Measuring up to Stability

Guidelines towards accurate energy consumption measurement results of Rust benchmarks

Rens Hijdra

Technische Universiteit Delft

TU Delft

Measuring up to Stability

Guidelines towards accurate energy consumption measurement results of Rust benchmarks

by

Rens Hijdra

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday May 7, 2024 at 14:00.

Student number:	4592794	
Project duration:	November 16, 2022 – May 7, 2024	
Thesis committee:	Prof. dr. A. van Deursen,	TU Delft, supervisor
	Dr. L. Miranda da Cruz,	TU Delft, daily supervisor
	Dr. C. Laaber,	Simula.no, daily co-supervisor
	Dr. M. Weinmann,	TU Delft, external member

TU Delft

Preface

This research was done to obtain a Masters degree, but also to help further the progress towards a greener future. This research is a small step towards improving energy benchmarking but has been a leap for me. This project has at times tested my patience, but I have enjoyed the process.

I want to thank Luis Cruz and Christoph Laaber for their weekly support and feedback meetings. I want to thank my Thesis Committee, Prof. dr. Arie van Deursen, Dr. Luis Cruz and Dr. Michael Weinmann.

I want to thank Lana for her (mental) support and guidance and for carrying me through the thesis; my parents for supporting me throughout the thesis; Thomas for helping me figure out Maths and Statistics and acting as a sanity check; and Jeroen for being a study-buddy and source of suggestions. At last I want to thank everyone who has supported me and wished me luck at some point along the way.

In this thesis I lay out the path I have taken during this adventure into academic research. Firstly I introduce the topic and the research questions. In the second chapter I explain relevant academic work, as well as a technical background of the technologies used in the research. The third chapter details the experimental setup and process of collecting and using the data. The results of the experimental investigations can be found in chapter four, and the answers to the research questions are discussed in chapter five along with suggestions for future work. Finally chapter six concludes this thesis.

Thank you for reading my thesis. I hope my contribution to the field of Sustainable Software Engineering inspires further research and in the contributes to a lower energy consumption some place, some time.

Rens Hijdra
Delft, May 2024

Contents

1 Introduction

Measure twice, cut once: a golden rule in carpentry; for software energy consumption there is no such rule. For software developers there is no clear-cut guidelines on how often they should measure the energy consumption of their code to be sure they have an accurate measurement. In this thesis we aim to provide an answer to this and other questions regarding the stability of energy consumption in software benchmarking.

In 2022 the estimated energy consumption of data centres world wide is estimated to be around 1 percent of the global demand, with an equal fraction in energy related global greenhouse gas emissions (Rozite, V., 2023); half that of the global aviation industries greenhouse gas emissions (Kim, H. and Teter, J., 2023). The energy consumption of datacenters is growing by 20-40% annually and is projected to reach up to 30% of some countries national energy consumption by 2030 (Rozite, V., 2023). Greenhouse gasses are the ”main driver for global warming,” which is ”associated with serious negative impacts on to the natural environment and human health and well-being” (European Commission, 2023).

There are two ways the IT industry can decrease its energy consumption: through hardware, or through software. The hardware route is for is being tackled by manufacturers and academia . The software route part of the ”Sustainable software engineering” movement; which is on the rise as concerns for developers and scientists world-wide grow for the impact global warming will have (Green Software Foundation, 2023, IEEE, 2024, ClimateAction.Tech, 2023). One of the aims of this (emerging) field is to lessen the environmental impact of software (Helmes, 2023) by decreasing its energy consumption.

Before one could focus on decreasing energy consumption, it is important to know the stability of your measurements. Developers need stability in their measurements for a variety of reasons. First of all, it can lead to a misrepresentation of the actual power consumption of the code. Unstable results lead to inconsistencies when comparing results against another, e.g. when comparing two versions of an algorithm. Without stable results, one cannot make sound conclusions.

It would be great if all software always consumed the same amount of energy. One could run the code once and you would know the exact energy consumption. This would enable developers to quickly iterate on changes when working on decreasing the energy consumption of their code. Since the performance of software is not entirely stable (Laaber et al., 2021), we might expect energy consumption to not be stable either. In this research question we investigate whether energy benchmarks produce stable results.

Interviews with developers by Ournani et al., 2020 have found there is a need for:

1. ”[investigation into effectiveness] of static code analysis for energy efficiency purposes.”

2. ”[simple tools that allow for] low-level diagnosis of the [energetic footprint of the] source code and identify the exact problems and solutions.”

3. ”[automation of analysis of and reporting on] the energy footprint along with other metrics like code quality, performance, and tests reports.”

We fill this research gap by creating a easy to use and automatable extension (tool) for gathering energy consumption; investigating the inherent stability of benchmarking energy consumption, and relations between source code and the stability of its energy consumption; and provide research into ensuring that the tool provides sound conclusions.

In this thesis we provide insight and guidelines towards the stability of energy measurements.

The first step towards stability is deciding on the measurement method, for which there are several available options: external hardware, or built-in APIs. To *Keep It Simple*; we provide a low effort method for developers and deliver accurate and timely feedback. We avoid the requirement of additional hardware by providing a drop-in software solution for collecting on-device consumption. This means using model-specific registers (MSRs) that are common on all recent Intel and AMD CPUs. By extending an existing benchmarking framework Criterion-rs, we enable developers to get results with little effort. This drop-in-place extension keep the threshold for getting started with sustainable software low and achieves the goal of being able to start at any moment.

In this thesis we investigated the energy consumption stability of benchmarks from a large set of open-source Rust projects that use the *Criterion-rs* benchmark framework, gathered from *crates.io*.

Just like in software testing, feedback should be timely and actionable. This report investigates the number of samples necessary to accurately determine the stability of software. We suggest a threshold for classifying stability and investigate commonalities between unstable software to provide equip developers with bets practices in writing stable code.

Finally the collected consumption data is used to answer the research questions about whether software can be deemed stable, and if not – what makes software unstable. We derive guidelines for users of our tool, and creators of new other tools, on how to get a stable result on the energy consumption. Furthermore we identify causes of instability in source code and measurement setups.

In this research we show the need for more than one sample of energy consumption due to instability over time when benchmarking. We find that for our experimental setup and set of benchmarks 500 samples gives results that are likely stable at a 1% threshold in their Relative Confidence Interval Width. Running benchmarks with a variable CPU clock-speed can lead to higher variability of measurements; as well as initialising benchmarks with random data. Likewise we investigate the effect of the length of benchmarks on their stability but we can not rule out that this is caused by the experiment setup. Lastly we identify control flow statements and code related to memory accesses as potential large influences of instability.

The code of the experimental setup, the study, and the Criterion-rs plugin have been released on github:

Experiment setup	https://github.com/RensHijdra/RustBenchmarkStabilityThesis/
Study	https://github.com/RensHijdra/RustBenchmarkStabilityThesisLab
Criterion-rs Plugin	https://github.com/RensHijdra/criterion-energy

1.1. Research Questions

1.1.1. Stable Software

In this group of research questions we investigate the stability of software in general. We determine the stability of software, find a number of samples for developers to aim for and investigate the effect of noise on energy consumption stability.

RQ 1.1 - Is energy consumption stable between two random independent measurements?
Performance benchmarks are not necessarily without variability and static source-code features can have an impact in the performance stability of micro-benchmarks (Laaber et al., 2021), and speed-optimised code is theorised to be most energy efficient(Yuki & Rajopadhye, 2014); the question arises whether energy consumption benchmarks share these fluctuations. Is the energy consumption of benchmarks stable across any two samples taken over time?

RQ 1.2 - How long should you measure for a stable result?
In a perfect world a developer Performance benchmarks require multiple measurements to give a stable results because of fluctuations. From the previous question we might conclude that one only needs

two measurements to reach a stable average consumption; or the stability might follow along with an increase in the number of measurements like performance benchmarks (Laaber et al., 2021). For widespread use of energy measurement developers should be able to measure their energy consumption within a reasonable amount of time.

How many samples are required to give a representative value for the stability measure RCIW, in order to determine the stability of a benchmark.

RQ 1.3 - What part of instability can be attributed to noise?
The proposed measurement method is not perfect and tries to minimise the measurement overhead and maximise the accuracy. We want to measure the instability in the overhead of the measurement method and inherent CPU instability. How much measurement noise is present in our method and how does it affect the stability in our measurements?

1.1.2. What are (likely) characteristics of unstable software?

After defining stable software in the previous research questions, we can now point our investigation to the cause(s) of (un)stable software. We look at CPU frequency, number of instructions in the program and source code features and practices.

RQ 2.1 - Does *powersaving mode* affect energy stability?
Using a powersave governor reduces the clock speed and power consumption, and results in less work per time-unit. We investigate whether there is a relation between stability and a lower CPU clock-speed.

RQ 2.2 - Does the number of instructions correlate with stability?
Our implemented benchmark method deals with a varying number of benchmark-runs in a set time-span. This has an influence on the number of runs that is sampled for gaining our data. In this RQ we investigate whether an increase in benchmark program length has an influence in the measured stability.

We use the number of instructions instead of runtime to capture possible effects of memory waits, cache misses, etc. These actions would increase the duration and any possible extra energy consumption could correlate more directly due to the simple physics of $energy = power \cdot time$ (Yuki & Rajopadhye, 2014). By using the program execution path length we prevent the possible concealment of the stability by the duration of an execution.

RQ 2.3 - What code features could make software unstable? – i.e. what can developers do to fix
In this question we investigate whether the unstable benchmarks share similar traits. If there exist source code features with a large impact on the stability of energy consumption, developers can use this knowledge to . We look for precursors for instability that developers should bear in mind when measuring energy consumption.

2
Background

2.1. Related works

This work is inspired by the paper ”Predicting unstable software benchmarks using static source code features” (Laaber et al., 2021), which found that static source code features provide some predictive value with regards to the stability of performance benchmarks.

2.1.1. Temperature and energy consumption

Wang et al., 2023 found that CPU temperature is highly correlated with total server power usage and thus changes in temperature can induce variation in power consumption. They also found that a stable CPU temperature causes a stable total server power consumption. This is corroborated by **powertemp2019** who state that ”when the performance increases, the temperature increases exponentially” as well as ”performance improves as the power consumption increases.” DeVogeleer et al., 2014 have shown ”power/temperature relationship is indeed very likely exponential.” Furthermore, Mesa-Martinez et al., 2008 have shown that an increase in temperature can lead to a higher power consumption.

We aim to stabilise the energy consumption within a measurement cycle by running the benchmark first for a warm-up period of 5 seconds. During this time the CPU should increase due to the applied load. This way it can ”get used to” the load and the temperature starts to move away from the temperature reached by the previous benchmark to this one’s.

2.1.2. Harrell-Davis median

These two robust measures allow us to compare variabilities for non-normal distributions of data. Within these methods, we used the Marrel-Jarritz HD median to account for the multi-modality of our data.

2.1.3. Language features

Predicting benchmark stability should be based on a number of features. The features features considered in this thesis are based on the research done by Laaber et al. (2021). This entails programming language elements and standard library calls that could have a variable performance. For starters we consider file metadata such as lines of code (SLOC) which has been found to predict performance stability. Certain language elements and calls to methods and functions in The Rust Standard Library that increase the complexity of the code are also taken into account. The features are listed in Table 2.1, from which we can see memory manipulation is a large part of the collected features.

2.1.4. Measuring of energy consumption

There are multiple options to measure energy consumption, divided into internal and external solutions. External measurements require additional hardware, which can be in the form of power-measuring plugs (Rohou & Smith, 1999) or voltage-dividers and dedicated chips soldered onto bare components or CPUs attached to multimeters (Mesa-Martinez et al., 2008). Power-plugs and similar solutions have a low power- and temporal resolution and often require handwork to sample. Soldering electronics to a

Type	Domain	Feature	Type	Domain	Feature
Language Features	Function Definitions	abi	Standard Library Calls	Memory Manipulation	core::pointer
		unsafe			core::reference
		await			core::str
		async			{core,alloc,std}::slice
		closure_capture			{core,std}::array
		closure			{core,alloc,std}::alloc
		closure_async			{core,alloc,std}::borrow
		closure_const			{core,std}::mem
		closure_static			{core,std}::ptr
	Control Flow	match			{core,std}::clone
		match_arm_pat			{core,alloc,std}::boxed
		match_arm_guard			std::box
		loop_for			{alloc,std}::vec
		loop_while			{core,alloc,std}::string
		loop_inf			{alloc,std}::rc
		nested_loop		Environment	std::env
		break			{core,alloc,std}::ffi
		continue			std::os
		if		Control Flow	{core,std}::iter
		else			{core,std}::result
		try		I/O operations	std::fs
		try_block			std::net
	Data & Variables	assign			std::io
		let_expr		Arithmetic	std::u128
		return			std::i128
		pointer			core::num::f32
		reference			core::num::f64
		reference_mutable		Asynchronous	{core,std}::future
		tuple			{core,alloc,std}::task
		index			{core,alloc,std}::sync
		array			{core,std}::pin
		field			std::process
		struct			std::thread

Table 2.1: Collected source code features

CPU gives great resolution and control, but requires in-depth hardware knowledge and is an involved process. Both solutions also require physical access to the device-under-test, which is not possible in large corporations renting cloud-based servers;

There are software solutions for determining power consumption on a process level (Bourdon et al., 2013) and method level (Noureddine et al., 2012), but these depend on a model to estimate the consumption. On their recent CPUs, Intel (Intel Corporation, 2016) and AMD (Advanced Micro Devices, 2019) provide access to a Running Average Power Limit (RAPL) interface. On AMD devices this is based on the hardware-written model-specific register (MSR) The MSR special file reading is the lowest level abstraction access we can get without kernel access, the only lower level access being the *rdmsr* assembly instruction.

2.1.5. Randomised Multiple Interleaved Trials

The complete experiment runs for approximately two weeks and encounters possibly a changing environment over this time. The order of execution of multiple trials of the same test ”can systematically bias experiment results” (Duplyakin et al., 2023). We use Randomized Multiple Interleaved Trials (RMIT) to distribute changes in the environment over the different trials to form ”a basis for the fair comparison” (Abedi & Brecht, 2017) between different projects and benchmarks. It is used by Scheuner and Leitner (2018) in performance benchmarking. The RMIT implementation is similar to an (incomplete) Randomised Block Design “Randomized Block Design”, 2008, which is often used to eliminate systematic error from an experiment. In RMIT a trial of all tests or measurements is repeated multiple times. For each trial the order of execution of the tests is randomised. An example of such an RMIT instance is shown in Figure 2.1, where 5 benchmarks (represented by 5 colors) are executed in a randomized order three times for. During each execution 300 samples are collected, thus at the end of the example experiment we end up with 900 energy consumption samples per benchmark.

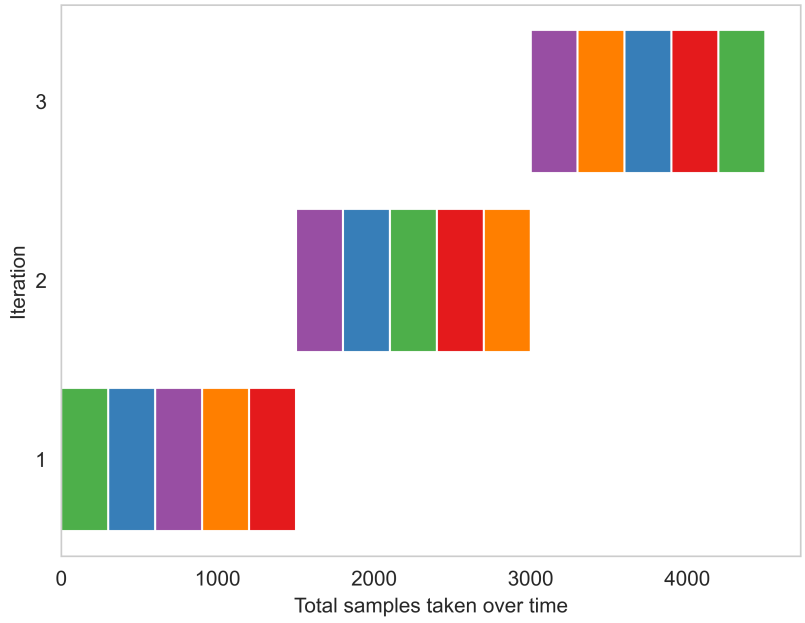


Figure 2.1: Example of 5 benchmarks (colours) executing 3 iterations, each of 300 samples using randomised order.

2.2. Technical Background

2.2.1. Rust

Rust is a programming language aimed at ”performance, reliability and productivity” (**rustlang**). It does this ”with no runtime or garbage collector”, ”rich type system and ownership model [that] guarantee memory-safety and thread-safety” and ”top-notch tooling” among others. In this section we look at a few of Rust’s features that have specifically been used in

Conditional Compilation
Rust has compile time feature gates: `#[cfg(condition)]`. These attributes act on the expression, statement, method or even file that it precedes. The condition of the ”configuration predicate” (The Rust Project Developers, 2024b) is evaluated at compile-time and the annotated thing is either in- or excluded based on result of the evaluation. This allows for example for building binaries for specific platforms, or excluding unnecessary features for certain builds such as only including test code when actually running the tests.

Traits
Traits are Rust’s way of implementing what is called interfaces in some other languages. By implementing a trait for a type, one gives the properties of that trait to the type and thus the type can be used as if it was the trait.

2.2.2. Benchmarking and regression detection

Kalibera et al., 2005: confidence intervals are used for regression detection. For example, a regression (change) in average (median) consumption is detected when the CI of both tests do not overlap. When software has large confidence intervals it makes it harder to detect such regressions. This argues in favour of knowing why software becomes unstable, as well as for determining a static number of samples to use.

2.2.3. Model Specific Registers

The custom measurement method makes use of the *Model Specific Registers* (MSR) `Core Energy Status` and `RAPL Power Unit`, which have the hexadecimal values `0xC001029A` and `0xC0010299`

respectively (Advanced Micro Devices, 2019). The Core Energy Status register contains the number of units of energy consumed, and the RAPL Power Unit contains the Energy Status Units (ESU): a 5 bit unsigned integer used in calculating energy consumption in Joules. The energy is calculated according to the formula $Consumption = CES * 0.5^{ESU} Joules$. The RAPL Power Unit register and Core Energy Status registers are both read only. The latter is updated approximately every 1ms by the hardware.

There are three ways of accessing the values within these registers, which will be discussed in increasing order of duration below.

