BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ANALYSIS OF SOFTWARE RESOURCE CONSUMPTION

ANALÝZA SPOTŘEBY ZDROJŮ V PROGRAMECH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ONDŘEJ MÍCHAL**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ PAVELA**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

146746

Institut:	Department of Intelligent Systems (UITS)
Student:	**Míchal Ondřej**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Analysis of Software Resource Consumption**
Category:	Software analysis and testing
Academic year:	2022/23

Assignment:

1. Get acquainted with the Perun project (performance version system) and the field of software profiling.
2. Study available methods for measuring software resource consumption (such as energy, memory or network usage) or other performance metrics (e.g., page faults or cache hits and misses).
3. Design and implement a Perun module that will measure at least one performance metric or a consumption of at least one resource. Study possible approaches of associating the measured consumption (or metric) to specific program constructions (functions, basic blocks or code lines).
4. Design and implement a suitable visualisation for interpreting the data collected by the resulting module (such as flame graph or heap map).
5. Demonstrate the solution on at least one non-trivial use-case.

Literature:

- Oficiální stránky projektu Perun: https://github.com/tfiedor/perun
- Gregg, B. (2020). Systems Performance, (2nd ed.). Pearson. ISBN: 9780136821694.
- Pinto, G. and Castor, F. (2017). Energy efficiency: a new concern for application software developers. https://doi.org/10.1145/3154384

Requirements for the semestral defence:
First two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pavela Jiří, Ing.**
Consultant:	Ing. Tomáš Fiedor, Ph.D.
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

Software resource consumption is a widely and actively researched area. Of the many resources utilized by software which can be profiled, energy consumption has long been the one resource without many generic, and yet comprehensive profilers. In the age of mobile devices and efficient processing units the need for such profiles is continuously increasing. In this work, we research methods for accurate measurement of energy consumption of software based on them create an open-source profiler and implement a comprehensive visualizer of the profiled data. Using the developed profiler we conduct a number of experiments to showcase its capabilities and demonstrate the usefulness of measuring software energy consumption.

Abstrakt

Spotřeba softwarových zdrojů je široce a aktivně zkoumanou oblastí. Z mnoha zdrojů v softwaru, které lze profilovat, byla spotřeba energie dlouho jediným zdrojem, který neměl mnoho obecných, a přesto komplexních, profilerů. V době mobilních zařízení a výkonných výpočetních jednotek je poptávka po takových profilerech neustále rostoucí. V této práci zkoumáme metody pro přesné měření spotřeby energie softwaru. Na jejich základě vytváříme open-source profiler a implementujeme komplexní vizualizér profilovaných dat. S vytvořeným profilerem pak provádíme řadu experimentů, abychom předvedli jeho schopnosti a demonstrovali užitečnost měření spotřeby energie softwaru.

Keywords

perun, energy consumption, profiling, version control system, vcs, ebpf, rapl, perf, system calls

Klíčová slova

perun, spotřeba energie, profilování, verzovací systém, vcs, ebpf, rapl, perf, systemová volání

Reference

MÍCHAL, Ondřej. *Analysis of software resource consumption.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Pavela

Rozšířený abstrakt

V současné době se spotřeba zdrojů softwaru stává stále častěji předmětem zájmu uživatelů i vývojářů softwaru. Vzhledem k tomu, že na trhu s elektronikou dominují mobilní zařízení, je potřeba optimální spotřeby zdrojů více než zřejmá. Ale stejná potřeba platí i pro jiné typy zařízení. Osobní stolní počítače, servery (především v oblasti HPC), zařízení IoT nebo jiné chytré stroje musí také pracovat efektivně.

Existuje poměrně široká škála různých zdrojů, které může software spotřebovávat. Pro mnoho z nich lze využít nepřeberné množství existujících analyzačních nástrojů, které pokryjí většinu, ne-li všechny případy použití. Mnohé z těchto zdrojů však ještě čekají na řádný výzkum. Zdrojem, který postrádá dobře zavedené nástroje, je spotřeba energie. Zdroje jako procesorový čas nebo paměť jsou ve většině softwaru optimalizovány, protože jejich optimalizace je nejjednodušší. Pokud by měl vývojář softwaru ve své sadě nástrojů i nástroj pro rychlou, spolehlivou a podrobnou analýzu spotřeby energie svého softwaru, mohl by začít optimalizovat software také pro optimální spotřebu energie. Optimalizace výkonu může často vést k přehřívání a obecně k nepříznivému poměru mezi dobou běhu a spotřebovanou energií, což by optimalizace spotřeby energie mohla pomoci zmírnit.

V současné době většina nástrojů pro Linux, které podporují měření spotřeby energie, neposkytuje dostatečně podrobné informace a chybí jim jakýkoliv kontext. Mezi stávající nástroje patří *powertop* [54] nebo *turbostart* [53]. Považujeme to za příležitost využít současnou nejmodernější technologii a existující výzkum k vytvoření open-source profileru pro podrobné profilování spotřeby energie. Věříme, že takový profiler by mohl být použit jako odrazový můstek pro další výzkum, zejména pokud by byl využit v nějakém širším projektu pro analýzu výkonu softwaru. Takovým projektem je Perun [21], open-source nástroj pro průběžné testování výkonu softwaru, vyvinutý výzkumnou skupinou *VeriFIT* na *VUT FIT*.

V této práci vytváříme první prototyp open-source energetického profileru, který bude integrován do systému Perun, a nástroj pro vizualizaci dat výsledných profilů. Ke sledování spotřeby energie systému by bylo možné použít externí hardware monitory, ale ty nemají požadovanou granularitu, mohou měřit spotřebu energie pouze jako celek a nejsou běžnou součástí dodávaného hardwaru u zařízení. Místo toho používáme čistě softwarové řešení využívající *Running Average Power Limit* (*RAPL*), což je funkce dostupná v moderních procesorech Intel (zavedená v generaci procesorů *Sandy Bridge*) a v současné době podporovaná i některými procesory AMD. Toto rozhraní prokazatelně [23, 28, 22] poskytuje vysoce kvalitní údaje o spotřebě energie s vysokou úrovní granularity. Pro poskytnutí kontextu pro spotřebu energie sledujeme systémová volání. Stávající výzkum [44, 15] ukazuje potenciál systémových volání k poskytnutí potřebného kontextu pro spotřebu energie. Implementace výsledků zmíněného výzkumu v plném rozsahu je mimo rozsah této práce, proto používáme mnohem jednodušší přístup, který přesto demonstruje potenciál energetického profileru. Ke sledování systémových volání a vzorkování údajů o spotřebě energie používáme technologii *eBPF*. Ta umožňuje spouštět programy v *sandboxu*[1] v privilegovaném kontextu, jako je linuxové jádro. Použitím *eBPF* dosáhneme vysoké granularity vzorkovaných dat, přičemž výkon profilovaného softwaru ovlivníme jen minimálně. Při testování jsme mohli vzorkovat spotřebu energie rychlostí tisíce vzorků za sekundu, aniž bychom významně ovlivnili výkon profilovaného procesu. Společně s profilerem jsme implementovali i nástroj pro vizualizaci výstupních profilovacích dat pomocí knihoven jazyka Python, jako jsou Pandas, Seaborn a Matplotlib. Vytvořené grafy zahrnují heatmapu nebo vodopádový graf.

[1] prostředí pro spuštění software s omezenými privilegii za účelem bezpečnosti

S vytvořeným profilerem jsme provedli řadu experimentů, které ukazují jeho možnosti. Jako předmět testování jsme použili grafický shell GNOME Shell, protože se jedná o netriviální software využívající všechny obecné hardwarové komponenty (CPU, RAM, GPU), který je velmi náchylný na režii způsobenou profilery. Pro experimenty jsme použili jeden testovací scénář, ale měnili jsme prostředí, ve kterých byly experimenty vykonávány, abychom zjistili, jak moc ovlivní výsledky profileru a jak moc je profiler náchylný na šum. Experimenty ukázaly, že profiler dokáže úspěšně lokalizovat zdroje vysoké spotřeby profilovaného softwaru a že rozdíly běhových prostředí mohou ovlivnit kvalitu výsledných profilů.

Analysis of software resource consumption

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jiří Pavela. The supplementary information was provided by Ing. Tomáš Fiedor Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ondřej Míchal
May 9, 2023

Acknowledgements

I would like to thank my supervisors Ing. Jiří Pavela and Ing. Tomáš Fiedor Ph.D. from the VeriFIT performance team for their guidance, honest feedback and support while making this thesis.

