did not manage to include in this thesis due to restricted resources is broader coverage of cloud providers and instance types with higher CPU count in particular. This could prove that runtime and cost reductions in the cloud are similar to what we observed in our shared virtual machine infrastructure. More measurements could also examine the required number of runs to achieve reasonable accuracy. In

our experiments, we used 20 runs per benchmark, but fewer runs may be sufficient to detect a certain level of MDS.

We did not examine the synchronized interference of benchmarks in depth. This interference can manifest predominantly in the asynchronous duet method, as shown in figure 2.3. One would need to look closer at benchmark workloads and their bottlenecks to see if workloads with different bottlenecks interfere in some synchronous manner while running in a duet.

Another thing that needs a closer look at individual benchmarks is an analysis of benchmarks on which the duet did not do well. Ideally, some specific common workload traits, such as high parallelism, would cause issues for the duet.

The last thing we mention is an exploration of methods that would allow determining the regression severity. The method of minimal detectable slowdown allows us to determine what slowdowns we can reliably detect. However, in a generic A/B run, a slowdown is unknown.

Bibliography

[1] Philipp Leitner and Jürgen Cito. "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds". In: *ACM Transactions on Internet Technology (TOIT)* 16.3 (2016), pp. 1–23.

[2] Christoph Laaber, Joel Scheuner, and Philipp Leitner. "Software microbenchmarking in the cloud. How bad is it really?" In: *Empirical Software Engineering* 24.4 (2019), pp. 2469–2508.

[3] Ali Abedi and Tim Brecht. "Conducting repeatable experiments in highly variable cloud computing environments". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 287–292.

[4] Lubomír Bulej et al. "Duet benchmarking: improving measurement accuracy in the cloud". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 100–107.

[5] Vojtěch Horkỳ et al. "Utilizing performance unit tests to increase performance awareness". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 2015, pp. 289–300.

[6] Aleksandar Prokopec et al. "Renaissance: benchmarking suite for parallel applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[7] Stephen M Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 169–190.

[8] Andreas Sewe et al. "Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, pp. 657–676.

[9] Oracle. *GraalVM Repository at GitHub*. 2023. URL: `https://github.com/oracle/graal`.

[10] Augusto Oliveira et al. “Perphecy: Performance Regression Test Selection Made Simple but Effective”. In: *Proc. of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan, 2017.

[11] Lubomír Bulej, Vojtech Horký, and Petr Tůma. “Do we teach useful statistics for performance evaluation?” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017, pp. 185–189.

[12] Lubomír Bulej, Vojtěch Horkỳ, and Petr Tůma. “Initial experiments with duet benchmarking: Performance testing interference in the cloud”. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pp. 249–255.

[13] Chandandeep Singh Pabla. “Completely fair scheduler”. In: *Linux Journal* 2009.184 (2009), p. 4.

[14] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[15] *Pod manager tool "podman" website*. 2023. URL: `https://podman.io`.

[16] Tomáš Drozdík. *Duet package repository at Github*. 2023. URL: `https://github.com/TomasDrozdik/asynchronous-duet`.

[17] Tomáš Drozdík. *Duet package documentation*. 2023. URL: `https://github.com/TomasDrozdik/asynchronous-duet/wiki`.