Assembly The first of way of accessing energy consumption is through the assembly instruction `rdmsr`. This assembly instruction could be in-lined in the measurement library and would cause the least amount of overhead. The instruction required kernel level privileges, which made its use infeasible, unless the code would be redesigned as a kernel module, which is outside the scope of this thesis.

Pseudo-file The second method of reading the register is by opening a file descriptor to a memory-mapped region. This pseudo-file present at the path `/dev/cpu/CPUNUM/msr`, where `CPUNUM` is an integer that represents the cpu number that will be measured [1]. The reading of this file takes in the order of microseconds, which is a negligible overhead with regards to the 1ms minimum update interval. Since this is a file, the permission structure of the linux file system applies. By default the MSR pseudo-file has the octal permission bits set of 600 (or `rw- --- ---`), which means only the owner –`root`– has read and write permission. To get around this a new user-group is made, the current used is added, and the file is set to accept reading by the newly created `msr` group. The commands to achieve this can be found in Appendix B.2

msr-tools The third method of reading the MSR is by using the command line utility `rdmsr`. This binary is installed when installing the Ubuntu package `msr-tools`[2]. This binary reads the special file as described above and adds other functionalities and permission checks. The run-time for this program is higher than the *Special File* method, and would require extra hexadecimal string parsing, which is already present when reading bytes from a file. This value to string conversion is also part of the reason why this binary is slower than reading the file yourself, it creates a double overhead. We tested the performance of the `rdmsr` command using the `perf` performance analysis tool, the command and complete result are available in section B.1. We find an average duration of 0.35 milliseconds per invocation, which is one third of a register update period, and deem this too slow for our uses.

The *Pseudo-file* method for reading MSR values is chosen for its ease of use and its sufficient speed. This method does require elevated permissions, but these can be set once by altering executable flags after compilation. These flags are CAP_ADMIN . After setting the permissions using root access, the program can run as any user and does not require additional privileges.

[1] https://man7.org/linux/man-pages/man4/msr.4.html
[2] https://manpages.ubuntu.com/manpages/jammy/man1/rdmsr.1.html

3

Methodology

In this chapter we discuss the implementation and technical details of the data collection and experiment. We also show the methods used to collect and parse coverage of the benchmarks into the source code features; and the collection of runtime code path length as number of CPU instructions.

The experiment started by collecting projects to test (section 3.3), all of whose benchmarks are configured to use a custom measurement method (subsection 3.6.2). In the experiment execution the projects are listed (subsection 3.4.1), compiled (subsection 3.4.2) and executed in a controlled manner (subsection 3.4.3).

A schematic representation of the methodology can be found in figure 3.1

3.1. Environment

The benchmarks are run on a dedicated machine in a climate controlled datacenter to reduce effects of external influences as much as possible. The machine has a AMD Ryzen™ 5 3600 core, with 64GB DDR4 RAM and two 512 GB NVMe SSDs in RAID-1 configuration. All experiments are run on Ubuntu 22.04, with the 5.19.0-41 kernel. The experiment framework is built using a recent nightly version of the Rust toolchain. All the individual projects are compiled using the toolchain `nightly-2023-04-15-x86_64-unknown-linux-gnu`. The experimental results have been collected in a consecutive timespan of three weeks. All commands were executed in a bash terminal running the terminal multiplexer `tmux`[1] to allow unattended operation of the experiment.

3.2. Criterion Benchmarking Framework

At the time of this research, the benchmarking framework in the Rust language itself has not reached the stable releases of the language. We find that there are not as many projects to be found on GitHub that use the language's own benchmarking framwork. for this reason we looked at external frameworks and found Criterion-rs[2], which is a "statistics-driven micro-benchmarking tool" (Heisler, 2024). This framework has since been forked and is actively maintained under the name `criterion2`[3]. In this research we have searched projects that used versions 0.3 and 0.4 of the original `Criterion`, which were most actively used at that time.

Criterion is a benchmarking tool focused and built for performance benchmarking, i.e. determining the duration of a piece of code. It gives developers the ability to create a benchmark by surrounding their method-under-test with a setup and iteration function. The setup takes care of naming and grouping benchmarks, as well as configuring the benchmark and initialising its data. Listing 3.1 shows and example of a configured benchmark as it would appear in the utilised projects.

In the process of developing a method of collecting energy consumption the iteration function turned out to be most interesting. The iteration function comes in two main flavours: `iter` and `iter_batched`. These are functions that accept a closure[4] as an argument and will execute that

[1] https://github.com/tmux/tmux/wiki
[2] https://github.com/bheisler/criterion.rs
[3] https://github.com/Boshen/criterion2.rs
[4] https://doc.rust-lang.org/book/ch13-01-closures.html

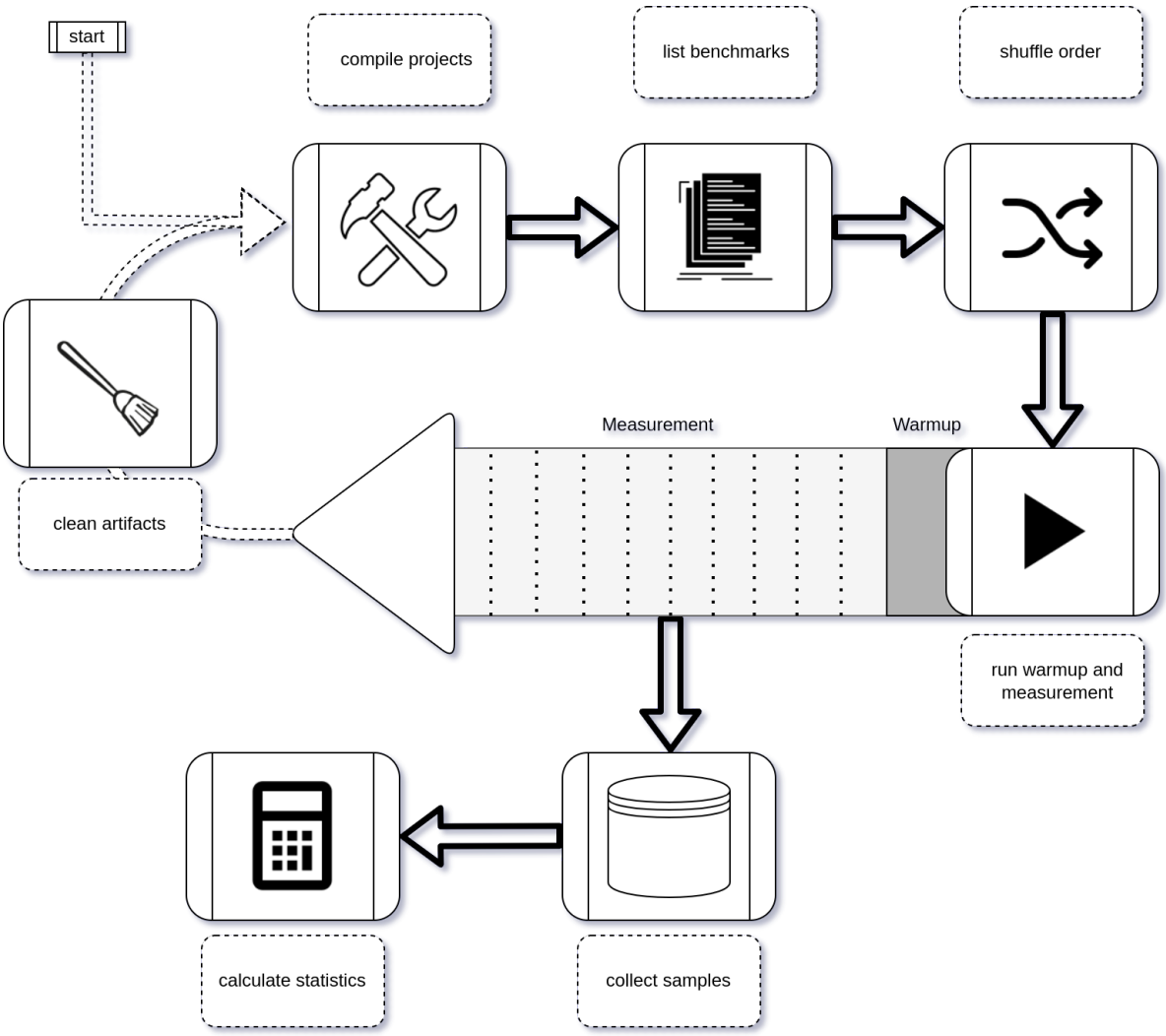


Figure 3.1: Schematic representation of the experiment pipeline

```rust
fn some_benchmark(criterion: &mut Criterion::<Energy>) {
    let mut group = criterion.benchmark_group("benchmark_group");
     group.sampling_mode(criterion::SamplingMode::Flat);
     group.bench_function("benchmark_name", |benchmark| {
         let variable = setup();
         benchmark.iter(|| {
             calculation(&variable)
         })
     });
}
```

Listing 3.1: Example of a benchmark using the Criterion-rs framework

closure to determine its performance. By default Criterion uses a warm-up period during which it determines the approximate duration of the benchmark after which it will run the method-under-test $k \cdot N$ times, where N is determined during the warm-up and k is is an incrementing integer starting at zero up until some predefined time limit has been reached; this is called Linear Sampling. In our research we opt to apply flat sampling to all benchmarks, that is: all samples have the same duration and, depending on performance stability, have the same approximate amount of benchmark invocations. We choose flat sampling for two reasons: linear sampling would start with a sample duration that is too short and measurements would come out as zero due to the 1 millisecond update period of the Model-Specific Register (MSR); and using flat sampling we average out the effects of benchmarks occurring during a MSR update, which would be unpredictable in linear sampling.

The function `iter` has the least overhead: it simply runs the closure for an automatically determined number of times and records some measurement before and after. For methods that require initialisation before each run there is `iter_batched`, which initialises and runs in batches. The measurement is started and stopped before and after each measurement and then added together. Firstly we attempted to track `iter` invocations using kernel probes, as documented in subsection 3.6.1 In subsection 3.6.2 we cover how we implemented our own measurement that works with energy instead of time.

3.3. Project selection

To investigate source code, a dataset of code is needed. The main requirement for projects used in this thesis is that they make use of the benchmarking library *Criterion*. To find projects a list of the reverse dependencies of *Criterion-rs* have been scraped from the Rust library-repository `crates.io` on 2022-11-05 using the library `crates-io-api`[5].

These are sorted by the number of *Stars* on *GitHub*, retrieved using Octocrab[6], which gives a sense of popularity and usefulness of the library. The complete list of 30 projects along with the aforementioned metadata is available in Table A.1 of Appendix A. These projects thirty projects are good for 868 functioning benchmarks. From this list a number of projects is excluded for the reasons of: not compiling, not actually using criterion-rs, or needing a unstable version of rust. For certain projects micro-benchmarks have been left out that take longer than the sample period. For each project we store the following data: project name, git URL, and the most recent release version tag that corresponds to a branch (or master by default). Using this data we can provide consistent data by using the identical versions of the benchmarks for replication of the experiment.

After cloning the projects from GitHub using git we add our plugin as a dependency to the `Cargo.toml` manifest: `criterion-energy = { path = "../criterion-energy"}` or a similar line. We go through the benchmarks and add the following line to all `criterion_group!` macro benchmark definitions: `config = Criterion::default().with_measurement(Energy)` as well as importing our measurement: `use criterion_energy::msr::measurement::Energy`. An example setup can be found in our custom benchmarks as shown in Listing B.1.

3.4. Experiment control

To guide the experiment and ensure repeatability, all steps have been programmed into a Rust program that can execute all necessary functions. In the following sections we walk through the steps needed to execute the experiment, as well as some methods that have been tried but did not end up in the final version. The schematic view of this process is shown in Figure 3.1.

The first step in executing the experiment is compiling the benchmark executables for all projects. The list of benchmarks over all projects is collected and randomly shuffled (see RMIT 3.5). The shuffled list is then sequentially executed. Such an execution is composed of a warm-up period of 5 seconds; followed by a measurement period of 30 seconds, during which the energy measurements are divided over 300 sample periods. This results in samples of 100 milliseconds, during which approximately 100 updates of the Core Energy Status MSR are expected to occur. The 100 millisecond sampling window is a compromise between being long enough to capture a few (i.e. more than 1) execution of each benchmark - and being short enough such that there is not yet a lot of averaging for the shorter

[5]https://crates.io/crates/crates_io_api
[6]https://crates.io/crates/octocrab

benchmarks. This trade-off is further discussed in Threats (5.7.3).

3.4.1. Listing benchmarks

The criterion benchmarking framework replaces the default Rust test-harness. This is something that has already been done by the project maintainers, and the reason these projects have been picked. We can programmatically extract all the benchmark names from the projects using the command `cargo bench -- --list`; this invokes the criterion framework which then lists all available benchmarks. The list of benchmarks is saved as a JSON file per project, along with information about which feature flags (subsection 2.2.1) are required to compile and the source-file that contains the benchmarks.

3.4.2. Compiling benchmarks

For each iteration we recompile all benchmarks since Rust compilations are not completely deterministic by design, for example: the HashMap implementation inserts a random seed at compile-time (The Rust Project Developers, 2024a). We start by removing the previous build artifacts with the command `cargo clean` in each projects folder. To compile a projects we read the JSON file created in the previous paragraph and use the required features to build a compilation command: `cargo bench --bench <benchmark_name> --no-run --message-format=json [--features optional,required,features]`. This command is run in the project directory and outputs machine readable JSON messages from which we collect the name of the generated executable. This executable is saved and used to create a command to run the benchmark in the next section.

Using Cargo's `bench` subcommand with the `-no-run` flag ensures the benchmarks are compiled with the same configuration they would when running the benchmark suite with that subcommand.

3.4.3. Running benchmarks

After compiling the benchmarks and collecting the executable path in the previous section we explain how to execute them here. To measure the consumption of a single benchmark we run each benchmar independently, which means specifying the benchmark in the run command. Criterion selects benchmarks based on a Regex, so we prepend a ând append a $ to the benchmark name in the command. Furthermore we set a warm-up period, the measurement duration and the number of samples to collect during this time. We set a warm-up period of five seconds and a measurement duration of thirty seconds; combined with the 300 samples this results in the desired sample period of 100 milliseconds.

This results in the following command: `<executable_path> --bench --measurement-time 30 --warm-up-time 5 --sample-size 300`. In a later section (3.5.3) we discuss how this benchmark execution gets highest priority on a core to do its work uninterrupted. The implementation of actual energy measurements is discussed in subsection 3.6.2.

3.5. Reducing noise

We look at the methods we used to reduce noise and possible outside influences that could have an effect on the measured values. We randomise te execution order to spread any possible changes in environment over all benchmarks (subsection 3.5.1), we fix the frequency of the CPU to ensure a steady number of executions during each sample (subsection 3.5.2); and limit the process to a single core and limit access to that core for other processes (subsection 3.5.3).

3.5.1. Randomized Multiple Interleaved Trials

We employ the Randomized Multiple Interleaved Trials (RMIT) method for scheduling benchmark execution, which is previously explained in subsection 2.1.5. For each of the thirty iterations the first step is to clean the compilation artefacts of the previous iteration and to recompile all projects sequentially. The order for compilation is the same each iteration. After compiling all projects, a list of commands to execute all the benchmarks is generated. This list is shuffled and the benchmarks are then sequentially executed. The Rust *ThreadRng* used for the shuffling is ”automatically seeded from [the OS] with periodic reseeding” (The Rand Project Developers, 2024), ensuring a different order for each iteration.

3.5.2. Fixed frequency

The frequency with which a micro-benchmark is run is a variable that can be eliminated to create a consistent environment for the experiment. On Ubuntu the CPU frequency governors powersave and

performance will throttle the frequency based on demand and thermal performance. A CPU with changing frequency could affect our measurements in a way where the frequency is not equal throughout one sample and thus could introduce variations in the energy consumption average of a single sample as well as between samples. To eliminate noise between resamples that could occur by this variation we enable the userspace governor and set the frequency of the core to a fixed value. The frequency of the experiment core is fixed at 3.6GHz; which is the highest available non-boost frequency. The frequency is set using the tool `cpufrequtils`[7], with commands `cpufreq-set -c 3 -g userspace`[8] and `cpufreq-set -c 3 -f 3.6G`. For RQ 2.1 (subsection 1.1.2) we also test the influence of the *powersave* frequency governor: `cpufreq-set -g powersave -c 3`. These commands are executed once, before the experiment stats and will last until the settings are changed or possibly a reboot.

3.5.3. Limiting used cores

We want to reduce the effects of other processes on our process as much as possible, by preventing the influence of scheduler interrupts as well as physical influences as much as possible. The AMD architecture design for the Ryzen 5 3600 processor consists of two Core Complexes (CCX) on a single Core Complex Die (CCD). Each CCX consists of a shared L3 cache and three physical cores with individual L1+L2 caches; two virtual cores share one physical core. To minimise physical interference in energy consumption we disable all but two virtual cores: core 0, which cannot be disabled and is needed for system interrupts; and core 3, to run the experiment on. These cores will draw a steady amount of current in their off state and will not heat up the rest of the CPU, as temperature is associated with power consumption (see subsection 2.1.1). These cores are indicated as Processing Units (PU) with Physical Index 0 and 3 in Figure 3.2 and correspond to cpu0 and cpu3 in Ubuntu, respectively. Virtual core 3 is the first core on the CCX that does not house core 0. Pinning the processes to a core is achieved through the use of CPUsets, as will be explained in Figure 3.5.3, where this method is used to assert priority on the selected core.