Contents

Glossary

ABI	Application Binary Interface. 16
API	Application Programming Interface. 16, 19, 46
attribute-centric data grouping	Data grouping where all the data of a single object are grouped by their attribute. 6
callgraph	A type of control-flow graph showing the relationship between function calls in software. 46
CI	Continuous Integration. 8
CLI	Command-Line Interface. 26
compositing	A process of combining multiple image sources into a single image. 43
daemon	A process running in the background. 41
display server	A software coordinating the input and output of its clients between the operating system and the system's hardware. 40
FFI	Foreign Function Interface. 27
graphical shell	A software providing a graphical user interface enabling the user to make use of the computer's capabilities in a convenient way. 40
HPC	High-performance computing. 3
IoT	Internet of Things. 3
IQR	Interquartile range. 34
JIT	Just-In-Time. 17
kernel-space	A section of computer's operating memory reserved exclusively for use by the operating system kernel. 21, 29
LLC	Last Level Cache. 18

MSR	Model-Specific Register. 18, 19
object-centric data grouping	Data grouping where all the data of a single object are grouped by the object they belong to. 6
PID	Process ID. 21, 22, 28
PMU	Processor Monitoring Unit. 29
ringbuffer	A data structure with fixed size buffer allowing to loop through the buffer. 28, 29
sysfs	A pseudo file system in the Linux kernel providing access to various information about the kernel, drivers and hardware devices. 29
TGID	Thread-Group ID. 22
TID	Thread ID. 28
tmpfs	A temporary file system where data is stored in a volatile memory rather than persistent storage. 28
user-space	A section of computer's operating memory outside of kernel-space. 15, 17, 21, 26, 29, 38
VCS	Version Control System. 9–11

Chapter 1

Introduction

Nowadays, resource consumption of software is increasingly becoming a concern for users as well as software developers. With mobile devices dominating the electronics market the need also for having optimal resource consumption is more than clear, but the same need applies to other types of devices. Personal desktops, servers (mainly in the HPC space), IoT devices or other smart machines need to perform efficiently as well.

There is quite a high number of different resources a software can consume. For many of these one can use an abundance of existing tools analysers to cover most, if not all, use cases. However, many of the resources are yet to be properly research. A resource lacking well established tools is energy consumption. Resources like CPU or memory are being optimized for in most software as their are the easiest to optimize for. If a software developer had in their toolbox a tool for quick, reliable and detailed analysis of their software energy consumption, they could start optimizing the software for optimal energy consumption as well. Optimizing for performance can often lead to overheating and in-general unfavourable execution time/energy consumed ratio which optimizing for energy consumption could help to mitigate.

Currently, most tools that support measuring energy consumption do not provide detailed enough information and lack any context. We see this as an opportunity to make use of current state-of-the-art technology and existing research to create an open-source profiler for detailed profiling of energy consumption. We believe such profiler could be used as a stepping stone for further research, especially if employed in some wider performance analyses project.

In this work we build on the research of the domain of energy profiling and different technologies and techniques for profiling energy consumption. We will apply these in a novel open-source energy profiler and a reporter of its findings with support for non-trivial visualizations. With the created profiler we then conduct a list of experiments to showcase its capabilities and demonstrate the usefulness of knowing energy consumption. As the last step we want to integrate the created profiler in Perun, a performance version system.

Structure of the Thesis Chapter 2 summarizes existing research related to the areas of energy profiling and energy-efficient programming. The performance tracker Perun is introduced in Chapter 3. Chapter 4 explains the *eBPF* technology, how to use and how it is used in this work. Chapter 5 cherry-picks technolgies and approaches to energy profiling that will be leveraged in this work from the research in Chapter 2. Chapter 7 covers the technical details of the resulting profiler and a reporting tool for creating visualizations of the profiled data. We explain the profiler's architecture and data format, how well does it

perform, and how exactly the visualizer works and what does it produce. The last Chapter 8 showcases on several existing open-source projects the capabilities of the created profiler.

Chapter 2

Background and Related Work

The task of software profiling is a complex one and researching a novel profiling approach is outside the scope of the thesis. Instead, this thesis heavily relies on research conducted in the past, mainly in the area of energy consumption profiling. Past research spans widely from exact approaches to profiling, to analysis of knowledge of developers on the topic of energy profiling. In this section we will briefly introduce the field and list selected related work. The related work is organized into three categories consisting of knowledge of developers about energy consumption (Section 2.1), programming-language-specific insights into their energy profile (Section 2.2), and existing approaches to energy profiling (Section 2.3). Additionally, research of side-channel attacks (Section 2.4) is mentioned as this group of attacks exploits weaknesses in software that could potentially be detected using the results of this thesis.

A great introduction to the scientific research of energy profiling is an article by Pinto and Castor about energy efficiency [46] where the authors discuss the general need to focus on the energy efficiency of software, the need to educate developers about the topic, and the existence of various research in the area. This chapter is based on this article and the research the authors reference in it.

2.1 Knowledge of developers

Profiling is a type of dynamic software analysis that requires from the developers deep knowledge of the problem domain to be able to discern what information needs to be collected and what the results of the analysis mean. Energy profiling being a specialized domain of profiling makes it potentially an even more difficult type of analysis.

Pinto et al. [46] conducted a survey on a sample of software developers with the goal to understand their perceptions about software energy consumption issues. The results of this survey uncovered that while 67% of the respondents do care about energy-related features, only in 50% of the cases when the respondents addressed energy-related issues in a mobile application only a fraction of the respondents used specialized tools and the rest of them depended on their own perception of an improvement. The sources of insight into solving the energy-related issues were mainly non-empirical, e.g. official documentations, StackOverflow, YouTube, blogs and other sources. These results make it clear that there is a lack of tools for developers to make informed software changes to address energy-related issues.

Pang et al. [43] conducted a survey yielding similar results. The authors summarized the results of the survey into 4 takeways for programmers: (1) having limited awareness of

software energy consumption (2) lacking knowledge of reducing software energy consumption (3) lacking knowledge of software energy consumption and (4) not being aware of software energy consumption's causes.

The conducted surveys suggest that software developers are under-equipped in both tooling and knowledge to correctly and accurately detect, analyze and fix energy consumption-related issues in software. There is a need for an easy-to-use tool providing detailed insight into the energy profile of software.

2.2 Energy efficiency of software constructs

Data structures are a vital part of any software and their knowledge is one of the fundamentals computer science students are taught. Common knowledge is, for example, that time complexity (be it asynmptotic or amortized) of operations differ largely on the type of underlying data container. Time complexity is one of many useful information for determining the performance and potential energy consumption, but it is not the only one. Performance and energy costs of different data structures, abstractions, threading models and more have been the target of a large number of researchers over the years.

Many papers analyze energy profiles of different software constructs. Pinto et al. [48] analyzed the energy efficiency of several implementations of data structures (lists, sets, and maps) in Java and their findings show significant differences. For example, an alternative implementation of a hashtable (`ConcurrentHashMapV8`) yields up to 2.19x energy savings in micro-benchmarks and up to 17% of savings in real-world benchmarks over the old implementation (`ConcurrentHashMap`). Hasan et al. [24] similarly studied the efficiency of data structures in Java in three different implementations. Using the gained knowledge, the authors created worst-case and best-case scenarios in several open-source libraries and applications. The results ranged from minimal differences of less than 1%, and up to 300%, in energy consunmption efficiency. Lima et al. [33] studied the energy efficiency of different data structures in Haskell. The results showed both marginal (below 1%) and significant (over 25%) differences in energy consumption and execution time when a different data structure with similar/same API was used.

Liu et al. [34] conducted an empirical study on how data access patterns, data precision choices, and data organization affect energy consumption in Java. Additionally, the authors studied how various application-level data management features respond to Dynamic Voltage and Frequency Scaling (*DVFS*). Among their findings is that Object-centric data grouping can be less energy efficient than Attribute-centric data grouping, and that down-scaling a CPU often leads to worse results in both performance and energy consumption. The down-scaling of a CPU lead to better results in energy consumption mainly in cases of programs performing excessive number of I/O operations.

Lima et al. [33] also studied three different thread management constructs and data sharing primitives in Haskell. The results show differences in energy consumption in tenths of a percent and show that there is no universaly better solution. Choosing the correct construct depends on the context of the specific application and profiling is needed to identify the most suitable constructs. Pinto et al. [47] performed an empirical study analyzing energy consumption of different thread management construct. Their findings corroborate to the fact that faster execution times do not always lead to lower energy consumption. In fact, the opposite is usually the case. The curves for the execution time and the consumed energy depending on the number of used cores are not the same. The execution time curve usually

display an inverse logarithmic shape while the consumed energy curve display usually a Λ (lambda) shape.

Implementation details of software can significantly affect ways energy consumption of software and there are often no universally applicable solutions yielding optimal results.

2.3 Energy profiling techniques

Aggarwal et al. [15, 14] explored in their work the use of *system calls* to detect changes in the energy profile of software across different versions. Their findings show that system calls are related to power consumption. Combining system call profiles with statistical methods gives a developer an easy-to-use tool capable of accurate assessment. The model authored by Aggarwal et al. achieved at least 80% accuracy of detecting differences in power consumption across different versions.

Pathak et al. [44] created a fine-grained energy profiler for mobile devices. They explored the granularity of energy accounting for the different components in a mobile device and asynchronous power behaviour of some modules (e.g., GPS, WiFi and Bluetooth modules). They based the model, similarly to Aggarwal et al. [15, 14], on a system call power model which provided them with fine-grained source data. The maximum measurement error reached only 6% for all tested applications. The sample of tested applications consisted of both simple (e.g. a Sudoku game) and complex (e.g., Angry Birds or Facebook) applications.