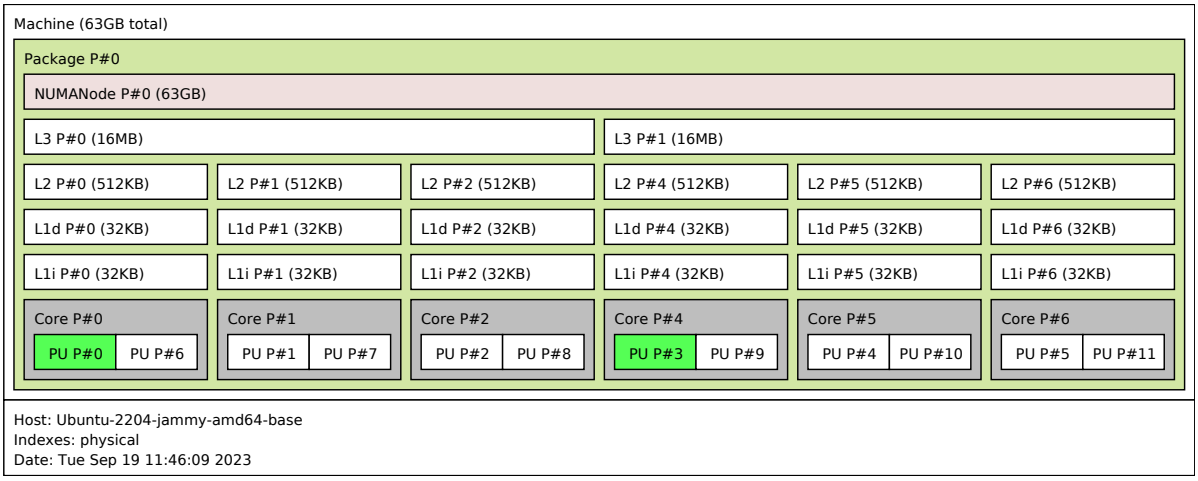


Figure 3.2: AMD Ryzen 5 3600 Core Topology exported by lstopo (hwloc package[9])

nice & taskset The first explored method for giving a process priority on a core was using `nice` for setting priority and `taskset` for pinning the process to a specific core. The command `nice`[10] sets the niceness (which can be seen as the priority for the scheduler) to -19, which is the highest priority. `taskset`[11] is used to set the CPU affinity of a process to a (set of) cores. Affinity implies that the schedular is likely to run the process on the specified core, but it is not guaranteed. We would chain the commands in a method such as: `nice -n19 -- taskset -c3 -- <benchmark command>`

[7] https://packages.ubuntu.com/jammy/cpufrequtils
[8] https://linux.die.net/man/1/cpufreq-set
[9] https://www.open-mpi.org/projects/hwloc/
[10] https://man7.org/linux/man-pages/man2/nice.2.html
[11] https://man7.org/linux/man-pages/man1/taskset.1.html

Both of these builtin commands require the Linux capability[12] CAP_SYS_NICE. Capabilities are a permission control system to allow programs to have certain elevated permissions without being executed as a different user, i.e. root. This capability would need to be given to the commands `nice` and `taskset` by the experiment control executable. Capabilities on processes are not inherently inheritable on Linux, thus simply giving the experiment executable the CAP_SYS_NICE capability would not grant this to any child process. We could solve this by adding the capability to the Inheritable and Ambient set of the nice and taskset commands (i.e. the files `/usr/bin/nice` and `/usr/bin/taskset` respectively) but this means altering built-in system commands and was deemed excessive.

Running the experiment control as `root` user is also not deemed feasible, since this would execute benchmarks as `root` as well and this imposes a security risk, hence we look at an alternative method: cpusets.

CPU sets

Cpusets are a method of pinning one or more processes to a set of cores, as well denying other processes access to those cores[13]. First we create a set containing only the experiment core using the command `cset set --set=BENCH -c 3 --cpu_exclusive`. The flag `--cpu_exclusive` ensures that the selected core 3 will be exclusively used for processes assigned to the set `BENCH`. Root permissions are needed to set up the cpuset, after which we can change the owner of the set by changing the owner and permissions of the cpuset files: `chmod --recursive g+rw /dev/cpusets/BENCH; chgrp --recursive $USER /dev/cpusets/BENCH;` During experiment execution instances of the benchmark framework van be started with the command `cset proc --exec BENCH -- <program to run>`, where the benchmark program is that as defined in section 3.4.

3.6. Measurement methods

The implementation of measuring the power consumption has undergone several iterations. The initial plan was to use `perf stat`. Due to issues with collecting the number of iterations for forty percent of the functions, the next solution was using `perf record`. This required a probe on a variable to capture the number of iterations. It was difficult to associate the probe name with the benchmark, which led to the final solution of extending Criterion with a custom measurement.

3.6.1. perf

`Perf` is a collection of ”Performance analysis tools for Linux” (“perf(1) - linux manual page”, 2022). The performance analysis tools use a kernel-subsystem that exposes performance counters on a hardware and software level. Using the RAPL API (subsection 2.1.4) perf provides a *events* related to energy consumption. Other built-in events include the number of cpu cycles, branch executions or misses.

perf probe

The `perf probe` command-set provides events by creating dynamic tracepoints that count the number of executions of a certain address in the target program. Such a tracepoint would be set on a recurring point within the code, such as a `jmp` or `cmp` assembly instruction related to looping the specific number of iterations. In code sample Listing 3.2, that would be at address 6037b where the loop happens based on the number of benchmark invocations.

Another option is to place a argument probe that captures the variable containing the number of iterations that is to be executed. An example of that would be capturing the value of register `%rcx` at address 60378 in code of the code sample Listing 3.2.

Both of these options have been explored and have been found to work. However, since the benchmark executable contains the code for multiple micro-benchmarks that have been defined in the same source file it has been found difficult to impossible to relate a probe-point to the respective benchmark. The benchmark function is given the generic name of `<criterion::bencher::Bencher<M>::iter>` in the compiled binary and as contains no static information about the specific benchmark that is called. Using a wildcard in the probe selector, and thus selecting the probes of all benchmarks corresponding to the same executable file was not a working due to the way `perf` groups its event counters: only the first probe in the list would trigger measurements and thus only one of the benchmarks would get

[12] See https://man7.org/linux/man-pages/man7/capabilities.7.html for information on capabilities.
[13] https://man7.org/linux/man-pages/man7/cpuset.7.html

```
0000000000060320 <criterion::bencher::Bencher<M>::iter>:
   60320:       41 56                   push   %r14
   60322:       53                      push   %rbx
   60323:       48 83 ec 18             sub    $0x18,%rsp
   [...]
   60370:       90                      nop
   60371:       ff ce                   dec    %esi
   60373:       75 fb                   jne    60370
   60375:       48 ff c2                inc    %rdx
   60378:       48 39 ca                cmp    %rcx,%rdx
   6037b:       75 e3                   jne    60360
   [...]
   603a3:       5b                      pop    %rbx
   603a4:       41 5e                   pop    %r14
   603a6:       c3                      ret
```

Listing 3.2: Excerpt of assembly displayed by `objdump`, generated by the rustc compiler and LLVM of the Criterion framework benchmark iteration function.

measured. In the paragraph below we describe the methods that were tried in attempting to collect energy data using `perf`.

perf stat and perf record
Initially we attempt to collect energy consumption over the execution of a benchmark executable using `perf stat`. This command counts the number of events that have occurred during the execution of the command. This is different from `record`, which saves samples at a default rate of 1000Hz so one can analyze later on the performance of the program under test. Both commands work in the

3.6.2. Criterion Measurement plugin

We implement an extension for the Criterion framework to measure energy consumption instead of the wall-time that is measured by default. The plugin is released and available on GitHub[14].

The crate criterion-rs supports custom Measurements[15] by providing a public trait: `criterion::measurement::Measurement` (see Section 2.2.1 about traits) that users can implement and provide to the framework using generics. The custom measurement is implemented by opening a file-descriptor to the pseudo-file `/dev/cpu/{cpu_id}/msr` as described in Section 2.2.3 and reading the energy value from it. The benchmarking framework Criterion calls the `Measurement::start()` method before running the micro-benchmark; at which point a file descriptor pointing to the MSR-file is opened and the Core Energy Status value is read, which is then stored. After the iterations have been completed the framework calls the method `Measurement::end()`, in which our implementation reads the Core Energy Status (CES) value again, together with bits 12-8 of the Energy Status Unit (ESU) register, which is used to calculate the energy consumption in Joules: $E = CES \cdot 0.5^{ESU}$. The Criterion framework collects these datapoints and stores them in a JSON file, together with the number of times the method-under-test has been executed. This file is located within the build artifacts, namely `target/release/criterion/<benchmark>/new/sample.json`, which is used when calculating the energy consumption per execution as detailed in section 3.9.

Fixing register overflows The Core Energy Status (CES) Model Specific Register (MSR) is a 64-bit register, of which only the first 32-bits are used to store the value. This means the register resets to 0 after a a value of $2^{32} - 1$ in bits, or after $2^{32} \cdot 0.5^{15} = 2^{17} = 131072$ Joules. Given the 65 Watts thermal design power[16] we can expect a single one of the six cores to draw about 11 Watts under full load. Using the overflow value we calculate that the register is expected to overflow roughly every

[14] https://github.com/RensHijdra/criterion-energy/tree/b8f0131a66a8862fc7ea93963278b155efd72c31
[15] https://bheisler.github.io/criterion.rs/book/user_guide/custom_measurements.html
[16] https://www.amd.com/en/product/8456

$131072J/11W \approx 11000s \approx 3.3$ hours. Over the course of the two-week runtime, such an overflow is bound to happen. When an overflow occurs during one sample period, we do a wrapping subtract and the answer is right anyways

To fix the values where such an overflow has occurred within the measurement period we check whether the value is greater than an unreasonable consumption for 100 milliseconds, e.g. $25W \cdot 100ms = 2.5J$. This value is then subtracted from $2^{32} - 1$ which is the actual consumption for the period.

3.7. Collecting code features

In order to relate the stability of a benchmark back to source code, statistics about the benchmarks need to be collected. We collect both language elements such as loops and if statements, as well as method calls to the Rust standard library. To do this in the most accurate way we collect dynamic coverage of the program.

To compile an executable for coverage collection we use the following command: `cargo +nightly-2023-04-15 bench --no-run --bench <benchmark_name> --message-format=json -Zbuild-std --target=x86_64-unknown-linux-gnu --features coverage`.

We also supply the following environment variables:

- RUSTFLAGS `"-Csymbol-mangling-version=v0 -g -Cinstrument-coverage -Zno-profiler-runtime"`
- CARGO_PROFILE_BENCH_DEBUG `true`
- CARGO_PROFILE_BENCH_LTO `no`
- CARGO_PROFILE_BENCH_OPT_LEVEL `0`

.

We need the feature `coverage` from our plugin to enable and disable the collection of coverage instead of reading the energy consumption at the start and end of each benchmark. We use `-Zbuild-std` and `--target` for compiling the standard library with coverage collection. We disable binary and link-time optimisations to ensure consistent and expected debug results (GNU Compiler Collection, 2024). We enable the new symbol mangler to enable consistent and decodeable debug symbols (Rust Language Team, n.d.).

3.7.1. LLVM Source-based Code Coverage

Rust supports collecting coverage of its compiled programs through the LLVM *Source-based Code Coverage* [17]. This method of coverage collection divides the AST into regions that are divided through diverging code branches. For each region an LLVM instrumentation counter instruction is inserted into the generated Mid-level Intermediate Representation (MIR) phase of the compiler. During compilation a Coverage Map is generated, which will be used to relate the counters back to the respective source code locations.

3.7.2. Standard Library Coverage

The Rust language is based upon three libraries: `std`, `core`, and `alloc`; which implement higher-level abstractions, low-level bindings, and pointer and memory management respectively. To build the standard library source with instrumentation counters we use the unstable feature flag `-Z build-std`. When building the standard library it is required to specify the target architecture; in this case that is `--target=x86_64-unknown-linux-gnu`.

Rust's profiler runtime The Rust language environment provides an implementation for this in the form of the profiler_builtins module, which is part of the language core. When the `-C instrument-coverage` configuration flag is passed to the `rustc` compiler, it will ensure this crate gets included as a dependency and adds the instrumentation pass to the compilation phases. This flag also enables the `-fprofile-instr-generate` flag to be passed to the LLVM compiler. The profiler_builtins module

[17] https://rustc-dev-guide.rust-lang.org/llvm-coverage-instrumentation.html

is marked as experimental and thus requires the use of a nightly version of rust, or a custom compiled compiler with profiling enabled. For this thesis we have attempted to compile a custom compiler and standard library. The configuration file needs to contain `profiler=true`, and the stage2[18] compiler is then compiled with the `"-C instrument-coverage` flag. This method will not work, however, since profiler_builtins is not a #[no_core] module it depends on the Core crate[19]. But, since the instrument-coverage flag is enabled, the compiler first tries to compile the profiler_builtins before the core crate, since it needs it to add instrumentation into the core crate.

Minicov profiler Runtime To solve this chicken-or-the-egg dependency problem we add a precompiled profiler runtime to our dependecies. Minicov provides a "modified version of the LLVM profiling runtime [...] from which all dependencies on libc have been removed" (d'Antras, 2024). This means it does not depend on the core or std crates and can thus be used to profile those them. Furthermore, this crate removes the need of compiling a custom compiler which simplifies the coverage collection process. When using a custom profiler we need to specify the unstable flag `-Z no-profiler-runtime` to tell the compiler not to use the default profiler runtime.

3.7.3. Source Code Feature Extraction

Collecting LLVM coverage data enables us to filter out unused parts of the Abstract Syntax Tree while parsing files for features. Using the coverage data we create two datasets for the dynamic features: one counting the occurrences in the code and a second summing the number of executions of each occurrence.

For this purpose we have a subcommand in the experiment program. This reads the LLVM profraw file and executes the `llvm-profdata show` command to generate the coverage data in JSON format that we read in our Rust program. The generated data contains mappings of source files and code blocks within. We use the `syn`[20] library to walk the Abstract Syntax Tree (AST) and keep count of the features the walk comes across. Since LLVM coverage data not only tracks whether a code block has been visited, but also how often, we can also keep track of the total count of a feature occurring. The coverage data contains multiple source files per benchmark, since any function that is called at any point is tracked, through user libraries and standard libraries. The features that we keep track of are detailed in Background subsection 2.1.3. The `syn` crate provides a framework for building an Abstract Syntax Tree visitor which we implemented with a feature counter. All elements that are listed in Table 2.1 have one or more associated `visit_*` function we implement and increment a counter. Since each benchmark has multiple sourcefiles reported by LLVM, we build an AST for each file and combine the counted features for each file per benchmark after visiting each AST. One such visitor function, `fn visit_expr_reference` is provided in Listing 3.3, counts the number of references and also whether a reference is mutable. We serialize the HashMap that keeps track of the counts and repeat this process for each benchmark.

We end up with a CSV that contains the benchmark name and the number of occurrences per feature for that benchmark.

3.7.4. Counting Instructions

We want to compare the RCIW to the number of instructions in a benchmark to determine whether there is a correlation to be found. A profiler such as Callgrind can determine the exact number of instructions being issued by simulating running the program. Callgrind supports the toggling of data collection from within the program using the `CALLGRIND_{START, STOP}_INSTRUMENTATION` and `CALLGRIND_TOGGLE_COLLECT` macros[21]. We use these to isolate the benchmark and ignore the Criterion framework overhead by utilising the `callgrind_partial` crate[22] that implements these macros in Rust. We create another drop-in replacement `Measurement` for the Criterion framework (subsection 3.6.2) and call the appropriate Callgrind macros in the `Measurement::start()` and `Measurement::stop()` methods. This implementation is placed behind Rust's conditional compilation feature gates (subsection 2.2.1) so that we can reuse the configuration for the original energy

[18] "the truly current compiler" - see Rust Compiler Bootstrapping https://rustc-dev-guide.rust-lang.org/building/bootstrapping.html
[19] Platform-agnostic, dependency free foundation of the Rust Standard Library. See: https://doc.rust-lang.org/core/
[20] https://crates.io/crates/syn
[21] https://gcc.gnu.org/onlinedocs/cpp/Macros.html
[22] https://crates.io/crates/partial-callgrind

```rust
    fn visit_expr_reference(&mut self, i: &'ast ExprReference) {
        // Determine whether the AST node is actually visited
        if !self.region.overlaps_span(&i.span()) {
            return;
        }

        // Count references
        self.count("reference");

        // Count if the reference is mutable
        if i.mutability.is_some() {
            self.count("reference_mutable");
        }

        // Recurse into any contained nodes with the default visitor implementation
        visit_expr_reference(self, i);
    }
```

Listing 3.3: Example AST visitor function visit_expr_reference which is called when a reference expression is parsed in the tree.

`Measurement` simply by adding the `-features callgrind` flag to the compilation command.

We add the functionality of collecting Callgrind information to a top-level subcommand of the project: `instructions`. When this command is run, all projects are compiled just like before only using the configuration meant for callgrind using the `--features criterion-energy/callgrind` flag. All benchmarks are run once through the callgrind profiler. An example of a full command is:

```
valgrind --tool=callgrind --fair-sched=yes --dump-instr=yes
 ↪  --callgrind-out-file=/dev/null --collect-atstart=no --instr-atstart=no
 ↪  <full_path>/chrono-b129a5080da883cf
 ↪  ^bench_get_local_time/bench_get_local_time$
```

The valgrind commands standard output will contain lines like the following:

```
==120215== Events    : Ir
==120215== Collected : 210237
```

The event `Ir` is the executed instruction count and the following line is the corresponding value; i.e. 210237. All results are collected in a CSV file which is saved for later use.

3.8. Baseline

We create our own project with benchmarks that execute the assembly instruction NOP many times, the code is visible in section B.3. The loop will be compiled to the something similar as the code in Listing 3.4:

```
60360:       be e8 03 00 00      mov    $0x3e8,%esi #0x3e8 == 1000
[...]
60370:       90                  nop
60371:       ff ce               dec    %esi
60373:       75 fb               jne    60370
```

Listing 3.4: Assembly excerpt showing a decrementing counter.

For the benchmarks of the projects under investigation, the instruction `NOP` at address 60370 would be replaced with the code-under-test; most often in-lined but occasionally behind a function call.

The size of the loop has been chosen in the range 10^2 up to 10^6, increasing with a factor 10. Using this maximum size we capture approximately $4.2\text{GHz} \cdot 100ms/10^6\text{instructions/execution} = 420$

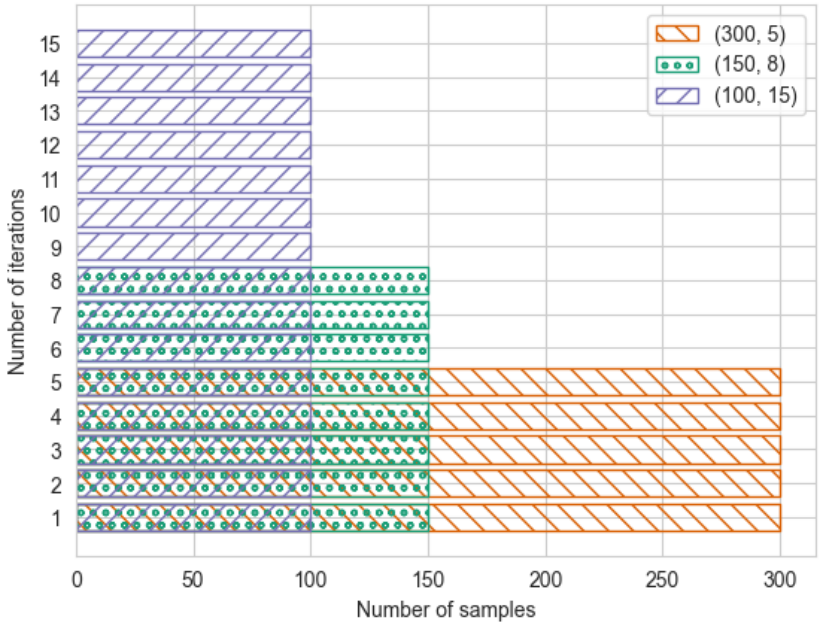


Figure 3.3: Three examples of picking combinations of number of iterations and samples. Successive iterations are stacked vertically and their samples are aligned.

benchmark executions in a single sample. These benchmarks are executed like any other benchmark has been in this project; the only difference is that they were executed at a different time than the main experiment.