Li et al. [32] created a solution for calculating the energy consumption of single lines of code in Android applications. Using the combination of external power meters readings, program analysis, and statistical modelling the authors created a highly accurate solution. The error in the calculated energy values were within 10% of the ground truth measurements and the statistical models had a high R^2[1] average of 0.93.

The research and energy profiling techniques in this section are further discussed in Chapter 5.

2.4 Side-channel attacks

In this section we'll introduce one of the application fields of exploiting energy consumption in practice –side-channel attacks. They are a family of attacks relying on physical parameters of the compromised system, such as electromagnetic emissions, execution time, power consumption [50] and others. Two of the most common techniques of side-channel attacks are the following.

Simple Power Analysis (*SPA*) is a technique involving direct interpretation of power consumption measurements collected during critical operations. The collected measurements are interpreted directly and allow to differentiate between kinds of operations (e.g., multiplication/squaring). The technique is simple as it does not involve any post-processing or more involved measuring. Defence mechanisms in hardware (e.g., protective cases) and software (e.g., branch-less programming, noise generation) are quite effective at limiting the susceptibility to the attack. The intention is to lower the power consumption variations such that *SPA* will not yield any key material. [30, 31]

[1] coefficient of determination

Differential Power Analysis (*DPA*) is a technique consisting of two phases: data collection and data analysis. The data collection phase is similar to *SPA* and the collected information is usually the device's energy consumption. The second phase involves statistical analysis and error correction techniques. Their application enables to identify key material at a much smaller scale. Defence against *DPA* involves several approaches, e.g., reducing signal sizes, introducing noise into power consumption measurements, and designing cryptosystems with realistic assumptions about the underlying hardware. [30, 31]

A recent example of a real side-channel attack is *hertzbleed* [55] which is based on a proof that *Dynamic Voltage and Frequency Scaling* (*DVFS*), power consumption and currently processed data directly affect each other. This allows for a remote timing attack where the attack side-channel is the execution time. The authors published along with their research paper the source code for their programs capable of reproducing the research side-channel attack on GitHub [58].

There is a large number of existing projects and solutions both open-source and proprietary for experimenting with side-channel attacks. Some of them are:

- *Pysca*, a toolbox for advanced differential power analysis of symmetric key cryptographic algorithm implementations [29]
- *Jlsca*, a side-channel toolkit in Julia [18]
- *ChipWhisperer*, the complete open-source toolchain for side-channel power analysis and glitching attacks [2]
- *eShard*, a comprehensive Side Channel analysis solutions covering profiling and non profiling attack techniques, including deep learning [7]

Prevention of side-channel attacks is difficult and involves several, often only partially related, approaches. One of them is lowering power consumption variations. The result of this thesis could potentially help to uncover areas for improvement in this regard. Integration in a *performance version system* such as Perun (see Chapter 3) could provide high accuracy (using post-processing and visualization modules) and automation (by integrating Perun-based testing into CI systems).

Chapter 3

Perun

***Per**formance **Un**der Control* (*Perun*) is an open source *Performance Version System* used for continuous tracking of a project's performance [20]. The project integrates with VCSs like *Git* or *SVN*, thus keeping performance testing closely tied to it. *Perun* also provides a tool suite for specification of test runs, performance metrics collectors, performance data postprocessors and analyzers. The results of this thesis will be in near future integrated in Perun. To provide a better understanding of the content of the work, we will provide an overview of Perun (Section 3.1), describe its architecture (Section 3.2) and workflow (Section 3.3) in the following sections.

3.1 Overview

VCS track how the code base of a project evolves, how the functionality changes, provides versions snapshots (e.g., tags) and often provide additional generic functionality in order to satisfy as many needs as possible. These systems are often flexible enough for keeping track of additional data but they are not optimized for such use cases. Perun fills the gap for tracking a project's performance by using VCSs to gain insight into a project development history and store its results into a `.perun` directory in the project tree (similarly to *Git* storing its state in a `.git` directory).

Perun offers the following advantages over the sole use of VCS or databases [20]:

- **Context.** Results of test runs (so called performance profiles) are tied to an exact version of the corresponding VCS, thus providing the necessary context (e.g., changed lines of code) to focus the analysis efforts on the recent differences in the source code only.
- **Automation.** Manual analysis is a common activity, but not in the context of automated test system. Perun provides a concept of *jobs* for defining test runs and a concept of *hooks* reacting to VCS events (e.g., commit, push, tag).
- **Genericity.** Every project is different and their expectations of Perun may vary. Perun thus provides a framework allowing for a straightforward extension of its capabilities (collectors, postprocessors and visualisations) via modules. Its data format (based on a *JSON* notation) is flexible enough for a quick adaption.

- **Ease of use.** Perun is available as a CLI[1] application with commands similar to common VCS applications, which lowers the learning curve of using the tool.

3.2 Architecture

Perun's architecture, illustrated in Figure 3.1, is made of four main components: *logic*, *data*, *check*, and *view*. The main components are complemented by a collection of utility methods as well as move advanced or specialized features: *VCS*, *workloads*, *fuzzing*, etc.

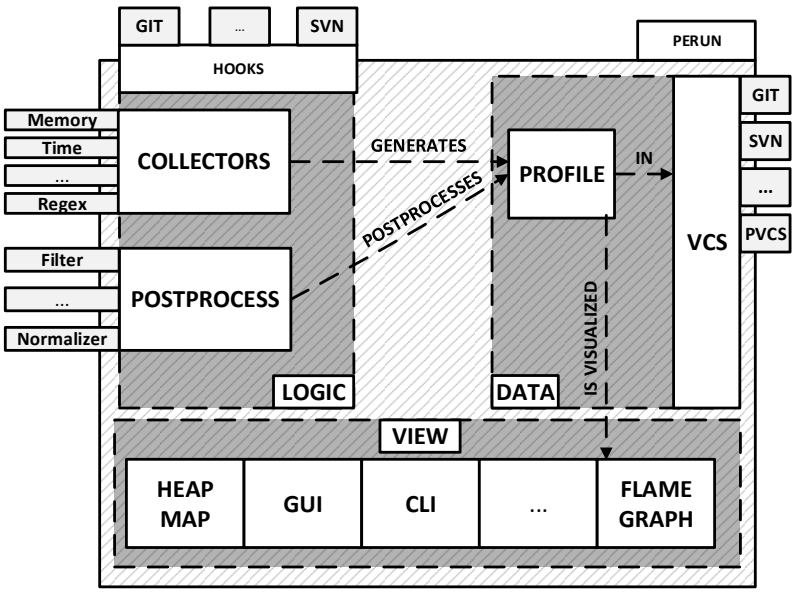


Figure 3.1: An overview of Perun's architecture showing its main components, *VCS* integrations and the flow of data in the program. [21]

Logic is responsible for the generation of performance profiles, automation (hooks) and higher-logic manipulations [45, 21]. Its two major components are:

- **Collectors** collecting performance data. They can be implemented either as wrappers of existing profilers and utilities or they can be fully-featured profileres on their own. Notable collectors are: *trace collector*, *complexity collector*, or the simple *time collector*.
- **Postprocess** applies various statistical analysers to the collected performance profiles. Notable postproceessors are: *moving average postprocessor*, *regression analysis postprocessor* or *clusterizer postprocessor*

Data is the core of Perun –performance profile manipulation is done in this component together with its storage in a VCS. It is the middleware between *logic* and *view*. [45, 21]

[1] Command Line Interface

Check implements various performance regression detections across different revisions (profiles) of a project. Several detection methods are available due to the high sensitivity of the used statistical techniques to the type of analysed data. Notable detection methods are: *linear regression*, *integral comparison*, or *average amount threshold*. [45]

View provides means for the collected and processed performance profiles to be visualized. Some of the visualizers are: *bars plot*, *flame graph* or *scatter plot* [45, 21]

3.3 Workflow

Perun's workflow can be illustrated in a series of steps the user should perform. The description of these steps is based on [45, p.24]. See Figure 3.2 for a visualization of Perun's workflow.

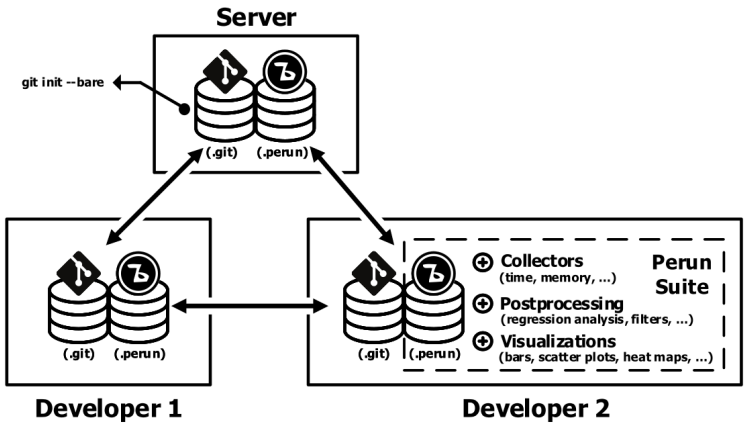


Figure 3.2: An illustration of the Perun workflow on a project repository using the Git VCS. [21]

The steps of an usual Perun's workflow can be described as follows:

1. We initialize a new Perun repository in a project managed by a VCS using `perun init` (the initialized Perun repository is not managed by the VCS).
2. We configure the initialized Perun repository and select the desired collectors, post-processors, workloads and other options.
3. We make changes to the tracked project and commit them into VCS (this may trigger the next step automatically depending on the configuration from the previous step).
4. We collect raw data using either the pre-configured collectors or using a manually selected collector using `perun collect`. The raw data are stored as a profile in the Perun repository.
5. We optionally process the raw data using either the pre-configured post-processors (e.g., regression analysis, clusterizer) or using a manually selected post-processor using `perun postprocessby`. The processed data are stored in the current profile in the Perun repository.

6. We compare the current profile with profiles paired with previous revisions of the project using either the pre-configured degradation checker (e.g., integral comparison) or a manually selected degradation checker using `perun check`. The check is done only for matching pairs of tested $binaries + workloads + configurations$. The results of the check are stored as a new object in the Perun repository.

7. We assess the severity, location and confidence parameters of reported potential performance changes.

Chapter 4

eBPF

> Super powers have finally come to Linux.
>
> Brendann Gregg

eBPF is an instrumentation framework allowing to run sandboxed programs in a privileged context like in the Linux kernel [8]. It allows to extend the capabilities of the kernel without changing the kernel source code or loading custom kernel modules. Efficiency and security of the executed programs are ensured thanks to JIT[1] compilation and a verification engine. We will use the technology in Chapter 7 for the implementation of profiling capabilities of the energy profiler and thus we first briefly introduce its capabilities and features.

4.1 Overview

At the core of *eBPF* stands a virtual machine running in a kernel of an operating system. This virtual machine has its own instruction set and clearly defined rules for programs running in it. Rules are set because *a)* all *eBPF* programs have to be guaranteed to finish, *b)* memory accesses are typed and bounded, and *c)* programs can have at maximum `BPF_MAXINSNS` instructions (4096 by default). These rules are necessary for the verifier to be able to reliably and quickly check all loaded programs. A program can be loaded using the `bpf()` system call which accepts *eBPF bytecode*. Such bytecode can either be handcrafted or generated. The standard compiler for generating the bytecode is *Clang* compiler containing a *eBPF* back-end.

eBPF programs run in kernel-space and very often they need to be configured before running or share data at runtime with user-space. For such purposes one can use so called *maps*. They are generic storage types in kernel-space available to *eBPF* programs and user-space. They can be used as hash tables, arrays or stacks. We can also use them for storing per-CPU data. A special map is ring buffer (`BPF ringbuf`). It is ideal for continuous data delivery from kernel-space to user-space (e.g., for logging). It offers great performance and ensures data order.

We mentioned that an *eBPF* program bytecode needs to be loaded and often initialized before running. Instead of accomplishing that manually using the `bpf()` system call this

[1] Just-In-Time

can also be accomplished using a *loader*. A loader takes care of loading an *eBPF* program, its initialization and how it is run.

After a *eBPF* program is loaded and verified, it is then further submitted to JIT[2] compilation that optimizes the performance of the program. Overall, *eBPF* is an instrumentation framework with excellent performance, high reliability and security, clear boundaries for use, and straightforward ways to use.

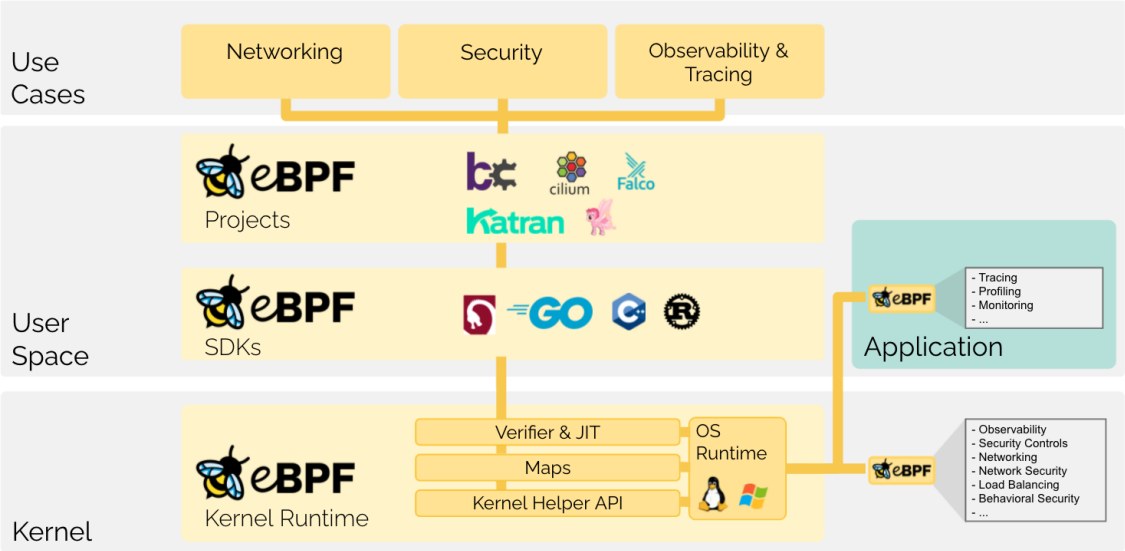


Figure 4.1: A diagram of *eBPF* architecture and how technologies utilizing it are structured. [8]

4.2 Developer toolchains

Writing *eBPF* bytecode by hand is not a recommended venture since the kernel ecosystem keeps evolving at an incredible pace. A possible solution is to leverage abstractions built on top of *eBPF* like *bcc* (4.2.1), *bpftrace* (4.2.2), or *libbpf* (4.2.3), providing a more expressive environment for writing *eBPF* programs. The abstractions universally depend on the Clang/LLVM toolchain implementing an *eBPF* bytecode backend for compilation of *eBPF* programs.

4.2.1 BCC

BPF Compiler Collection (*BCC*) is a toolchain providing C, Python and Lua interface for writing *eBPF* programs. The *BCC* toolchain is based on the LLVM toolchain. It provides a BPF-specific front-end for C, making it easier to write valid *eBPF* programs. Along the C language front-end, additional front-ends for Python and Lua are provided [9].

Since 2020, the Python interface of *BCC* is considered deprecated for writing new performance tools [17]. It is recommended to use the C interface and *bpftrace* for quick *eBPF* scripting.

[2] Just In Time

4.2.2 bpftrace

bpftrace is a high-level frontend for *eBPF* utilizing *BCC* for interfacing with the *eBPF* Linux system. It is ideal for writing *one-liners* and short scripts. The *bpftrace* language is inspired by *awk*, *C* language and tracers like *DTrace* and *SystemTap*. [1]

An example of a *bpftrace* one-liner counting the number of page faults by process is:

```
$ bpftrace -e 'software:faults:1 { @[comm] = count(); }'
```

4.2.3 libbpf

libbpf is a user-space *C/C++ eBPF* loader library providing APIs for interacting with the *eBPF* Linux system. *libbpf* bootstraps the loaded *eBPF* object file, allowing for hooking up to different phases of *eBPF* program lifetime, and providing useful API similarly to *BCC* (albeit not as high-level). The development of *libbpf* over the years lead to closing the gap in features between *BCC*, fixing issues in the API or addressing scenarios of incompatibilities between different versions of the kernel. This work culminated in the release of version 1.0 in August 2022. [39]

The lifetime of *eBPF* programs has four phases for which *libbpf* provides a way to define them and execute them. These phases are [37]:

1) **Open phase** loads up the *eBPF* object file into the memory, upon all present *eBPF* programs, maps and global variables are made available. It allows one to make adjustments to the structures because the program has not been loaded into the kernel, yet.

2) **Load phase.** creates *eBPF* maps, verifies the programs and loads them into the kernel, but the programs are not executed, yet. At this phase it is still possible to adjust the state of *eBPF* maps.

3) **Attachment phase.** starts the *eBPF* program by attaching it to its defined hook points.

4) **Teardown phase.** detaches and unloads *eBPF* programs from the kernel. All *eBPF* resources (e.g., maps) are freed.

bpftool

To make it convenient for programmers to work with *eBPF* programs exists the *bpftool* tool. It serves for inspecting and manipulating *eBPF* programs and maps. One can also use it to generate a *skeleton header file* from an object file of an *eBPF* program to ease the development of User-space programs. The generated header file contains the *eBPF* program bytecode, provides functions for loading it and provides typed access to all maps, programs, global variables and such.