3.9. Data processing

Research question 1.2 investigates the number of samples required to get a representative value of stability. To do this we want to calculate the RCIW for different number of samples to compare across and determine the best value.

3.9.1. Calculating energy per execution

Before we can answer our research questions we need to pre-process the data. Over the course of three weeks the experiment has collected more than 8.1 million data points by running 904 benchmarks thirty times; while taking three hundred samples during thirty seconds. These data points come in three parts: the benchmark it belongs to; the number of executions of the benchmark during the 100 millisecond sample period; and the energy consumed by the processor core during this time. We group the data by benchmark, and for each sample we calculate the energy consumed per execution:

$$\frac{\text{energy consumed in sample}}{\text{executions in sample}} = \text{energy per execution.}$$

3.9.2. Sample selection

After collecting the energy data for all projects we have 9000 samples per benchmark that are collected as sets of 300 samples over thirty iterations. Since these iterations are executed at distinct moments in time, we cannot simply concatenate them and take an increasing number of samples from the whole lot; environmental noise could have had a different effect on the data at different moments in time. Therefore we take an equal number of samples from each iteration when calculating the RCIW.

Figure 3.3 shows three such subsets of samples. On the vertical axis we see the number of iterations taken into account and the horizontal axis shows the number of samples per iteration used in the calculations. Subset `(150, 8)` represents a set of 1200 samples: taking the first 150 samples of the first eight iterations, that were taken at different moments Both the subsets `(300, 5)` and `(150, 10)` have a total of 1500 samples but reach that number using 300 and 150 samples per iteration and 5 and 10 iterations, respectively.

3.9.3. Calculating relative variability

Using (subsets of) these 9000 “energy per execution” data-points we can calculate measures of stability for each benchmark. We look at the Relative Median Absolute Deviation (RMAD) and Relative Confidence Interval Width (RCIW), which are robust measures of variability.

3.10. Random Forest Models

In the final research question we want to determine what source code features are important to keep in mind when tackling instability in their code. We create a Random Forest Regressor[23] that we initialise with a random state of 566784. We feed the features as found in subsection 3.7.3 into this regressor through a 30-fold Cross Validation step.

[23] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

4

Results

In this chapter we present the results obtained using our methodology. We answer the research questions laid out in the introduction. In the first section we dive into the theme: what does it mean to be stable. This includes calculating the stability of the tested projects and determining the number of samples required to get a stable result. In the second question group we explore possible reasons for instability.

Before answering any questions we take a quick look at the data we retrieved in general. Figure 4.1 shows Kernel Density Estimate of the Harrell-Davis median energy consumption over all 9000 collected samples and number of executed assembly instructions of each benchmark. A glance at the figure raises the suspicion of correlation between energy consumption and benchmark length (in number of instructions) and a Spearman Rank Correlation test (for $\alpha < 0.05$) confirms this: we retrieve a strong positive correlation of $\rho = 0.957$.

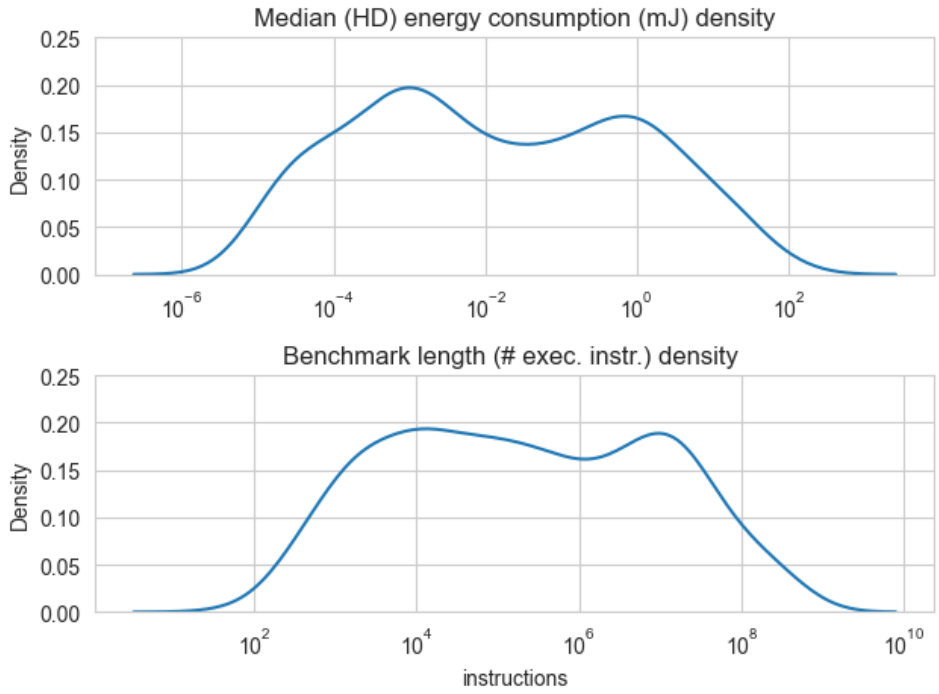


Figure 4.1: Distribution of energy consumption and number of instructions

Normality We perform a normality[1] test with significance at $p < 0.05$ on the datapoint grouped per benchmark. We find that most (861 of 868) benchmarks' measurements do not conform to a Normal distribution, therefore we we use robust "distribution-less" statistics in answering the Research Questions.

4.1. RQ 1.1 - Is software energy consumption generally stable?

In the experiment we have collected energy consumption data over thirty iterations. For this research question we only take into account the first sample of each iterations. These samples have been taken at random intervals over a three week period and thus simulate developers running the test at different moments. The interval between subsequent samples of the same benchmark lies between 0 and approximately 17 hours, with a theoretical mean of 8:40h. This means that the values could be representative to measuring the energy consumption by a developer during a few weeks.

We take these thirty first samples and calculate the Relative Dufference between the minimum and maximum per benchmark: $\frac{maximum-minimum}{minimum}$. We use this Relative Difference to be able to compare the spread between the minimum and maximum of consumption across different benchmarks without being sensitive to the magnitude of this consumption. Figure 4.3a shows a histogram of the calculated relative maximum differences with those above 50% left out and instead shown in Figure 4.3b. From this histogram we can see there is a wide spread of relative change between the minimum and maximum consumption of benchmarks. Two modes in this figure show that a majority (69%) of benchmarks exhibits a first-sample maximum difference of at most 10 percent, which is considered an upper limit on stability in some literature (Laaber et al., 2021); and a quarter of the benchmarks (26%) slightly above this up to 25%. The remaining five percent of benchmarks have a relative change of up to 500%.

In Figure 4.4 we plot the 45 benchmarks that have at least a twenty-five percent relative change. Each unit on the horizontal shows one iteration of the experiment, while the vertical axis shows the power consumption of a single benchmark execution in millijoules. We can see that there is one benchmark with an order of magnitude difference between the minimum and maximum energy consumption. Many benchmarks exhibit differences between samples that would not necessarily be ironed out by taking a single sample or the mean of two samples. To expand on this we see in Figure 4.2 where all iterations have been stiched together horizontally that the outliers are not merely a first-sample effect but the difference in energy consumption stays throughout an iteration.

When used for regression detection, data collected from a single iteration would not be enough to eliminate false positives. This indicates that, in order to accurately detect regressions, it would make sense to collect more than one iteration of energy consumption.

While some benchmarks do show little variation (less than 10%) between the minimum and maximum of the first samples, energy consumption can vary significantly for other benchmarks where the difference between two samples can be up to 5 times larger.

To conclude, we do not find the energy consumption of benchmarks of our investigated projects to not be stable over time, despite our efforts to reduce environmental influences.

4.2. RQ 1.2 - How long should you measure for a stable result?

From the previous question we determine that we need to collect more samples over time to determine an accurate median energy consumption. In this research question we find a guideline for users in setting up their benchmarks: what is an appropriate amount of time and samples to measure for.

We calculate our metrics over the range iterations [1, 30] and samples [25, 300, step=25]. This means we take into account all measured iterations and a selection of the possible combinations of samples. By using a step-size of 25 for the samples we aim to find a middle ground between resolution and calculation time for the RCIW in finding an answer to the research question. For each of these values we take up to that number of iterations and samples into account; this is visualised in Figure 3.3.

In Figure 4.5 we see that comparing over an increasing number of samples, for all values of iterations, there is a monotonic decrease in mean RCIW. When comparing for distinct values of samples, an increasing range of iterations we see that the trend is not strictly monotonic. Therefore we plot for a selection of 6 iterations values ([5, 10, 15, 20, 25, 30]), the RCIW for each tested benchmark over

[1] https://docs.scipy.org/doc/scipy-1.12.0/reference/generated/scipy.stats.normaltest.html (D'Agostino & Pearson, 1973)

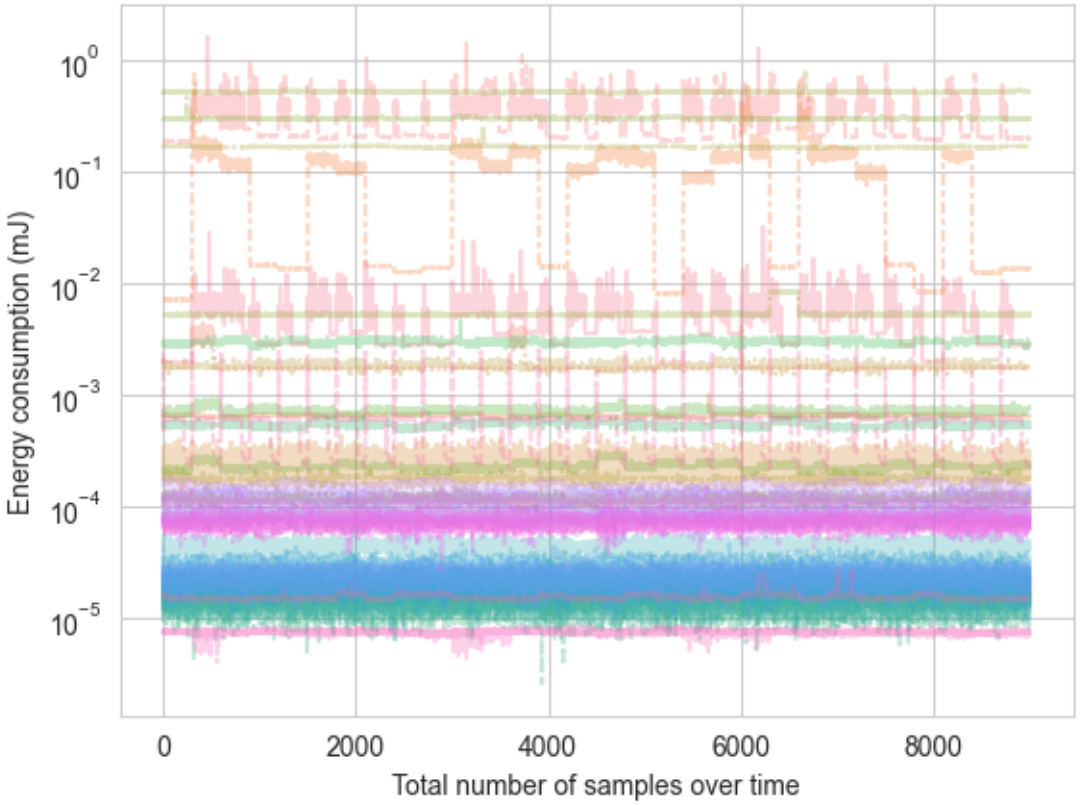
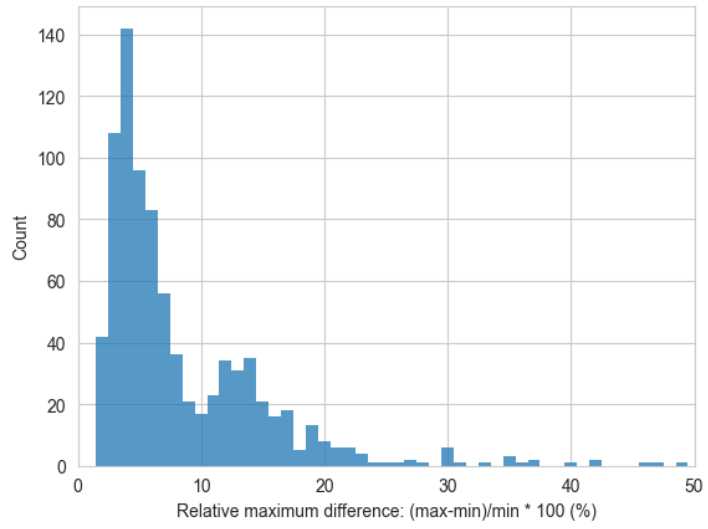


Figure 4.2: Plot of the energy consumption per sample for benchmarks that have a Relative Maximum Difference of at least 50%. All iterations have been stiched together on the horizontal axis.



(a) Histogram

61.45	94.72	124.46
78.44	94.88	132.35
80.43	99.34	149.21
81.86	113.52	178.49
87.39	120.06	187.06
88.90	120.47	293.10
94.22	123.78	5204.74

(b) 50%+ values

Figure 4.3: Occurrence counts of the relative maximum difference between the first samples of iterations.

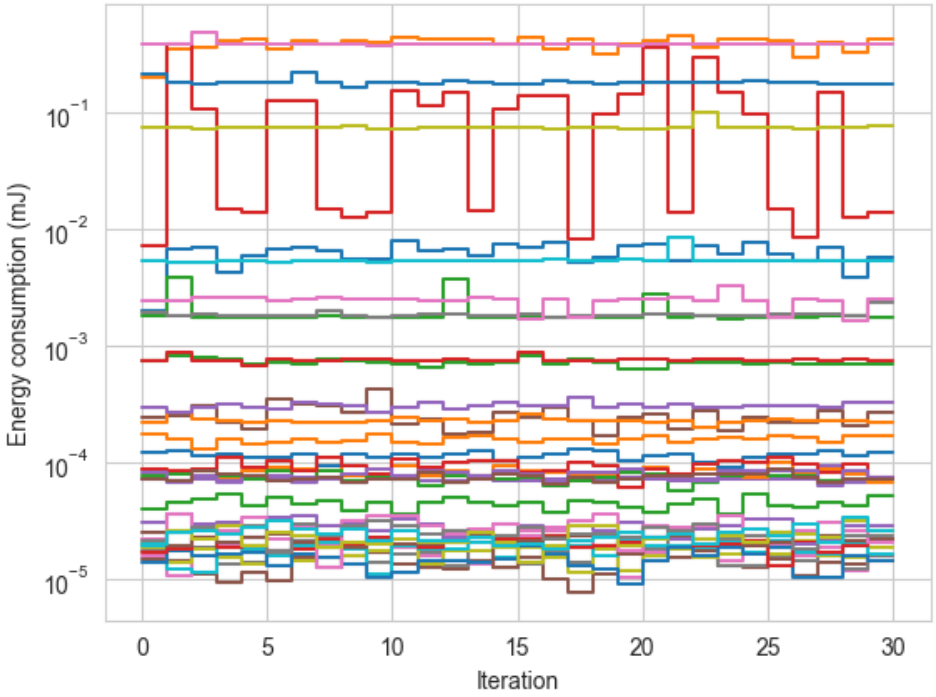


Figure 4.4: Energy consumption: first samples of 30 iterations for benchmarks with a at least a 30% difference between max. and min.

the range of samples [25, 300]. This can be found in Figure 4.8 where For brevity we pick these six iterations; all plots can be found in the Appendix: Figure C.1.

We do not consider any iterations under 4 to be relevant which is supported by Laaber et al. (2021). When using less than five iterations, the opportunity arises for the data to get multiple modes that lie further apart, resulting in a wide confidence interval. We can see this in Figure 4.5 where the median of all RCIW generally tends to decrease as the number of iterations increases. This is not true for the number of iterations below five, where I=2 and I=3 have a higher median RCIW than I=1, and I=1 and I=4 lie approximately at the same values.

In Figure 4.8 we can see that most benchmarks follow the trend where a larger number of samples results in a smaller RCIW. Since the Relative Confidence Interval Width is based on the variance we expect this behaviour: variance generally inversely proportional to the number of samples, which is also the case for the confidence interval by Maritz and Jarrett (1978).

The few benchmarks that do not follow this trend can be characterised, which is done in section 4.6.

We can see in Figure 4.5 that the median of all RCIW tends to decrease as the number of iterations increases. This is not true for the number of iterations below five, where I=2 and I=3 have a higher median RCIW than I=1, and I=1 and I=4 lie approximately at the same values.

We can take a look at the trend of the medians with regards to both the number of iterations and the number of samples taken into account (Figure 4.5). For both lines we see the values start with a steeper decrease moving to a shallower one. For Figure 4.5 (a), the decrease along the horizontal axis (number of samples) is monotonic, but for (b) it is not (number of iterations). This occurs because adding an extra iterations into the dataset since this new iteration might have a different mean than the ones added earlier.

We see the same median trends of Figure 4.5 in Figure 4.6, where we plot 1, 2 and 3 iterations as well as 5, 15 and 30 iterations, with additional dispersion data in the form of boxplots. At the top of each boxplot we have indicated its number of outliers, this will be used later in this research question.