4.3 BPF CO-RE

BPF Compile Once –Run Everywhere (*BPF CO-RE*) [38] is an approach to creating *eBPF* programs that solves many issues with portability of *eBPF* programs. The technology was introduced at the LSF conference in 2019 [5]. Many *eBPF* tools originally written using BCC or other toolchains have already been converted to *BPF CO-RE*. [38]

Portability of *eBPF* programs is troublesome. The user-written *eBPF* programs need to work within the set kernel environment over which they have little to no control. Any program requiring access to raw internal kernel data can not rely on any notion of stability from the kernel. *BCC* tackles this problem by embedding an *eBPF* program in a user-space binary and compiling it on-the-fly when requested. The compilation uses *Clang/LLVM*, embedded in *BCC*, and kernel headers. Programs created by *BCC* then require it to be installed on the system to compile the programs on-the-fly and the programs are inherently unstable due to possible ABI/API change in the Linux kernel (e.g., they can contain unhandled cases of renamed structures across kernel versions). Note that, kernel headers are not installed on systems by default which are required for *Clang/LLVM* to be able to compile an *eBPF* program. [38]

BPF CO-RE overcomes the portability problem of *eBPF* programs by leveraging several functionalities in the used components: kernel info format (*BTF*), compiler (*Clang*), and user-space *eBPF* loader library (*libbpf*).

4.3.1 BTF

BTF stands for *BPF Type Format*. It is a debugging data format which is an alternative to the *DWARF* format. It achieves up to 100x size reduction over *DWARF* while containing all necessary type information of C programs. Having such a space-efficient data format makes it feasible for including it in Linux operating systems for the Linux kernel by default. The information is available at path `/sys/kernel/btf/vmlinux` and can be used to generate a C header file using the tool `bpftool`. [38]

4.3.2 Clang

The *Clang* compiler was extended with a certain feature. It makes it possible to track field existence, removals, offset relocations or size changes. These bits of information are then emitted for the *eBPF* program loader to pick up and use it for making the necessary adjustments. [38, 37]

4.3.3 libbpf

Before loading an *eBPF* program and then submitting it to the verification engine, *libbpf* inspects the *eBPF* object file and, if necessary, makes adjustments to it to match the *BTF* information of the running kernel. These adjustments are done completely transparently and automatically. [38]

4.4 eBPF in practice

Since its creation *eBPF* has been constantly growing in popularity and the number of use cases for it only keep growing. Nowadays it is possible to not just monitor and profile but also change the behaviour of the kernel at runtime [4]. Some noteworthy *eBPF* projects are:

- *libbpf-tools* and *bcc-tools* are collections of small but very useful *eBPF* programs contributed by the members of the community made available for all users to use for profiling. [9]

- *ebpf_exporter* is a metrics collector/exporter for Prometheus [49] created by CloudFlare. [11]
- *bpfilter* is a direct competitor of *iptables* and *nftables*. It offers a compatibility layer translating iptables rules into *eBPF* programs while also offering the possiblity of writing firewall rules in C. [19]
- *wachy* is a dynamic tracing profiler providing a TUI[3]. [10]

4.5 Alternative technologies

In this chapter we've introduced the *eBPF* instrumentation framework, but there exist alternative instrumentation frameworks. In this section we'd like to introduce some of the alternatives we've considered when selecting the appropriate technology for the implementation of tracing capabilities of the created energy profiler.

	eBPF	SystemTap	Pin
Kernel-space instrumentation	✓	✓	
User-space instrumentation	✓	✓	✓
Instrumentation stability	✓		✓
Instruction set specific support			✓
Requires kernel debuginfo		✓	
Relies on recent kernel version	✓	✓	

Table 4.1: A comparison of instrumentation frameworks *eBPF*, *SystemTap* and *Pin*

SystemTap is a general-purporse tracing and profiling framework for the Linux kernel. It supports a large number of mechanisms for gathering data about both the kernel and processes running on the system. In general it is regarded as one of the most powerful profiling frameworks. *SystemTap* instruments code by loading a a custom kernel module into the Linux kernel. These modules are created automatically by *SystemTap* from custom scripts which are then first transpiled into the C language. For most of its functionality, *SystemTap* requires for the Linux kernel to be present together with the *SystemTap* program. And because *SystemTap* is implemented out-of-tree of the Linux kernel and is not designed with safety/resiliency first, it has a history of causing system lock ups, freezes or even kernel panics. [36]

Intel Pin is a dynamic binary instrumentation framework developed by Intel enabling the creation of dynamic program analysis tools. *Pin* is mainly used in tracing of User-space applications without the need to recompile the traced binaries. It also provides cross-platform support and in general offers stellar performance. *Pin* instruments code on the instruction level by using JIT compiler. *Pin* ensures that the original behaviour of the analysed software is kept even during profiling. *Pin* operates on a level above the operating system and thus can only capture user-space code. It also supports only a limited number of instruction sets (IA-32, x86-64 and MIC). [36, 6]

[3] Text-based User Interface

Chapter 5

Existing Methods and Technology

In Chapter 2 we introduced a number of methods for energy profiling. In this Chapter we will further describe them in the context of this work. We will emphasize on their strengths and limitations and discuss their potential usage for the created energy profiler.

5.1 RAPL

Running Average Power Limit (*RAPL*) is an interface introduced by Intel in the Sandy Bridge architecture of Core processors. It allows controlling the power consumption limits. This ability to control also comes with the ability to monitor. Originally this feature was backed by purely software models of consumption. With the Haswell architecture the interface became backed by hardware power meters embedded in the chip which significantly raised the accuracy of the power readings. The interface is made of non-architectural MSRs. [28]

This thesis is set in the context of the Linux kernel, so the following paragraphs will use its terminology for the *RAPL* interface (mainly domains).

The *RAPL* interface exposes several domains (visualized in Figure 5.1):

a) **cores** shows the power consumption of all cores in a package,

b) **pkg** shows the power consumption of all cores in a package together with LLC,

c) **gpu** shows the power consumption of so called *uncore* device (on desktop this typically corresponds to integrated GPU),

d) **dram** shows the power consumption of DRAM,

e) **psys** shows the power consumption of the whole platform or System-on-Chip (SoC).

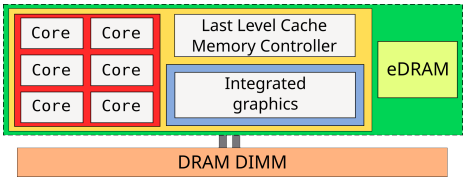


Figure 5.1: Power domains supported by *RAPL* (visualization based on [28])

The different domains provide a reasonable level of insight into a systems energy consumption. But it does not cover the whole system. The interface is closely tied to the CPU and the components closely tied to it. It does not provide insight into other areas of the system including peripherals, Wi-Fi modules, Bluetooth modules and more. Relying purely on the *RAPL* interface as the only source of energy consumption will cause the resulting profile not to have all energy consumption information of the system. Despite the fact, prior art [15, 14, 44] shows that the power readings correlate well with the whole system's power consumption and thus the data incompleteness is not fatal.

The interface was first made available in the Linux kernel in version 3.14 [57] and it was an interface implemented only for Intel CPUs. In a later version of the Linux kernel the interface has been generalized so that other hardware drivers can make use of it. This made it possible for AMD to add support for their AMD Ryzen processors (though in a limited form). The AMD implementation originally exposed only the *package* domain and recently exposed also the *core* domain. If the other MSRs for other domains are present, they are deemed unreliable enough to not be made available.

The precision of the *RAPL* interface is important for assuring high quality profiling data. It does differ across vendors and architectures. The interface exposes for all domains three values: *a)* an actual *value b) unit* usually as a string with the name of the unit *c) scale* used for scaling the *value* to fit the *unit*. E.g., on Intel Skylake processor architecture the *unit* is `Joules` and its *scale* is `2.3283064365386962890625e-10`, making a single increment of *value* an increment of `0.23 nJ` (nano-joule).

High precision of the *RAPL* interface is not a guarantee of high accuracy. A number of researches [23, 28, 22] on the accuracy of the *RAPL* has already been conducted. The results generally show that the interface provides data with high level of correlation with data measured by AC power meters [28, 22]. The quality of data provided by the interface has even been increased, since its first introduction in the Sandy Bridge architecture, thanks to the transition from pure software models to *fully integrated voltage regulators* (*FIVR*) [23].

5.1.1 Reading the values

In the Linux kernel there exist several ways of reading the hardware information. In the following, we will list these ways together with their basic principles and caveats.

For the purposes of this thesis a single optimal approach needed to be chosen.

sysfs

The first approach uses of *sysfs*, a pseudo file system in the Linux Kernel available at the `/sys` path in. In particular, the *RAPL* interface is available under `/sys/class/powercap/intel-rapl`. This approach is the one with the lowest entry barrier as it only involves reading the content of files and thus requires no special knowledge. But for the purposes of this work this approach is not optimal due to its inefficiency for high frequency measurements. The inefficiency is caused by the file API not being designed for high frequency re-reads of the same file.

perf_events

Perf is a tracing and profiling tool developed in parallel with the Linux kernel allowing instrumentation of CPU performance counters, tracepoints, kprobes[1], uprobes[2] and a lot more. Alternatively it can be referred to as `perf_events`. One can make use of *perf* directly using the *perf* CLI application which has a broad offering of features or using the `linux/perf_event.h` C header file to create individual applications (for more advanced usage). `perf_events` are quite efficient at being used for high frequency tracing.