We plot the first three iterations to show how a low number of iterations influences the stability, i.e. a larger spread. The second row of iterations shows the trend when increasing the number of iterations more significantly. We can see from the outliers and the approximate shape of the plots that the distribution does not greatly change across iterations, apart from some sporadically acting

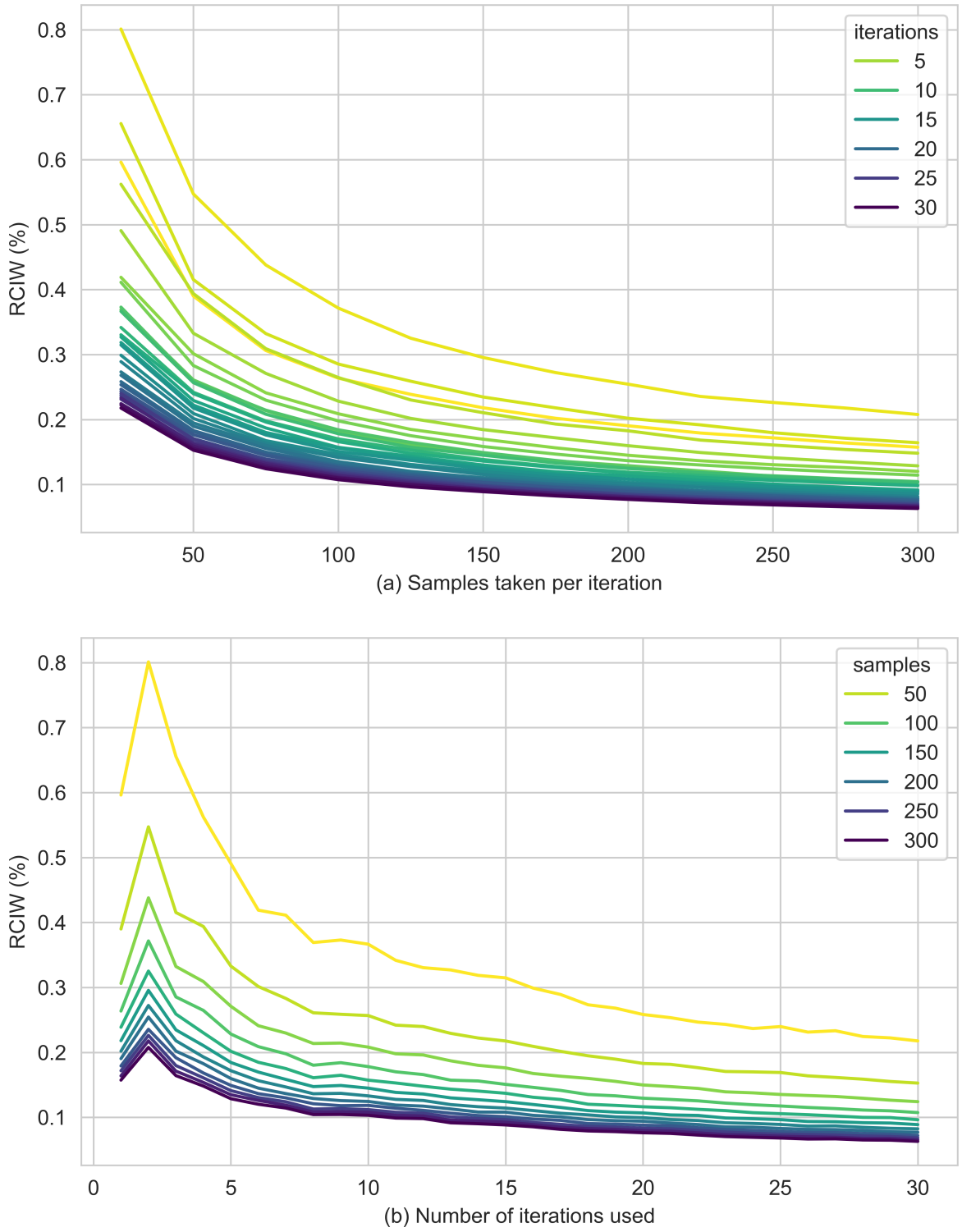


Figure 4.5: Median RCIW over all benchmarks. Plotted for (a) a selection of iterations across all sample values and for (b) a selection of sample values for all values of iterations.

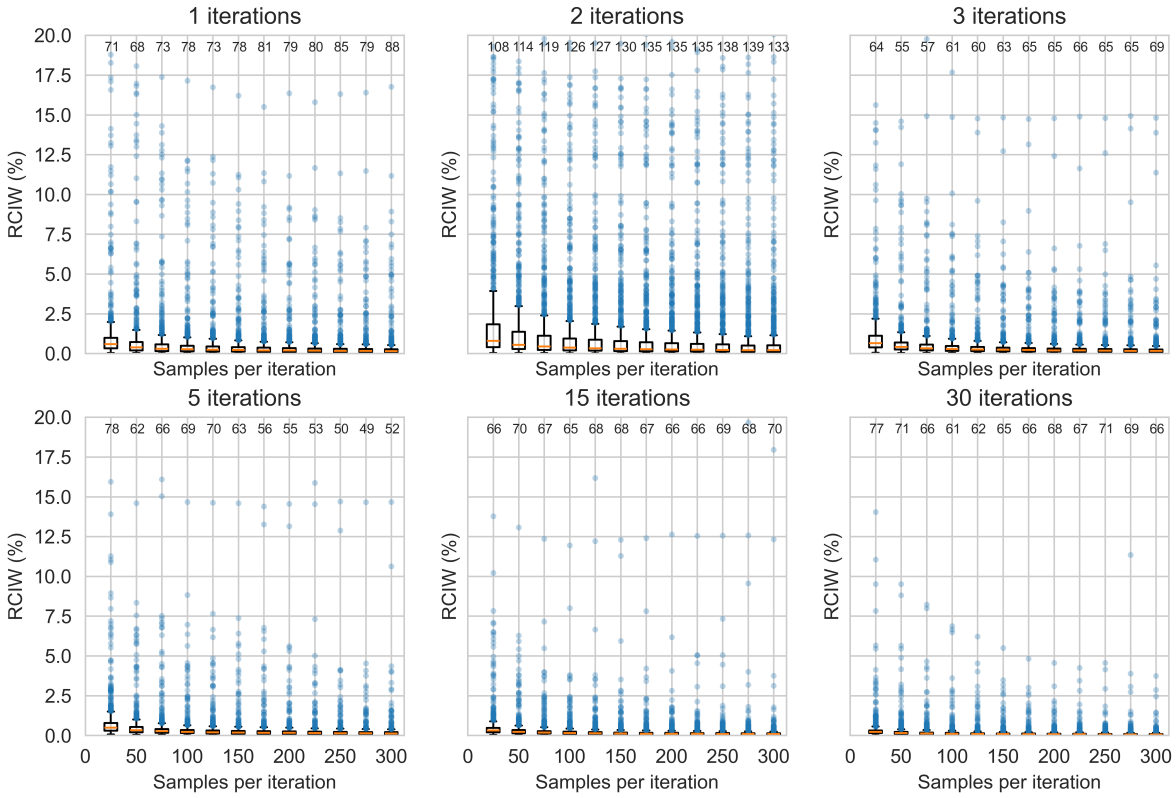


Figure 4.6: Boxplots of RCIW per benchmark, for a selection of iterations and all samples.

benchmarks. This is corroborated by the medians we saw in Figure 4.5 and by the three-dimensional plot in Figure 4.7. This last plot shows the relation of RCIW to the total number of samples as indicated by the colourbar from which we learn that picking either number of samples per iteration or number iterations are viable options in reducing the confidence interval width.

In statistics outliers are defined as lying outside of 1.5 times the Inter Quartile Range (1.5IRQ). We adopt this method in finding an appropriate number of steps to reach a valid answer. We calculate the IRQ and outlier limit on the RCIW values per number of samples and iterations, which are show in in Figure 4.10a. In Figure 4.10b we can see that the number of outliers (defined by being outside 1.5IRQ) is in the range [52, 78], except for iterations 1 and 2.

The colours for the outlier limit value in Figure 4.10a has been configured to be divergent around the one percent mark, which we have chosen as the RCIW threshold for stability. Additional testing has shown that, while the limit per iteration appears to be 80 ”unstable” benchmarks, there are up to 120 unique ”unstable” benchmark per iteration value when checking all sample values. This corresponds with the findings of Figure 4.8 where we see the per-benchmark RCIW converging but not acting monotonically. That is: there is some amount of benchmarks that is converging but fluctuating around RCIW 1% and their boolean ”classification of stability” might switch multiple times when calculating RCIW over different numbers of samples.

The points where 1.5IRQ + Q3 reaches below 1% can be considered a pareto-front, with no true best value. This pareto-front is approximated by the total number of samples as can be seen in Figure 4.9.

4.3. RQ 1.3 - What part of instability can be attributed to noise?

To figure out what part of the stability could be attributed to the different benchmarks and not to noise that happened during the sampling, we run benchmarks on code that runs empty loops. This creates a predictable pattern for the CPU with a stable load, that should inform us of the measurement stability when no actual code is run; this is further detailed in section 3.8.

In Figure 4.11 we can see the stability of the baselines over a the same range of values as in the previous research question: iterations [1,30] and samples [25, 300, step=25] . At five iterations we see

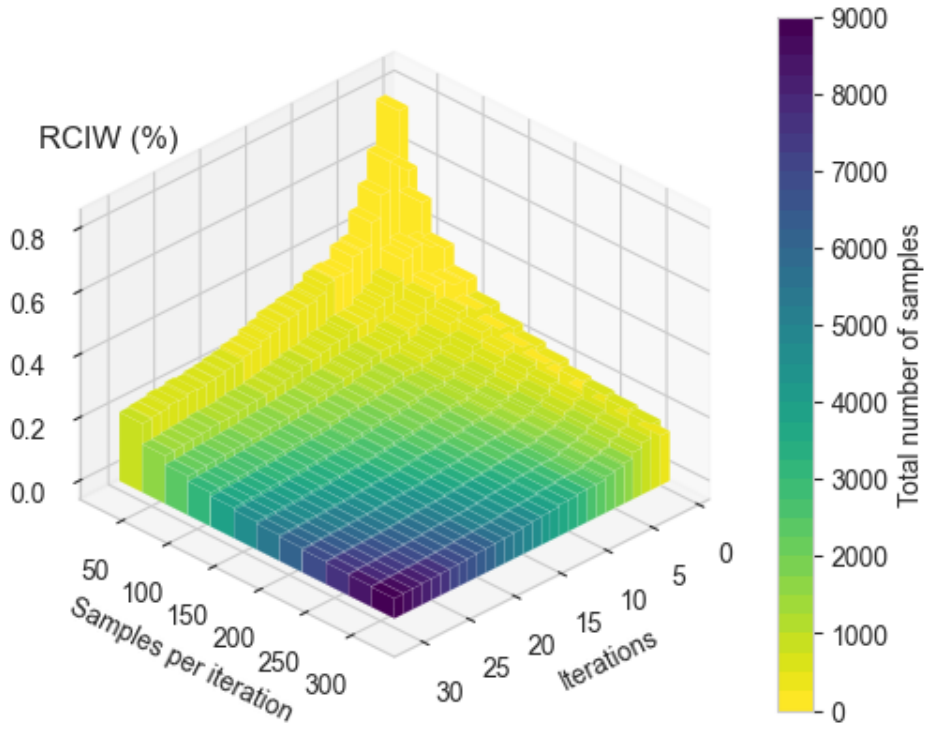


Figure 4.7: Median energy consumption plotted as the vertical axis across the two horizontal axes number of iterations and number of samples.

the RCIW for two of the benchmarks (100 and 1,000,000 iterations) start relatively high with regards to the other three benchmarks and approaching the other benchmarks when coming to 300 samples. For the other five samples we see all benchmarks exhibiting a shallow downward slope, always below 0.2% and often below 0.1% RCIW. In comparison: for the tested benchmarks the median only reaches this value for a high number of samples as we see in Figure 4.8.

The baseline RCIW corresponds approximately to the minimum RCIW values attained by the experiment benchmarks, below 0.05%. The median RCIW for the experiments starts at a maximum of 0.8% for very low number of total samples and approaches 0.5% for the maximum number of samples in this thesis, 9000.

From this we can conclude that the measurement and environmental noise in energy consumption follows the same trend in stability as the energy consumption. Building on RQ1.1 – values below 1% RCIW can be considered stable – together with the measurement noise being below this threshold at all number of samples we conclude noise is not a contributing factor in the instability of measurements of benchmarks.

4.4. RQ 2.1 - Does powersaving mode affect energy stability?

The userspace governor is set at a fixed frequency of 4.2GHz during the entire experiment. The powersave governor differs in two main ways: the frequency is not fixed and differs over time which results in a longer time-span and less executions per iteration; and the frequency is likely to change based on demand. Both of these differences could produce an decrease in stability.

We plot the RCIW for both the userspace (at fixed 4.2GHz) and powersave governor. Figure 4.12 shows the RCIW for the five different benchmark sizes for both frequency governors.

We perform a Wilcoxon signed-rank test to determine the difference between the two governors. We perform the tests at a 95% confidence level, i.e. reject the null hypothesis if the p value is below 0.05.

Firstly to determine whether the data is coming from the same distributions, we perform a two-sided test with the null-hypothesis $H_0 : \mathrm{RCIW}_{Userspace} = \mathrm{RCIW}_{Powersave}$ and an alternative hypothesis of $H_a : \mathrm{RCIW}_{\mathrm{Powersave}} \neq \mathrm{RCIW}_{\mathrm{Userspace}}$. We retrieve a $p = 2.09 \cdot 10^{-57}$ and reject the null-hypothesis,

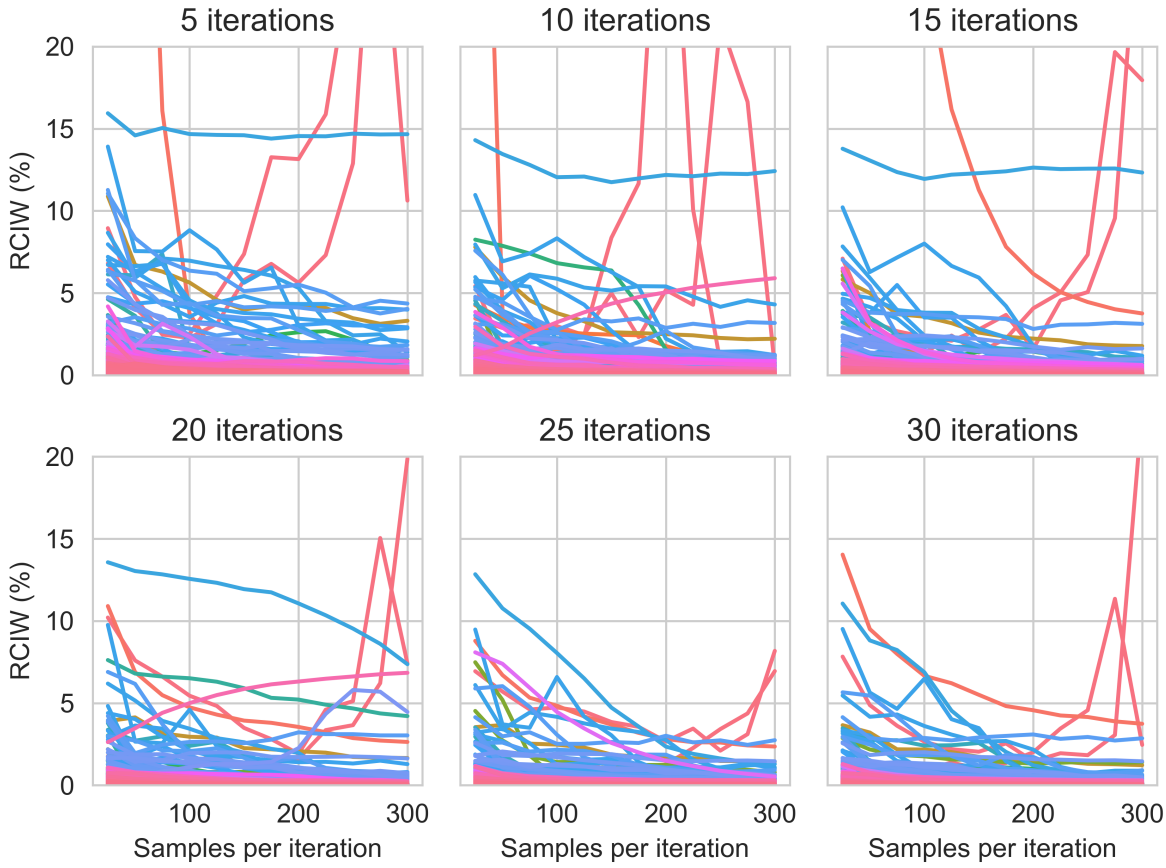


Figure 4.8: All benchmarks plotted across a variable number of samples. A selection of number of iterations has been chosen.

and conclude the data is not from the same distribution; this suggests there is a difference between the stabilities achieved under the two frequency governors.

To test whether our intuition is right, we perform another Wilcoxon test, with Null-hypothesis H_0 : $\mathrm{RCIW}_{Userspace} = \mathrm{RCIW}_{Powersave}$ and alternative hypothesis H_a : $\mathrm{RCIW}_{Userspace} < \mathrm{RCIW}_{Powersave}$. This test results in $p = 1.05 \cdot 10^{-57}$, thus we conclude that the powersave frequency governor does increase the instability of energy benchmarks.

We perform a $\mathrm{Vargha} - \mathrm{Delaney} A_{12}$ (Vargha & Delaney, 2000) test to determine the effect size of the difference between powersave and userspace governors. The calculated effect size is *small*. This means there is small decrease in stability when using the powersave governor over the fixed frequency userspace governor; and thus we recommend the latter when doing energy performance benchmarking for more accurate results.

4.5. RQ 2.2 - Does the number of instructions correlate with stability?

Since our experiment uses fixed periods for measurements, the number of executions of different benchmark functions differs. A longer codepath results in less executions per sample; when the calculated mean energy consumption is based on less executions a variation in the number of executions can be a cause of perceived instability itself. If this is a large influence we would expect a negative correlation between number of instructions and stability. The method of determining mean energy consumption and its possible effects are further detailed in subsection 5.7.3.