A minimal example of using `perf_events` from C code is:

```
#include <linux/perf_event.h>
#include <sys/syscall.h>
#include <unistd.h>

struct perf_event_attr attr;
long long clock;
int fd;

attr.type = PERF_TYPE_SOFTWARE;
attr.config = PERF_COUNT_SW_CPU_CLOCK;
attr.freq = 1;
attr.sample_period = frequency;

fd = syscall(SYS_perf_event_open, &attr, 0, 0, -1, 0);
read(fd, &clock, sizeof(clock));
close(fd);
```

RAPL perf events belong to a certain category of events: Kernel PMU[3] events. Since this category of events depends on the hardware architecture, their availability is not assured. One can check the availability of different perf events using the *perf* CLI application by using the `perf list` command or one can search the content of *sysfs* (`/sys`) for the different events.

The issue of watching multiple PMU perf events is that it cause so called *time multiplexing*. During the multiplexing the kernel switches which event uses the capabilities of the hardware. This phenomenon skews the monitored data which need to be scaled to at least approximate of the true data. Thankfully, this phenomenon does not happen with *RAPL* because at least on Intel architectures the interface has its own dedicated counters.

Comparing the approaches

To demonstrate the two approaches to reading energy consumption via the *RAPL* interface, we will conduct a series of benchmarks using different sample programs utilizing the different approaches. In total we will conduct three benchmarks:

a) **sysfs**, where a C program reads at a set interval (using `<sys/timerfd.h>`) of 9999Hz the content of *RAPL* files in `sysfs`,

[1] kernel probes
[2] user probes
[3] Performance Monitoring Unit

b) **perf-userspace**, where a C program reads at a set interval (using `<sys/timerfd.h>`) of 9999Hz perf events of all *RAPL* domains,

c) **perf-ebpf**, where a C program reads at a set interval of 9999Hz perf events of all *RAPL* domains using an *eBPF* program.

For this benchmark we use the same testing environment as in Chapter 8. In the benchmarks the provider of the results are the `perf` and `bpftool` utilities. The `perf` utility is run as `perf stat -e cycles -e instructions -e L1-dcache-loads -e LLC-load-misses -p <benchmark-pid> -timeout 5000` where `cycles`, `instructions`, `L1-dcache-loads`, and `LLC-load-misses` are the benchmarked statistics, `<benchmark-pid>` is the PID of the benchmarked program which was started prior to the start of the `perf` utility, and `5000` is the duration in milliseconds for which the `perf` utility is run. The `bpftool` utility is run as `bpftool prog profile name collect_rapl_info duration 5 cycles instructions l1d_loads llc_misses` where `collect_rapl_info` is the name of the profiled *eBPF* program, `5` is the duration in seconds for which the `bpftool` utility is run, and `cycles`, `instructions`, `l1d_loads`, and `llc_misses` are the benchmarked statistics. The used statistics are based on the `bpftool` utility which only supports a limited number of statistics and can only measure four at a single time. In benchmarks *sysfs* and *perf-userspace* we only use the `perf` utility as the source of the results but in benchmark *perf-ebpf* we use both the `perf` and `bpftool` utilities because effectively there are two programs running during the benchmark. One in User-space and the other in Kernel-space.

Table 5.1 shows the results of the benchmarks. The *perf-ebpf* test case contains two rows of values where the top row contains the statistics for both the User-space and the Kernel-space (*eBPF*) programs and the bottom row contains the statistics for only the *eBPF* program. The results show that the most inefficient approach to high-frequency reading energy consumption via the *RAPL* interface is the `sysfs` approach. Of the two programs utilizing perf events the more efficient is the one using *eBPF* reading the perf event counters in Kernel-space.

Based on the results of the benchmarks we will use for the implementation of the energy consumption reading capabilities in Chapter 7 the *perf_ event* approach implemented using the *eBPF* technology.

Benchmark	cycles	instructions	L1-dcache-loads	LLC-load-misses
sysfs	3,795,412,368	3,069,819,731	777,323,949	56,504
perf-userspace	1,174,388,152	769,703,008	191,072,023	50,263
perf-ebpf	760,739,586	656,435,608	158,702,720	32,285
	49,995	370,994,417	52,538,065	8,491

Table 5.1: Results of the benchmarks the efficiency of the different approaches to reading energy consumption values via the *RAPL* interface.

5.2 System calls

System calls are an interface for services of the operating system made available to the user. The purpose of system calls varies greatly but they can be roughly classified into six major groups: *a)* process control, *b)* file manipulation, *c)* device manipulation, *d)* information maintenance, *e)* communications, and *f)* protection [51]. All programs, even if not directly,

make use of system calls. This makes them a good candidate for giving energy usage the context of why is energy being used at a certain point of time.

System calls can be traced in multiple ways. The commonly known approach uses the `ptrace()` system call. This approach is for example used in *strace*, a tool dedicated to tracing system calls. *strace* is a great tool but does not serve our purposes as we are also tracing other domains than system calls. Another approach can make use of *eBPF* by attaching an *eBPF* program to the `tracepoint/raw_syscalls/sys_{enter,exit}` tracepoints. The details of using *eBPF* for tracing system calls is in Chapter 7.

While tracing, we're interested in the system call ID, its entry and exit times, the cpu they're running on and their associated PID and TGID. We can use the entry and exit times for rating the single system calls using energy values (*RAPL*). Using the CPU number, PID and TGID we can further classify this information.

In this thesis we will only use the entry and exit time of system calls to calculate their energy score. But more detailed analysis of system calls in the context of energy consumption can lead to much more detailed and interesting results. We will describe on a high level some of these analysis methods.

5.2.1 I/O operation bundles

All I/O operations in an operating systems are carried out using system calls. System calls can be used to track single I/O operations but those operations often do not happen isolated. Instead, often I/O operations happen in bulk. Reading a big file, reading multiple files for the same purpose, writing to files in set intervals,.... The purpose of I/O operations differs in the number of involved files/streams, how often they happen, in what intervals and what is the size of read/written data. Detecting such *bundles* could help understand the energy behaviour of the profiled software. [44]

5.2.2 Asynchronous Power Behaviour

Apart from the use of system calls for I/O or process management, they are used for general interaction with all hardware components. And all hardware components have their own energy behaviour. And only a selected few of these components make their energy behaviour known to the kernel. And often they might not know the information themselves as they might not have a dedicated power measurement module. Among such components can be Wi-Fi modules, Bluetooth modules, GPS modules or fingerprint readers.

Pathak et al. [44] discuss how all the different hardware components often operate in different *modes*. Each of these *modes* causes the component to enter a different *power state* changing its energy consumption. For example, in a GPS module some of the different *modes* could be: *a)* idle *b)* locating satellites *c)* location found (low accuracy) *d)* location found (high accuracy).

Some modes of operation can be directly requested by the user (e.g., switching a Wi-Fi module on/off) but some modes are assumed without being directly requested (e.g., a spinning drive spinning after already reading a file). Pathak et al. [44] write about such modes as *tail state*. Such states can contribute in a significant way to the total energy consumption. System calls could be used to detect changes in modes of operation of these hardware components, track their current state in finite state machines and use this information to improve the accuracy of energy profiling.

Chapter 6

Analysis of Requirements

In this chapter we give a brief summary of the intended functionality of the designed energy profiler and we present a list of its formal functional and non-functional requirements. Some of the requirements were inspired by existing works [45, 36].

6.1 Analysis of Requirements

This work aims to design a novel energy profiler and implement in a suitable way for integration into Perun (see Chapter 3), a performance version system. Because Perun is implemented in Python, we want to use Python as the implementation language of the profiler as well. As for profiling, we want to make use of the *eBPF* technology for which the go-to language is C, so the core of the profiler will be implemented as a C shared library.

Energy consumption. Energy consumption is a difficult metric to measure accurately. Complete coverage of measuring energy consumption would require for voltage monitors to be installed in every part of hardware in a system. That is practically unfeasible in most systems. The *RAPL* (see Chapter 5.1) interface is currently the best combination of accuracy and practicality for measuring energy consumption of a system. To read the energy readings provided by the *RAPL* interface we will use *perf_ events* which are sampled in an *eBPF* program.

Runtime context. Without context, the measured energy consumption data will provide minimal informational value and thus additional *execution context* needs to be provided. For this purpose we chose to use system calls. To keep track of system calls executed by a process we will use two *eBPF* programs tracking the start and end of system calls. Using system calls as a context for energy consumption requires the adoption of a scoring algorithm. In Chapter 5.2 we discuss a number of methods for utilizing syscalls in the context of energy consumption. As the scope of this work is limited, we chose to implement a much more trivial scoring algorithm.

Visualizations. Since we are creating a novel energy profiler with complex results, we create will a reporting tool. The tool will be capable of creating text reports with brief summaries of the results accompanied by suitable visualizations providing a visual aid for understanding the results of the energy profiler.