We want to find out whether there is a relation between the execution length of a benchmark and its stability. We use the data collected using Callgrind (subsection 3.7.4) and plot the Kernel Density Estimate of the number of instructions in Figure 4.13. In this graph we see the majority of benchmarks fall in the 10^3 to 10^9 instruction range. From the raw data we determine the extremes to be 230 and

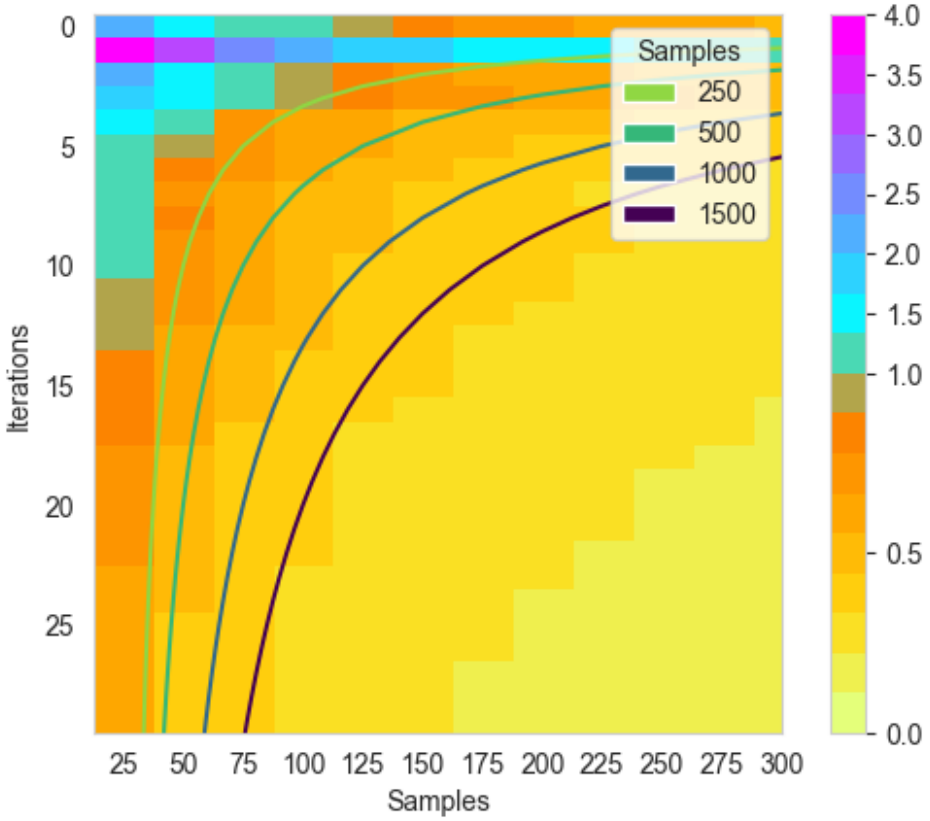
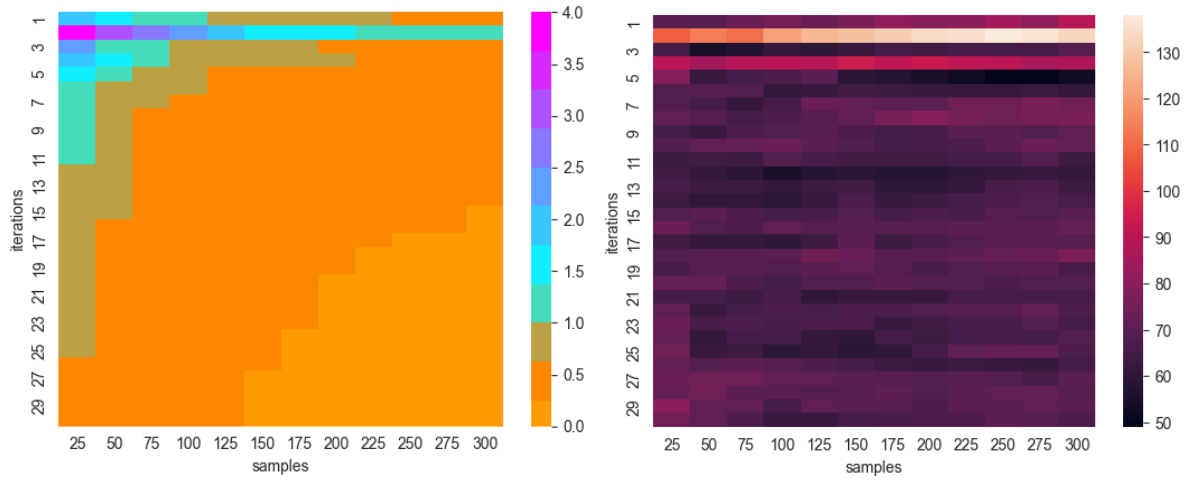


Figure 4.9: Calculated outlier limit Q3 + 1.5IRQ with total number of sample contours overlaid.



(a) Outlier threshold: Q3 + 1.5 · IRQ (RCIW %)

(b) Number of outliers

Figure 4.10: Outlier classification of benchmarks w.r.t RCIW per sample/iteration combination

$1.4 \cdot 10^{10}$, with a median of $3.0 \cdot 10^{6}$.

In the Introduction to this chapter 4 we have seen a strong but not perfect correlation between the energy consumption and benchmark length.

To test for a relation between the number of executed assembly instructions and RCIW, we calculate the Spearman Rank-Order Correlation coefficient ρ for all combinations of experiment iterations and samples with significance $\alpha = 0.05$. Since the Spearman test investigates the monoticity of data (de Winter et al., 2016), so we plot the RCIW against the number of instructions on a log-log plot in Figure 4.14 to investigate this for our data. For each subplot we have chosen a single value for the number of iterations, the same way as in the previous Research Questions. Per subplot all values of samples (i.e. 25 through 300, steps of 25) are plotted for the specified iteration. With the exception of the outliers at approximately 10^3 instructions the plots follow a generally monotonic trend and thus we can continue with determining Spearman's ρ.

The results are in the range $[-0.288; -0.151]$ with a median of -0.232, and can be found in Figure 4.15. This heatmap shows the spearman coeficient statistic for these calculated combinations on the range [-1, 1]; the possible range of the Spearman test.

All results achieve a $p < 0.05$ and we can reject the Null Hypthesis and determine that stability and instructions are not independent. From the magnitude of the Spearman ρ we determine there is a weak negative correlation between the stability and number of executed instructions; that is to say that when the length of a benchmark increases it is more likely to produce stable results in this experiment.

4.6. RQ 2.3 - What code features could make software unstable?

For a developer it could be useful to increase the stability of their benchmark in order to be more sure about the average consumption as well as making the measurement more accurate in regression detection. In research question 1.2 we have used the boxplot $1.5 \cdot IRQ$ rule to determine outliers and set a guideline for a number of samples. To determine the reasons for instability we need a dataset of unstable benchmarks. First of all, we take a look at the set of benchmarks that falls above the $1.5 \cdot IRQ + Q_3$ value for every iteration-sample combination in our experiments. This leaves us with 19 benchmarks that we will investigate, both manually and using the source code features collected as described in section 3.7. When taking benchmarks that classify as an outlier at some iteration-value combination, we note 181 benchmarks. This difference indicates that our method is not perfect, and that benchmarks can reclassify as either stable or unstable as the number of samples changes.

4.6.1. Manual investigation

For the nineteen benchmarks that display constant high RCIW we inspect the source code of the setup and code-under-test. The projects and names of these benchmarks are listed in Table 4.1.

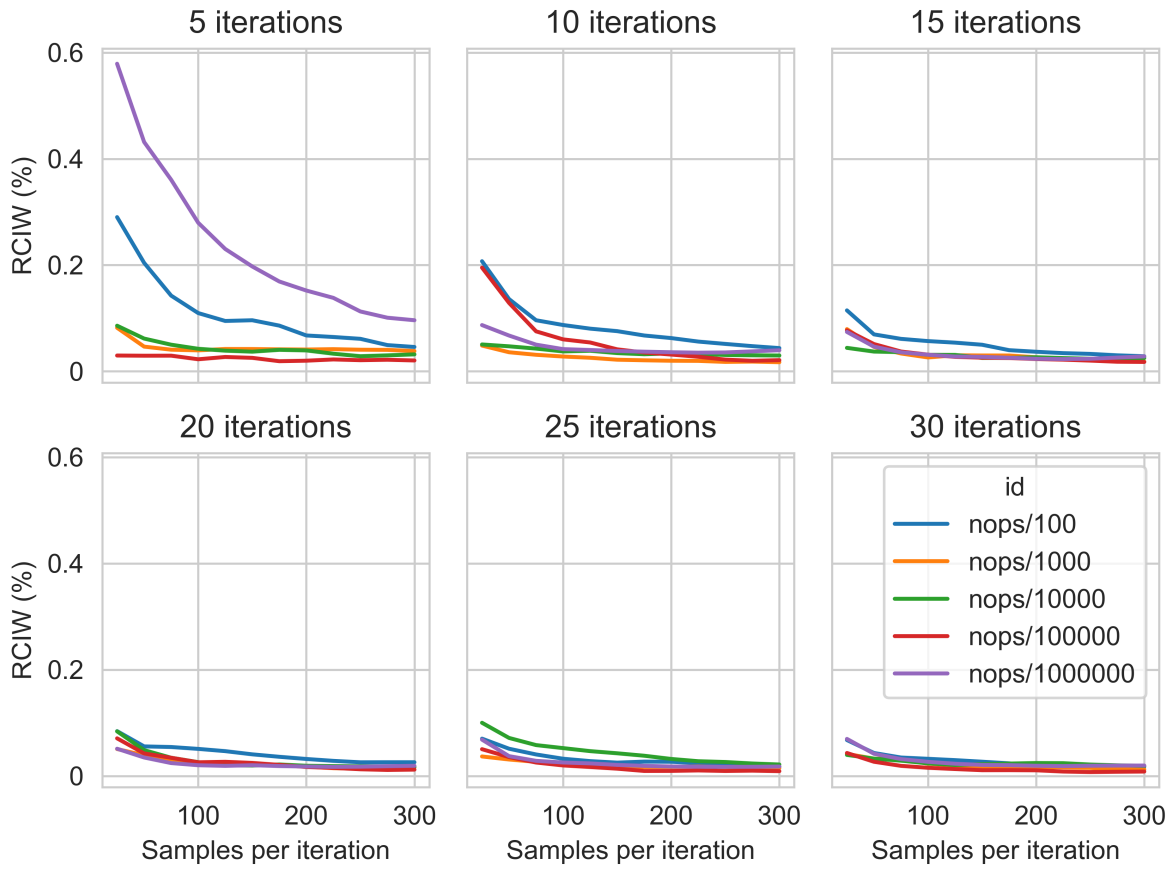


Figure 4.11: RCIW of the baseline benchmarks over values of iterations and samples.

The first thing that stands out is the project `aHash`, which is responsible for 17 out of 19 outlier benchmarks. `aHash` is a library that provides alternative hashing algorithms solely for in-memory HashMaps, and tests these hashing functions against other libraries in their benchmarks. When inspecting these benchmarks we find one thing in common: the setup of the benchmark makes use of randomisation. The unsigned integers that are to be hashed are generated by the `rand`[2] library: `rand::rng().gen::<u8>()`; where `u8` represents any unsigned integer between `u8` and `u128`. This random number generator is not seeded, i.e. the random numbers it generates are truly pseudo-random and not the same numbers every iteration as would be the case when a ”seed” is supplied to the generator. Furthermore, `aHash` makes use of the `iter_batched` function of the Criterion framework; with the setup of each batch operating on a different random number and thus different initialisation per sample occur.

The second project that has an outlier benchmark is rust-base64, which provides utilities for handling Base64 encoding and decoding. The encoding method that is benchmarked in this outlier is one that is, similarly to aHash, initialised with random data. Before iterating over the encoding function, the setup code creates a vector of unsigned 8-bit values (u8) and fills this with ”weak” random data ”to not be completely friendly to the branch predictor” (**rust_base64**). Once again similarly to aHash this random initialisation shows differences in energy consumption between iterations.

The final project is rust-prometheus, an interface for the metrics interface software Prometheus. The offending code is the threaded utilisation of a histogram counting implementation. Threads, as opposed to processes, always operate on the same core as their parent process; the four threads this benchmark spawns do not act truly concurrently but share the available processing power. Where all other benchmarks act in a single thread and can complete their task in an uninterrupted fashion this is not the case for this benchmark where the context switching comes into play. The CPU scheduler

[2]https://github.com/rust-random/rand

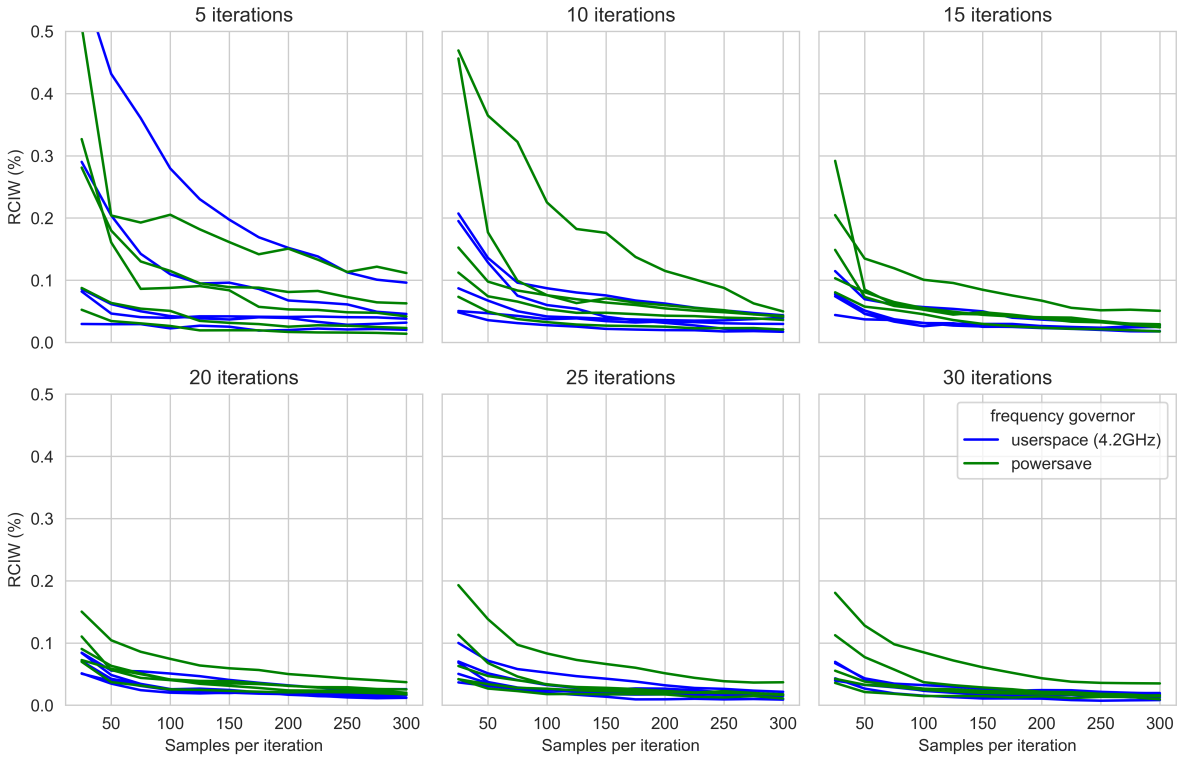


Figure 4.12: RCIW for several iteration sample combinations for the userspace and powersave governor.

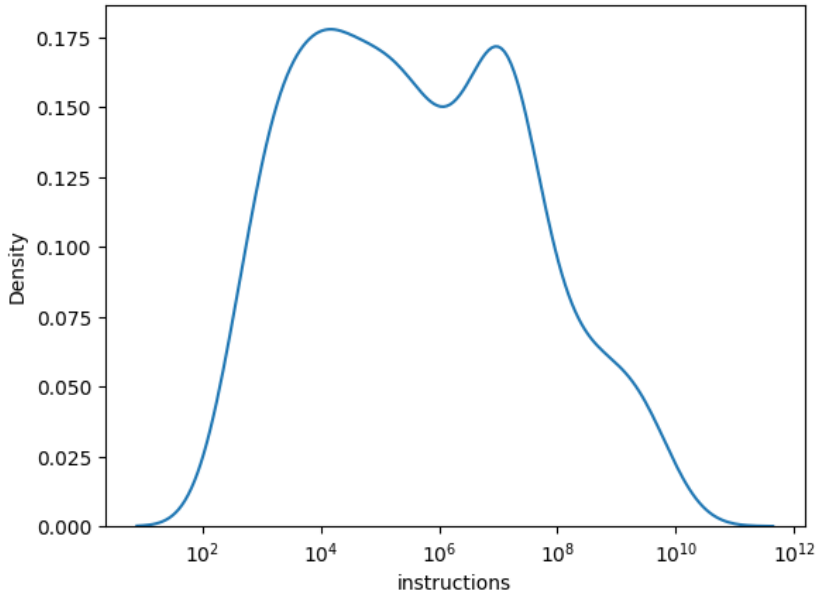


Figure 4.13: Kernel Density Estimate plot of the number of instructions in all benchmarks

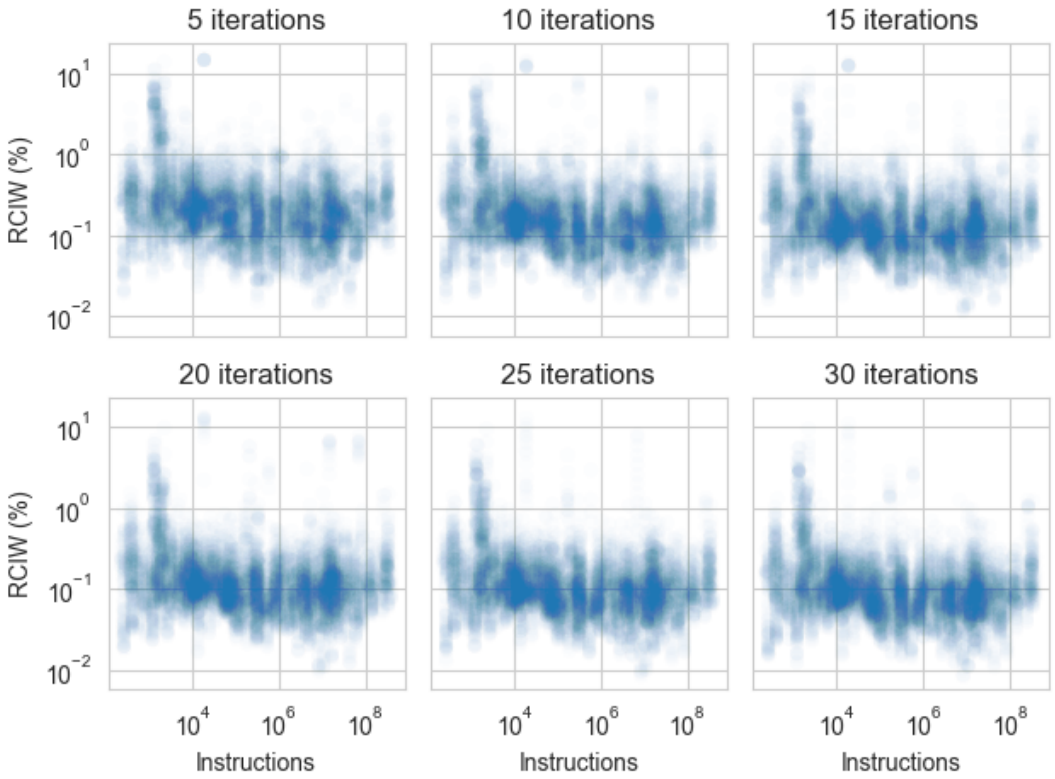


Figure 4.14: Number of instructions vs RCIW on a log-log plot, for six values of iterations, all samples.

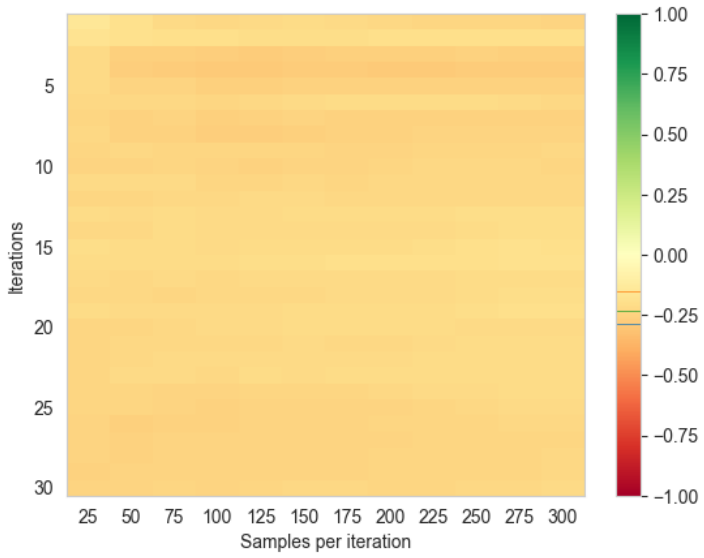


Figure 4.15: Spearman □correlation coefficient between number of assembly instructions and RCIW for values of iterations and samples. ([min -0.288; med: -0.232; max: -0.151], highlighted in colorbar)

Project	Group	Benchmark	Project	Group	Benchmark
ahash	fallbackhash	u8	ahash	fx	u64
		u16			u128
		u32		sea	u16
		u64			u64
		u128			u128
	fnv	u8		sip	u16
		u16			u64
		u32			u128
	fx	u8	rust-base64	encode_small_input	encode_string_stream/100
		u32	rust-prometheus	concurrent	concurrent_observe_and_collect

Table 4.1: Benchmarks that are outlier at any iteration-sample combination.

determines when a thread is allowed to run; when the current thread is pre-empted and the following resumes, a context switch occurs. Context switches can take from microseconds up to milliseconds depending on the amount of data that needs to be switched and are also not comepletely stable (Li et al., 2007), so it stands to reason these context switches have an adverse effect on the stability of the entire benchmark.

4.6.2. Feature importance

We use the collected source code features in combination with the calculated RCIW values to determine the influence of the features on the stability of the code. We do this by creating a Random Forest Regression on which we apply a 30-fold cross-validation. We achieve a best coefficient of determination of only $R^2 = 53\%$, but we use the feature importances as determined by this regressor, the results of which are displayed in Figure 4.16. The labels of the features classes have been explained in subsection 2.1.3. Features with a share of less than one percent have been grouped under the ”other” label. The features with most correlation to RCIW are language features `references` (memory) and `nested loops` (control flow), and standard library calls to `core::slice` and `core::mem` which are both modules that mainly handle memory manipulation. Together with the number of instructions, which has been handled in Research Question 1.2, these features make up half of the predicted influence on the RCIW. Looking at the rest of the features we see many that fall in the Control Flow and Data or Memory domains. The final domains that are present are Environment through `std::env` and `core::ffi`; and Input/Output through the filesystem calls of `std::fs`. The likely main features that make benchmarks unstable are memory accesses and unpredictable control flows.

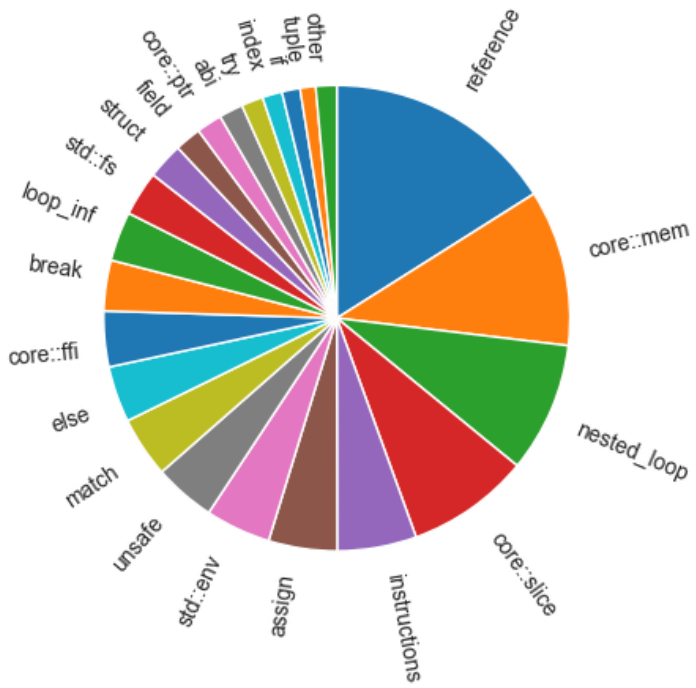


Figure 4.16: Feature importances as determined by Random Forest regressor

5 Discussion

In this chapter we reflect on the research questions and the results we achieved for them. We analyse and give context to the results obtained through our experiments. We look at the implications of our results and the way they how they further the field of sustainable software engineering. We also discuss the implications of the results and compare them against relevant literature.

Furthermore, we discuss the possible underlying reasons and influences that could have led to the observed results. Moreover review the threats and limitations of our study and their possible implications for the validity. We provide suggestions for future work, and

5.1. RQ 1.1 – Is energy consumption stable between two random independent measurements?

Our first research question seeks to answer the question ”Is energy consumption stable between two random independent measurements?” The data shows that this is not necessarily the case: the worst case difference between two samples that have been taken at different moments in time has been between zero and up to 5000% in our experiments. The values in this research question are the largest differences measured in the first samples of each iteration run; they are however, not the first execution of the benchmark in those iteration runs since each measurement period is preceded by a warm-up period of five seconds. The argument that it is not just a matter of first-sample instability is further supported by Figure 4.2 which shows that the average energy consumption differs between iterations as well. Research questions 1.3 and beyond investigate the possible reasons for this deviation in energy consumptions.

The Relative Maximum Deviation as used in this question is, however, not a good metric for measuring stability since adding more samples can only make the value higher. Because of this we introduce a new metric in the following sections: the Relative Confidence Interval Width. While there are more metrics available to measure stability, such as Relative Median Absolute Deviation we opted for the Maritz Jarrel version of RCIW because of its multi-modal capabilities and its scaling properties when changing the number of samples. We invite researchers to suggest and investigate different metrics in their research to energy stability if they are deemed better predictors or more effective.

5.2. RQ 1.2 – How long should you measure for a stable result?

How long should you measure for a stable result? – or in other words: How many samples should one take for their energy measurements.

First of all, there is no agreed upon definition of stability. In this paper we focus on developers that want to improve the energy consumption of their program and by extension know the magnitude of that consumption. For this question we plot our dependent metric, RCIW, against our two independent variables, iterations and number of samples per iteration, in several ways. The first takeaway from all plots in section 4.2 that on average the RCIW decreases with an increasing number of samples.

There are counter-examples to this: Figure 4.5 shows an increase of RCIW when taking two iterations into account instead of one; and Figure 4.8 shows individual benchmarks with respect to iterations

and samples per iteration. The first counter-example follows from RQ 1.1; since there could be large differences in the median consumption between iterations adding the second iteration could shift the median to somewhere in-between the medians of the two iterations. When the median lies in the middle, all samples are further away than when the median lies withing the samples; hence the higher spread and thus higher RCIW. Further inclusions of iterations to the dataset since they either lie closer to the median or move the median closer to either of the existing iterations' medians, which does not cause a higher spread. The second counter-example, the irregular an non-monotonic lines of Figure 4.8 show that not all benchmarks increase the stability with an increasing number of samples. These samples are classified as unstable outliers by the guidelines determined in this research question. They are simply unstable because of the way they are programmed as was found in *RQ 2.4 – What makes benchmarks unstable?*

Apart from no definite measure for stability, there is also no predetermined threshold for stability. What can be classified as stable differs per developer or per use-case, as well as per metric one uses. Laaber et al. (2021) for example, suggests thresholds for an RCIW metric between 1 and 15% based on other literature. In RQ 1.2, we have found that the median RCIW and along with it many individual benchmarks reach below the one percent mark.

Furthermore, in this research question we determine that 500 samples could be a reasonable limit to set for developers when benchmarking for energy consumption. Here arises another trade-off: the total of 500 samples can be composed of Samples per iteration $*$ Iterations in many variations. Figure 4.5 and Figure 4.10a show respectively that venturing below 4 iterations might not give a significant change relative to just one iteration; or going below 50 samples might reintroduce the unwanted instability into the results therefore we recommend picking values above the aforementioned. While it seems that the choice is arbitrary with regard to the duration of the benchmark, this is not true because of the warm-up period that happens before each iteration. A sample as defined in this research takes 100 milliseconds, thus 500 samples will take 50 seconds to complete. Add to this five times the number of iterations and one needs between 65 and 100 per benchmark. Whether these durations are acceptable is a decision that needs to be made by the performing developer.

These findings are limited by the facts that they come from a limited set of benchmarks, written in the Rust language with a single version of Rust and measured on one machine. These benchmarks have been executed under strictly defined circumstances:

- 5 second warm-up period
- Benchmarks running on a dedicated core
- Other cores on the machine disabled
- 100 millisecond sample duration
- Fixed CPU clock frequency
- Iterations spread over three weeks

Future work needs to be done to find whether the findings of this research extend to other benchmarks, other programming languages, other Rust versions, other hardware platforms and running under less stringent circumstances. Another interesting question could be whether the linear sampling mode as originally suggested by the Criterion framework also shares these results, or perhaps even better ones.

While we suggest 500 samples as a reasonable number to determine stability, and a threshold of one percent that classifies this stability, one should keep in mind the non-monoticity of the RCIW values with regards to the number of samples on a per-benchmark basis.

5.3. RQ 1.3 – What part of instability can be attributed to noise?

To verify that our measurements are actually measuring variations brought by the benchmarks and that they do not inherently exist as fluctuations in the base energy consumption of a processor, we performed ”baseline” benchmarks. These benchmarks aim to measure the energy consumption of a processor where nothing is happening.

One might argue that a loop of no-operations is still work for the processor, as it needs to increment the loop counter and check the jump condition. We can check that there is at least no additional overhead caused by the loop using the `perf` tool, which shows that the overhead in number of cpu cycles does not grow with loop size, and neither does the number of branch-misses. In our interpretation this means that our benchmarks might be using more energy than an absolutely vacant core might do, but the stability of the consumption is not influenced. Additionally, creating a single executable that has up to one million `nop` will either get folded into a loop by the compiler, or even at a single byte per instruction not fit in program memory – which causes loads from memory as instabilities. There might be a golden middle way in which the benchmarks loop over a long list of `nop` assembly instructions that barely fits in program memory; which could be investigated in future research. Depending on the implementation on the CPU, a sleep command could act as an appropriate alternative as well.

The data in this experiment shows significantly lower values than that in the previous research question, leading us to the conclusion that the baseline energy consumption of the core itself is stable enough not to influence the measurements for actual benchmarks. We also notice in Figure 4.11 that the Relative Confidence Interval Width does not scale with the size of the benchmark loop, i.e. apart from 5 iterations there is no discernible difference between the different benchmark sizes. This could mean that the absolute Confidence Interval Width is growing with a longer benchmark. This growth gets cancelled out by the increase in energy consumption that follows from the formula $\mathrm{Energy} = \mathrm{Power} \cdot \mathrm{Time}$. We hypothesise the cause for this possible increase in absolute CIW lies within our experiment setup: because of our fixed sample period longer benchmarks have less executions within a sample frame, which could lead to less of an averaging effect within a sample.

5.4. RQ 2.1 – Does powersaving mode affect energy stability?

One of the requirements of the experimental setup is the fixed clock frequency of the core that the experiment is running on. In this research question we investigated whether is a necessary one. We find that allowing a lower, possibly variable frequency lowers the stability by a *small*[1] amount. A possible cause of this, similarly to the previous research question, is less executions of the benchmark per sample period. Once again this might decrease the averaging effect that the sample period has on the benchmarks, causing a higher variability. A more conspicuous cause for higher instability lies in the fact that the frequency of the powersave mode is not fixed; which causes the number of benchmark executions not be consistent across all samples. For this reason we recommend doing energy benchmarking on a fixed frequency, especially when benchmarking for regression.

A limitation in our answer of this research question is only investigating the baseline benchmarks. Due to time restrictions we did not run the entire suite of 868 benchmarks again with a different frequency governor, but opted to do so only for the baseline benchmarks. The result we achieve from this can be used as an indication but we recommend this research question to be investigated further with a larger number of benchmarks, that actually perform some operations.

5.5. RQ 2.2 - Does the number of instructions correlate with stability?

One feature of benchmarks that could influence its stability. As seen in the previous two research questions, benchmark length could be an explanation for the occurring phenomena . For this question we number of executed instructions as reported by the Callgrind tool. We come to the conclusion that for our set of benchmarks there is a small negative correlation between the number of instructions and the RCIW of a benchmark, i.e. a longer benchmark is more stable. This conclusion actually opposes the findings of the previous two questions, where we suggested an increase in benchmark length correlates with a lower stability. This is additional reasons to repeat the larger benchmark suite with a different frequency governor, as well as reasons to investigate the effects of different sample periods in combination with difference in benchmark length.

[1] Small as defined by Vargha and Delaney (2000)

5.6. RQ 2.3 - What code features could make software unstable?

Finally, as the last research question we look into the possible causes of instability within the source code of the benchmarks.

Using the thresholds determined in RQ 1.2 we single out a group of benchmarks that classify as unstable for any number of samples. A manual investigation of these benchmarks shows that the leading cause of instability in energy measurements is initialising the benchmarks with random data. These benchmarks come from the aHash project, but are not the only benchmarks that get loaded with random data within aHash or all projects for that matter. Since there are more benchmarks with random data that seem more stable, there must be confounding factors.

We also perform a Random Forest regression from which we learn that operations involving memory and control flow statements are most likely to predict RCIW. Since a Random Forest gives us the importance of a feature in its prediction, we use this to argue the reason for their relevance. The number of instructions is fifth on the list of importances, which we discussed previously in section 5.5

Furthermore, a general cause of difference between the different medians that have been observed between iterations at different moments in time could be related to the temperature of the environment and processor, which is known to influence power consumption (Mesa-Martinez et al., 2008). Temperature was not measured or taken into consideration in this thesis, but since it can have an effect on energy consumption, its effect on stability should be investigated as well.

5.7. Threats

5.7.1. Benchmark randomisation order

This research has shown that despite efforts to mitigate changes in environment, it is possible for benchmarks to significantly change in median consumption when running at different moments in time.

In this project we have considered the benchmarks from all projects as one large list. This was done because there was a higher expectation of instability, and thus more need for generalised datapoints to use in the prediction models. The shifted focus to the research question *How long should you measure for a stable result?* arose after it became clear the differences between benchmarks were not as stark as expected. Running the experiment in a more authentic developer-like scenario, i.e. grouping benchmarks from the same project together, and running the iterations for the same project consecutively, might give different conclusions.

We suggest investigating both batching projects together as well as the possible reason for the change in energy consumption over time. Research suggests that the temperature of the core is the reason for differences in energy consumption, but it could also be the reason for instability;.

We suggest finding a relation between the instability of benchmarks and the performance stability. This could be done by adding a timing measurement to the code of this project, or creating a new dataset and study.

It is recommended to run an entire benchmark suite instead of a single benchmark. That way a suite can be randomised and iterations can be spread to take into account changing environmental influences.

5.7.2. Sampling mode

We have used a flat sampling method, where each sample contains the same number of iterations of the tested benchmark. This was deemed necessary because of the experiment setup and the focus on finding sources of instability. Allowing a changing amount of benchmark executions per sample would mean that we no longer have control over the number of MSR energy status updates that occur per sample. When this happens it becomes more and more likely that a benchmark invocation is ongoing when an MSR update happens. Since this benchmark is now half and half in two energy measurements, it is possible that we sample the data and get incorrect energy consumption measurements. With our setup this is still occurring, but the static sample duration averages these effects over the entire iteration: each sample is influenced as much or as little as the next.

5.7.3. Measuring accuracy

For measuring energy, a Model-Specific Register (MSR) is read, which contains the amount of energy consumed by the CPU. This register is updated approximately every one millisecond. Criterion decides

the number of iterations to execute within each sample period by exponentially increasing the number of iterations in the first sample, until exceeding the period. This number of iterations is then used for the remaining samples. Since the starts of sample periods are not synchronised with the updating of the MSR and a instability in performance of the benchmarks might exist, it is possible for a measurement period of 10ms to contain eight to twelve MSR updates. This is why 100ms is a better value.

5.7.4. Experimental setup vs. code in production

In this thesis we constrained the environment of the experiment significantly to be able to correlate specific source code features to instabilities without other possible sources of noise disturbing the relations. We find there is some correlation between certain source-code features such as memory manipulation and control flow statements. Additional research needs to be performed to see whether these effects are still present in a more conventional setting, i.e. an active server or a developers PC.

5.7.5. Security Threats

Reading from Model Specific Registers (MSRs) by using the assembly requires privilege level 0, also known as kernel level privileges. This is most commonly the `root` user on Linux based systems. Any user that can exercise this privilege level has unlimited access to everything on the device.

Previous work has shown that ”software-based power side-channel attacks” are possible on Apple hardware (Chawla et al., 2023), and on Intel the ”unprivileged access to the Running Average Power Limit (RAPL) interface” (Lipp et al., 2021) has formed a side-channel attack that is able to leak cryptographic keys.

In this research we have opted for a lower level approach to collecting energy consumption than the RAPL described above, which could be used for more sophisticated attacks. Therefore care should be taken when giving unknown code access to the power consumption APIs of ones devices; also when this code has a legitimate reason to access this, such as a energy benchmarking framework. Inadvertently running code provided by a bad actor could result in leaked private keys and loss of access to critical data or infrastructure.