6.1.1 Functional Requirements

A high-quality profiler needs to, in general, meet certain criteria and an energy profiler has its own set of unique requirements. We compiled a list of criteria that will guide our design process and implementation of the energy profiler.

- **FR_IU (Independent usability).** The energy profiler is usable even when used as a stand-alone solution.
- **FR_PI (Perun integration).** The energy profiler is integrated into Perun and extends its functionality as one of its collectors (see 3.2).
- **FR_BPF (eBPF technology).** The energy profiler leverages the *eBPF* technology to implement its profiling capabilities.
- **FR_RO (Runtime overhead).** The instrumentation overhead introduced by the energy profiler as minimal as possible to obtain results close to the original program performance.
- **FR_PPT (Post-processing time).** The processing of raw profiled data is fast to keep the user experience as fluent as possible.
- **FR_IT (Interpretation time).** The interpretation of the profiling results is made easy to allow swift iteration on the findings.
- **FR_EDS (Energy domain sampling)** The core premise of an energy profiler is that it samples energy consumption.
- **FR_EDC (Energy domain coverage).** Because energy can be consumed by a large number of components, the energy profiler covers as many of these components to achieve the highest possible accuracy in energy readings.
- **FR_EDSR (Energy domain sampling rate).** Sampling energy readings at a higher rate can raise the quality of the results.
- **FR_SCT (System call tracing).** System calls are used as providers of runtime context for the energy readings.
- **FR_SCS (System call scoring).** System calls need to be put into the energy consumption context using a scoring algorithm.
- **FR_RR (Result reproducibility).** The energy profiler produces the same, or at least highly similar, results across repeated profiling sessions.
- **FR_NS (Noise susceptibility).** While energy readings can be affected by a high number of factors, the profiler can leverage the runtime context to minimize the impact of the noise on the precision of the readings.
- **FR_VIZ (Visualization).** The energy profiler provides a tool for interpreting the generated profile data for further manual analysis.

6.1.2 Non-functional Requirements

Aside from the functional requirements, every software has additional non-functional requirements to keep the quality of the software at a reasonable level.

- **NFR_SS (Storage space).** The resulting profile data require minimal possible amount of storage as smaller files are generally faster to load and process.
- **NFR_MD (Minimal dependencies).** The energy profiler uses only a bare minimum of mandatory dependencies to lower the entry barrier for new users to use the energy profiler.
- **NFR_PCP (Profile Comprehensibility).** The energy profiler outputs profile data containing information about both the profiled system and the profiled process.
- **NFR_RCP (Reporting Comprehensibility).** The reporting tool provides comprehensive outputs with easy-to-understand names, legible formatting and easy-to-read graphs.
- **NFR_EU (Ease of use).** The energy profiler generates a single file containing all the necessary information for their interpretation. The reporting tool utilizes all this information and does not require anything else. Both tools can be parametrised but they provide sane defaults.
- **NFR_RP (Results portability).** The reporting tool generates the same results regardless of the runtime environment. All the information it needs are provided by the profile data and no machine-specific info should is used.

Chapter 7

Design and Implementation

This chapter introduces an implementation of the novel energy profiler called `sysrapl` using methods from Chapter 5 and using the *eBPF* technology from Chapter 4. At the beginning of the chapter we describe its implementation (Section 7.1). Next we describe the implementation of a reporting tool (Section 7.2) called `sysrapl-report` capable of creating text reports and visualizations. At the end of the chapter we discuss the performance (Section 7.1.3) and limitations (Section 7.3) of the created tooling.

7.1 Energy profiler

In this section we'll present the different parts of `sysrapl`: (1) *Engine*, and (2) *Post-processor*. *Engine* is responsible for the instrumentation, and *Post-processor* converts raw data generated by the *Engine* into a final profile in JSON format. A high-level diagram of the profiler can be seen in Figure 7.1.

For the implementation of the energy profiler were used two programming languages: Python and C. We chose Python because Perun (see Chapter 3) is implemented in Python and thus using Python will help us ultimately achieve the **FR_PI** requirement. The C language was chosen because it is basically the standard language for Linux system programming and the `libbpf` library (see Section 4.2.3) we use for making use of the *eBPF* instrumentation framework is implemented in the C language. The build system used for the project is *Meson*.

The profiler is implemented across these files:

- `sysrapl.py` containing the CLI, interface for the C shared library and the *post-processor* implementation,
- `sysrapl.c` implementing the User-space side of the profiler which handles the instrumentation of *eBPF* programs,
- `sysrapl.h` defining data types and structures used in the project, and,
- `sysrapl.bpf.c` implementing the different instrumented *eBPF* programs for tracing/sampling.

In this thesis we focus primarily on the **FR_IU** requirement rather than the **FR_PI** requirement. We still structured the top-level of the implementation of `sysrapl` to mirror the structure of Perun's collector (see Section 3.2) to make future integration in Perun more straightforward.

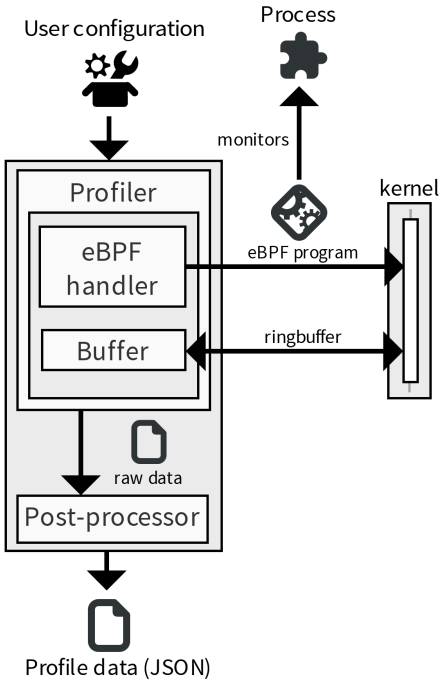


Figure 7.1: Schema of the *sysrapl* profiler. The runtime of the profiler has two phases: profiling and post-processing. The profiling phase instruments *eBPF* programs into the kernel generating raw data. The post-processor then processes the data into the final profile.

7.1.1 Engine

The *Engine* is in charge of collecting raw performance data by instrumenting *eBPF* programs. In the model of a Perun *collector* it implements the `collect()` function.

Because it is implemented in C, it needs to be made somehow available in the Python script (`sysrapl.py`). Python does support FFI for using shared libraries. There are multiple modules providing complex and robust capabilities, but in line with the **NFR_MD** requirement, we chose to use the built-in `ctypes` module [3]. The module is ideal for simple use-cases where minimum number of functions is called and a small number of structures from the shared library is used as introspection of the loaded library is very limited. However, it has the major downside of requiring to know the absolute path to the shared library. We overcame this limitation in a robust manner by making it mandatory to use the build system to both build `sysrapl` and to install it.

The *Engine*, again in line with the **NFR_MD** requirement, only requires the standard C library and the `libbpf` at runtime. The *Engine* provides a single function `sysrapl_profile()` which serves as the entry-point for profiling. It takes as a parameter a structure used for configuring the behaviour of the *Engine* (see Listing 7.1).

```
struct sysrapl_profile_opts {
    char *output_file; // path to a file for outputing raw profiling data
    int delay; // delay before starting attaching the instrumented eBPF programs
    int filter_pid; // PID of the profiled process
    int frequency; // Frequency at which energy readings are sampled
};
```

Listing 7.1: Structure for configuring the behaviour of *Engine*

The *Engine* implements in total three *eBPF* profiling programs:

a) `sys_enter()` is attached to the **tracepoint/raw_syscalls/sys_enter** tracepoint. It marks the start of a system call by using the current TID as a key in a hashtable (map `syscall_start`) where the value is the time when the system call started (in nanoseconds). The program only tracks a system call when the current PID equals to the profiled PID.

b) `sys_exit()` is attached to the **tracepoint/raw_syscalls/sys_exit** tracepoint. It marks the end of a system call by checking whether in the map `syscall_start` is the current TID. If yes, the program reserves a chunk of memory in the Ringbuffer big enough to hold the `sysrapl_data_t` type. In the chunk of memory is written the id of the system call, its entry and exit times (in nanoseconds), the current CPU and the current TID. Then the chunk of memory is submitted to the Ringbuffer.

c) `collect_rapl_info()` is attached to a **perf event**. This is a special case which requires the program to be manually attached to a perf event during the setup of the *eBPF* program. This program is intended to be attached to the `cpu-clock` software event, which is always available. The software event can be configured with a frequency, which in `sysrapl` is by default 99Hz, and can be configured using the CLI. This fulfils the **FR_RDS** requirement. Every time this event is triggered, the *eBPF* is run and all enabled *RAPL* perf events are sampled. And because all of these events are CPU-independent, each needs to be sampled just once. All the sampled values are then put into a memory buffer reserved in and then submitted to the Ringbuffer.