6

Conclusion

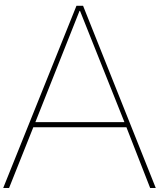
In this thesis we did research into the stability and cause of instability. To do this we found thirty projects with a total of 868 benchmarks. Each of these benchmarks has been run for thirty iterations of thirty samples, during which a total of 9000 samples of average energy have been collected. To capture the energy data we have created an extension for the Criterion-rs benchmarking framework that reads the Model-Specific Register (MSR) for energy consumption that is available on some recent CPUs on a per-core basis. Apart from the energy samples we also collected data on the source-code features of the tested benchmarks such as control flow statements and memory manipulation calls. Because of the non-normality of the data we used the robust Maritz-Jarrell version of the Relative Confidence Interval Width as our measure of stability. We find that there is a correlation between RCIW and the number of samples taken and suggest that in light of this experiment, 500 samples of 100ms is a reasonable amount to determine stability The stability at this point can be classified with a threshold of one percent. With these values, one should keep in mind they have experimentally been found under strict conditions set forth in this thesis for a limited selection of benchmarks.

For samples taken at different moments in time we find that significant differences are possible between their energy consumption, but generally – for benchmarks that one would classify as stable – the Relative Confidence Interval Width decreases with more and more samples.

We find that running benchmarks with a variable CPU clock-speed can lead to higher variability of measurements. Another clear influence on instability is initialising benchmarks with random data, although this is not always a cause for high instability and not necessarily the only cause for instability when present. Likewise we investigate the effect of the length of benchmarks (as the number of executed instructions) on their stability and find there is a small negative correlation. We can, however, not rule out that this is a consequence of our experiment setup, in which longer benchmarks have less invocations per sample than shorter benchmarks because of the fixed period samples.

Looking further at source-code features we identify control flow statements and code related to memory accesses as potential large influences of instability. Control flow statements can throw off the branch predictor and memory accesses often require waiting for the memory to load, both of which will introduce variations in energy consumption.

At the end of this thesis we provide a guideline in the number of samples to take under certain experimental setup and identify likely causes of instability. We suggest further research needs to be done into the generalisability of this guideline and under different conditions investigations isolating different possible causes of instability.



Project List

(See next page for Table A.1)

		GitHub				Crates.io		
Crate Name	**Repository URL**	**Forks**	**Stars**	**Watchers**	**Subscribers**	**Last edit**	**Total Downloads**	**Reverse Dependency Count**
ahash	https://github.com/tkaitchuck/ahash	50	636	636	16	2022-11-10 22:52:27 UTC	43633036	273
async-io	https://github.com/smol-rs/async-io	47	276	276	11	2022-11-11 01:57:21 UTC	11723126	129
base64	https://github.com/marshallpierce/rust-base64	81	395	395	9	2022-10-21 15:27:19 UTC	105474640	2618
bumpalo	https://github.com/fitzgen/bumpalo	78	863	863	16	2022-11-14 02:35:13 UTC	27892991	60
bytecount	https://github.com/llogiq/bytecount	21	177	177	8	2022-11-05 09:57:01 UTC	6909773	64
chrono	https://github.com/chronotope/chrono	387	2399	2399	28	2022-11-25 15:57:02 UTC	68672528	6058
combine	https://github.com/Marwes/combine	87	1109	1109	14	2022-09-17 12:41:55 UTC	14455293	70
curve25519-dalek	https://github.com/dalek-cryptography/curve25519-dalek	238	602	602	33	2022-11-25 17:45:43 UTC	14120291	165
handlebars	https://github.com/sunng87/handlebars-rust	109	899	899	13	2022-11-25 13:12:42 UTC	9324798	320
hex	https://github.com/KokaKiwi/rust-hex	39	136	136	3	2022-11-24 04:33:08 UTC	50082687	2340
httparse	https://github.com/seanmonstar/httparse	89	414	414	9	2022-11-25 15:37:35 UTC	57182357	189
image	https://github.com/image-rs/image	487	3396	3396	77	2022-11-25 15:50:25 UTC	9890146	1326
itertools	https://github.com/rust-itertools/itertools	237	1888	1888	20	2022-10-28 20:06:44 UTC	73768389	3315
plotters	https://github.com/plotters-rs/plotters	188	2582	2582	34	2022-11-18 17:57:06 UTC	14144567	116
png	https://github.com/image-rs/image-png.git	117	242	242	47	2022-11-25 15:14:53 UTC	10804922	203
prometheus	https://github.com/tikv/rust-prometheus	162	882	882	64	2022-11-15 10:49:12 UTC	10965940	167
prost	https://github.com/tokio-rs/prost	335	2427	2427	34	2022-11-25 09:52:57 UTC	22370303	634
pulldown-cmark	https://github.com/raphlinus/pulldown-cmark	194	1472	1472	25	2022-11-19 06:29:16 UTC	10721042	311
pyo3	https://github.com/pyo3/pyo3	469	6936	6936	70	2022-11-24 09:40:52 UTC	9348410	266
quick-xml	https://github.com/tafia/quick-xml	163	786	786	15	2022-11-18 19:18:59 UTC	9782501	291
rustls	https://github.com/rustls/rustls	413	3910	3910	70	2022-11-25 08:47:57 UTC	34467233	377
thread_local	https://github.com/Amanieu/thread_local-rs	29	192	192	6	2022-11-18 16:47:37 UTC	62671949	59
tracing	https://github.com/tokio-rs/tracing	459	3197	3197	43	2022-11-25 01:04:22 UTC	56756600	2389
tracing-log	https://github.com/tokio-rs/tracing	459	3197	3197	43	2022-11-25 01:04:22 UTC	23871778	97
tracing-opentelemetry	https://github.com/tokio-rs/tracing	459	3197	3197	43	2022-11-25 01:04:22 UTC	6313462	52
tracing-subscriber	https://github.com/tokio-rs/tracing	459	3197	3197	43	2022-11-25 01:04:22 UTC	29705595	1282
tungstenite	https://github.com/snapview/tungstenite-rs	148	1204	1204	16	2022-11-25 09:32:02 UTC	14067322	150
typed-arena	https://github.com/SimonSapin/rust-typed-arena	52	381	381	12	2022-08-02 11:53:04 UTC	7363654	72
uint	https://github.com/paritytech/parity-common	179	211	211	41	2022-11-24 10:34:19 UTC	6447374	85
unicode-xid	https://github.com/unicode-rs/unicode-xid	23	31	31	8	2022-09-15 17:15:11 UTC	125059819	92

Table A.1: Projects selected for this experiment along with the metadata they have been selected on.

B

Code Snippets

B.1. rdmsr performance tests

```
# perf stat -e duration_time -r 1000 rdmsr 0xc001029a -p 1 >/dev/null

> Performance counter stats for 'rdmsr 0xc001029a -p 1' (1000 runs):
>
>             346864 ns   duration_time                    ( +-  0.16% )
>
>       0.000348978 +- 0.000000569 seconds time elapsed  ( +-  0.16% )
```

B.2. Setting MSR read permissions

```
addgroup msr
usermod -aG msr <USER>
chgrp -R msr /dev/cpu/*/msr
chmod g+r /dev/cpu/*/msr
```

B.3. Baseline benchmarks

```
use std::arch::asm;

use criterion::{criterion_group, criterion_main, Criterion};
use criterion_energy::msr::measurement::Energy;

macro_rules! nop {
    ($group:expr, $number:literal) => {
        $group.bench_function($number.to_string(), |bencher| {
            bencher.iter(|| {
                for _ in 0..$number {
                    unsafe {
                        asm!{
                            "nop"
                        }
                    }
                }
            });
        });
    };
}

fn simple(c: &mut Criterion::<Energy>) {
    {
        let mut group = c.benchmark_group("nops");
        group.sampling_mode(criterion::SamplingMode::Flat);

        nop!(group, 100);
        nop!(group, 1000);
        nop!(group, 10000);
        nop!(group, 100000);
        nop!(group, 1000000);
    }
}

criterion_group!(name=benches;
    config = Criterion::default().with_measurement(Energy); targets = simple);
criterion_main!(benches);
```

Listing B.1: Criterion and benchmark setup for our custom benchmarks to measure baseline consumption.

C

Full plots

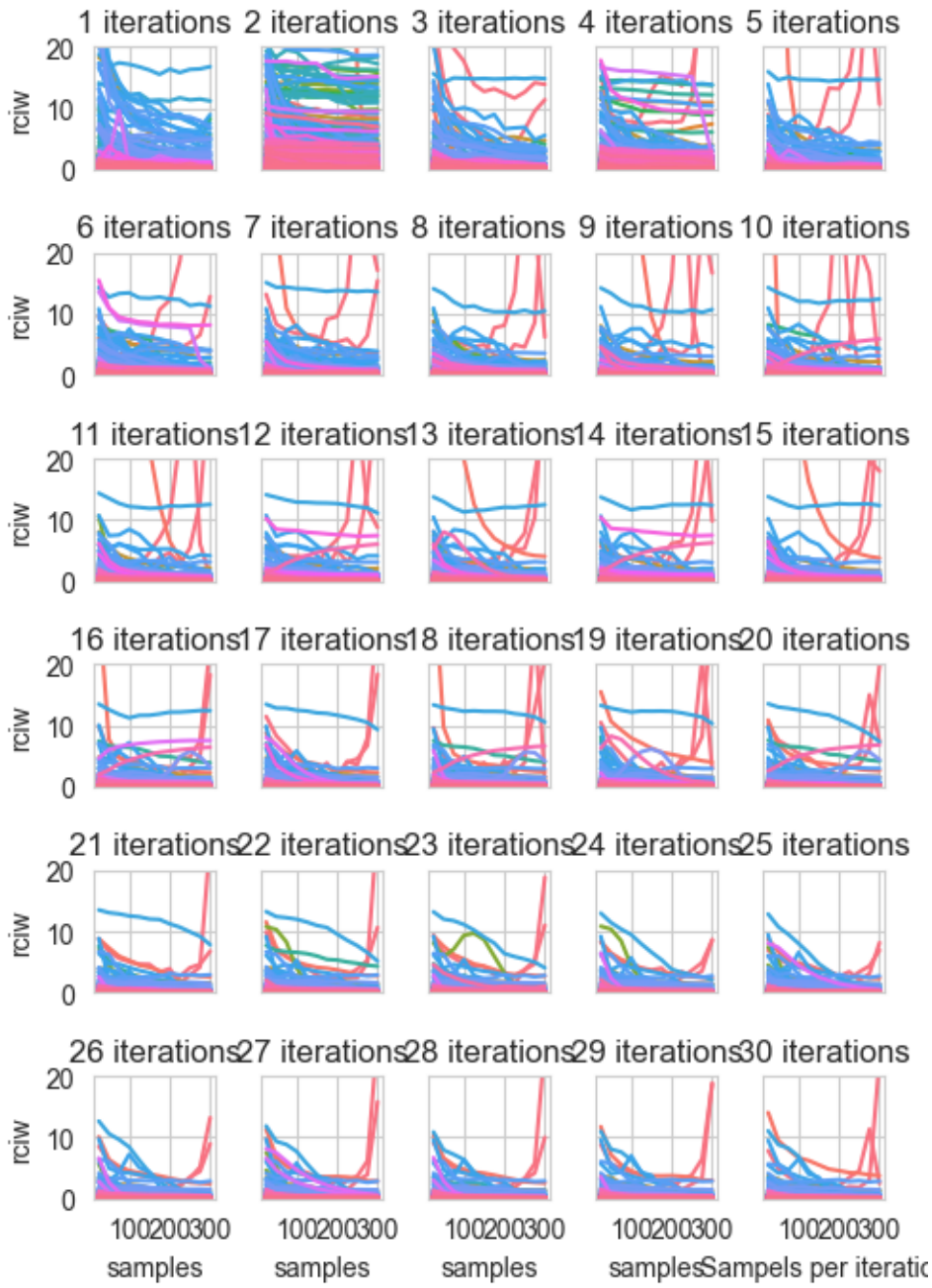


Figure C.1: All thirty iterations of RCIW vs sample per iteration plots

Bibliography

Abedi, A., & Brecht, T. (2017). Conducting repeatable experiments in highly variable cloud computing environments. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 287–292. https://doi.org/10.1145/3030207.3030229

Advanced Micro Devices. (2019). *Processor programming reference (ppr) for amd family 17h model 71h, revision b0 processors* (Rev 3.06). Advanced Micro Devices, Inc.

Bourdon, A., Noureddine, A., Rouvoy, R., & Seinturier, L. (2013). Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, *2013*(92). http://ercim-news.ercim.eu/en92/special/powerapi-a-software-library-to-monitor-the-energy-consumed-at-the-process-level

Chawla, N., Liu, C., Chakraborty, A., Chervatyuk, I., Sun, K., Hamasaki, T. M., & Kawakami, H. (2023). The power of telemetry: Uncovering software-based side-channel attacks on apple m1/m2 systems. *arXiv preprint arXiv:2306.16391*.

ClimateAction.Tech. (2023). *ClimateAction.Tech*. Retrieved January 10, 2024, from https://climateaction.tech/

D'Agostino, R., & Pearson, E. S. (1973). Tests for departure from normality. *Biometrika*, *60*, 613–622.

d'Antras, A. (2024). Minicov: Code coverage for c/c++ projects [Accessed on February 12, 2024]. https://github.com/Amanieu/minicov/tree/v0.3.3

DeVogeleer, K., Memmi, G., Jouvelot, P., & Coelho, F. (2014). Modeling the temperature bias of power consumption for nanometer-scale cpus in application processors. *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 172–180. https://doi.org/10.1109/SAMOS.2014.6893209

de Winter, J. C. F., Gosling, S. D., & Potter, J. (2016). Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. *Psychological Methods*, *21*(3), 273–290. https://doi.org/10.1037/met0000079

Duplyakin, D., Ramesh, N., Imburgia, C., Sheikh, H. F. A., Jain, S., Tekta, P., Maricq, A., Wong, G., & Ricci, R. (2023). Avoiding the ordering trap in systems performance measurement. *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 373–386. https://www.usenix.org/conference/atc23/presentation/duplyakin

European Commission. (2023). *Causes of Climate Change*. Retrieved January 10, 2024, from https://climate.ec.europa.eu/climate-change/causes-climate-change_en

GNU Compiler Collection. (2024). Gcc optimization options [Accessed on February 13, 2024]. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Green Software Foundation. (2023). Coding for a greener future: Exploring upcoming events in green software. Retrieved December 15, 2023, from https://greensoftware.foundation/articles/coding-for-a-greener-future-exploring-upcoming-events-in-green-software

Heisler, B. (2024). Criterion.rs documentation. https://bheisler.github.io/criterion.rs/book/index.html

Helmes. (2023). *Sustainable Software Engineering*. Retrieved January 10, 2024, from https://www.helmes.com/sustainable-software-engineering

IEEE. (2024). *IEEE Sustainable ICT Initiative*. Retrieved January 10, 2024, from https://sustainableict.ieee.org/

Intel Corporation. (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. Intel Corporation.

Kalibera, T., Bulej, L., & Tuma, P. (2005). Automated detection of performance regressions: The mono experience. *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 183–190. https://doi.org/10.1109/MASCOTS.2005.18

Kim, H. and Teter, J. (2023). *Aviation*. International Energy Agency. Retrieved January 10, 2024, from https://www.iea.org/energy-system/transport/aviation

Laaber, C., Basmaci, M., & Salza, P. (2021). Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering*, *26*. https://doi.org/10.1007/s10664-021-09996-y

Li, C., Ding, C., & Shen, K. (2007). Quantifying the cost of context switch. *Proceedings of the 2007 workshop on Experimental computer science*, 2–es.

Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., & Gruss, D. (2021). Platypus: Software-based power side-channel attacks on x86. *2021 IEEE Symposium on Security and Privacy (SP)*, 355–371.

Maritz, J. S., & Jarrett, R. G. (1978). A note on estimating the variance of the sample median. *Journal of the American Statistical Association*, *73*(361), 194–196. https://doi.org/10.1080/01621459.1978.10480027

Mesa-Martinez, F. J., Brown, M., Nayfach-Battilana, J., & Renau, J. (2008). Measuring power and temperature from real processors. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 1–5. https://doi.org/10.1109/IPDPS.2008.4536423

Noureddine, A., Bourdon, A., Rouvoy, R., & Seinturier, L. (2012). Runtime monitoring of software energy hotspots. https://doi.org/10.1145/2351676.2351699

Ournani, Z., Rouvoy, R., Rust, P., & Penhoat, J. (2020). On reducing the energy consumption of software: From hurdles to requirements. https://doi.org/10.1145/3382494.3410678

Perf(1) - linux manual page. (2022). https://man7.org/linux/man-pages/man1/perf.1.html

Randomized block design. (2008). In *The concise encyclopedia of statistics* (pp. 447–448). Springer New York. https://doi.org/10.1007/978-0-387-32833-1_344

Rohou, E., & Smith, M. D. (1999). Dynamically managing processor temperature and power. *2nd Workshop on Feedback-Directed Optimization (FDO-2)*.

Rozite, V. (2023). *Data Centres and Data Transmission Networks*. International Energy Agency. Retrieved January 10, 2024, from https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks

Rust Language Team. (n.d.). *Rust symbol name mangling*. Retrieved February 12, 2024, from https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html

Scheuner, J., & Leitner, P. (2018). Estimating cloud application performance based on micro-benchmark profiling. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 90–97. https://doi.org/10.1109/CLOUD.2018.00019

The Rand Project Developers. (2024). Thread_rng - rust documentation. https://docs.rs/rand/latest/rand/fn.thread_rng.html

The Rust Project Developers. (2024a). Hashmap - rust standard library documentation. https://doc.rust-lang.org/std/collections/struct.HashMap.html

The Rust Project Developers. (2024b). The `cfg` Attribute - Rust Language Reference. https://doc.rust-lang.org/reference/conditional-compilation.html#the-cfg-attribute

Vargha, A., & Delaney, H. D. (2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, *25*(2), 101–132.

Wang, Y., Nörtershäuser, D., Le Masson, S., et al. (2023). Potential effects on server power metering and modeling. *Wireless Networks*, *29*, 1077–1084. https://doi.org/10.1007/s11276-018-1882-1

Yuki, T., & Rajopadhye, S. (2014). Folklore confirmed: Compiling for speed = compiling for energy. *Languages and Compilers for Parallel Computing*, 169–184. https://doi.org/10.1007/978-3-319-09967-5_10