The implementation of the two *eBPF* programs: `sys_enter()` and `sys_exit()` fulfils the **FR_SCT** requirement. The implementation of the `collect_rapl_info()` fulfils the **FR_EDS** requirement.

The *Engine*'s `sysrapl_profile()` function operates in phases which are tied to the lifetime of *eBPF* programs as bootstrapped by *libbpf* (see Section 4.2.3):

1. **Open phase.** The first step of the phase is to open a temporary file under `/tmp` directory to contain the raw profiling data.

   Most Linux distributions mount */tmp* as a Tmpfs, making the *Engine* write the raw profile data into operating memory. This is desired due to I/O on storage devices being generally more expensive.

   The next step is to „open“ the group of *eBPF* programs and start initializing the read-only section of the programs. The read-only section of the created programs contains:

   - PID of the profiled process
   - Booleans for controlling which RAPL domains to sample
   - Maps (see Section 4.1) for:
     - file descriptors of *perf_event* for every sampled RAPL domain (one map per file descriptor),
     - tracking of running system calls,

– Ringbuffer for exchanging data between Kernel-space and User-space

Because not every machine supports all of the *RAPL* domains, a check needs to be done, to only open the perf events that exist. This mechanism helps to fulfil the **FR_EDC** requirement. And because the *RAPL* perf events are PMUs, their *type* value and *config* value (see Section 5.1), used when „opening“ the perf events, can vary across machines, so during the check these values are collected. The values for *RAPL* perf events can be found in Sysfs under `/sys/devices/power`. The present events are then opened and marked as usable using the booleans in the read-only section. The events are not enabled and are not put yet into the maps, as the programs are not loaded in-memory, yet.

2. **Load phase.** Following the load of *eBPF* programs the maps of the sampled *RAPL* perf events are populated with file descriptors of the present perf events. A Ringbuffer is also created and assigned to the prepared map in the *eBPF* programs. The ringbuffer is used for sending data from the *eBPF* programs running in Kernel-space to the *Engine* running in User-space During the creation of the Ringbuffer a function is assigned to it to handle data arriving from Kernel-space during polling. The function is `handle_ring_buffer()` and it outputs the received data in the temporary file opened during the *Open phase*.

   To make the `collect_rapl_info()` program work, it needs to be attached to the `cpu-clock` perf event. A helper function `attach_perf_event_sampling()` opens the `cpu-clock` perf event and attaches the program to it.

   If the profiler is configured to wait for a period of time before the start of profiling, the profiler waits in this phase. This capability exists to help mitigate the effect of noise caused by starting the tool or other software.

   Right before the next phase, the *RAPL* perf events are reset, to ensure proper initial values in the counters, and are enabled.

3. **Attachment phase.** Once an *eBPF* program is attached, it starts functioning right away and can no longer be configured. In this phase the *Engine* enters a poll loop where it waits for data to arrive via the Ringbuffer. When data arrive, the poll function triggers a handler function, assigned to the Ringbuffer in the *Load phase*.

   The polling continues until the user sends to the profiler the *SIGINT* signal.

4. **Teardown phase.** In this phase the different resources, including the *eBPF* programs are cleaned up. The only resource left intact is the temporary file holding raw profiled data.

7.1.2 Post-processor

The *Post-processor* is in charge of transforming the raw profile data into a readable format, which can then be used for further analysis.

The *Post-processor* operates in two stages:

1. **Processing of energy reading samples.** In the raw data the energy reading samples contain values present at the time in the counters of the perf events. To make sense of these values the *Post-processor* needs to scale these values using the scale

assigned to every single perf event (see Section 5.1) to convert them to proper units (usually Joules). After the conversion the values represent the energy consumed since the last overflow of the counter. In further analysis we're more interested in how much energy was consumed during each sampled time window. So, two neighbouring samples are used to calculate the difference between them and the result is assigned to a time window.

2. **Processing of system calls.** After the previous stage the energy consumption of different time windows hence computed. This stage uses this data to annotate energy consumption scores to every single system call.

   To fulfil the **FR_SCS** requirement this stage implements a scoring algorithm but as explained in Section 6.1 the implementation is intentionally trivial. The steps of the algorithm are:

   1) Select a system call from the raw data.
   2) Set the system call's energy score to 0.
   3) Find all time windows overlapping the system call run time.
   4) Calculate the ratio of the runtime of the system call in the first time window.
   5) Multiply the time window's energy readings by the ratio.
   6) Add the energy reading of the time window to the system call energy score.
   7) Remove the first time window from the list of windows.
   8) If there is another time window, go to step 4.

After the end of the two stages, the post-processor outputs the processed profile data in the JSON format. By default `sysrapl` outputs the profile to `stdout` but it can be configured to save it into a file. The advantage of this approach is that it can be used in pipes with other UNIX utilities.

Profile data format

In previous sections we've mentioned that `sysrapl` outputs the final profile data in the JSON format. In this section we'll describe what information are put into the profile data and what is its structure.

Machine information. Energy profiling varies largely between different hardware configurations, specific hardware models and what software runs on the hardware. In the profile data `sysrapl` includes, alongside the information about the profiled process, the time when it was profiled, the duration of the profiling session (in nanoseconds), and information about the machine on which `sysrapl` ran. The system information are only high-level and created by combining the output of `platform.uname()` and the content of `/proc/cpuinfo`. Including this data satisfies the **NFR_PCP** requirement. Listing 7.2 shows a snippet from the data JSON with a section containing this information.

```
{
    "info": {
        "process-name": "gnome-shell",
        "pid": 1598,
```

```
        "time": "2023-04-26T09:03:46",
        "duration": 30215400875,
        "frequency": 99,
        "host-info": {
            "system": "Linux",
            "node": "localhost",
            "release": "6.2.9-300.fc38.x86_64",
            "version": "#1 SMP PREEMPT_DYNAMIC Thu Mar 30 22:32:58 UTC 2023",
            "machine": "x86_64",
            "processor": "Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz"
        }
    },
    ...
}
```

Listing 7.2: *info* section of the final profile data format containing information about the profiled software and the host system

Energy consumption samples. This section of the data JSON contains all data related to system-wide energy consumption: a) the unit and the scale of the energy values, b) sampled energy domains, c) total consumption across all domains (corresponding to those described in Section 5.1), and, d) a list of all sampled time windows with energy consumed (using the unit and the scale in the profile). Listing 7.3 shows a snippet from the data JSON with a section containing this information.

```
{
    ...
    "rapl": {
        "unit": "Joules",
        "scale": 0.23283064365386963,
        "domains": ["cores", "pkg", "ram", "gpu", "psys"],
        "total": {
            "time": 30151676016,
            "cores": 5413635254,
            "pkg": 28118835449,
            "ram": 24002197265,
            "gpu": 869750976,
            "psys": 115713439942
        },
        "events": [{
            "time": 921404653026,
            "cores": 9521485,
            "pkg": 22277832,
            "ram": 11596679,
            "gpu": 1647949,
            "psys": 72692871
        },...]
    },
    ...
```

```
}
```

Listing 7.3: *rapl* section of the final profile data format containing system-wide energy consumption data

System calls. This section contains a list of all system calls ordered by their exit time. Each item in the list contains: a) the system call ID, b) the system call name, c) the entry time (in nanoseconds), d) the exit time (in nanoseconds), and, e) the energy consumption data (using the unit and the scale in the profile). Listing 7.4 shows a snippet from the data JSON with a section containing this information.

```
{
    ...
    "syscalls": [{
        "id": 39,
        "name": "getpid",
        "cpu": 2,
        "tid": 2779,
        "entry_time": 1202281290743,
        "exit_time": 1202281292402,
        "cores": 2796.828618969,
        "pkg": 5082.4088385840005,
        "ram": 3007.342549827,
        "gpu": 120.293721702,
        "psys": 18204.446818404
    }...]
}
```

Listing 7.4: *syscalls* section of the final profile data format containing a list of all the traced system calls

7.1.3 Performance

In this section we want to showcase the performance of `sysrapl` on a synthetic benchmark to see how big of an overhead does the energy profiler introduce. As the benchmark we used an open-source TCP server capable of handling simple echo messages implemented using `io_uring` [25] coupled with an open-source TCP client application for testing such simple servers [26]. In the benchmarks the provider of the results is the client application which at the end of its runtime prints statistics collected during its runtime, where the statistics show the number of requests sent and responses received.

For this benchmark we use the same testing environment as in Chapter 8. In the benchmarks the server is run as `io_uring_echo_server 8080` where `8080` is the port the server listens on, and the client is run as `echo_bench -a 127.0.0.1:8080 -t 10 -c 4`, where `127.0.0.1:8080` is the address and the port the client application connects to, `10` is the number of seconds for which the application is run and `4` is the number of clients the applications spawns in separate threads. In total we will conduct four different benchmarks:

a) clean, where the server is not profiled. This benchmark will be used as the baseline.

b) **sysrapl-default**, where the server is profiled using `sysrapl` with default parameters (*RAPL* value sampling at 99Hz).

c) **sysrapl-highfreq**, where the server is profiled using `sysrapl` sampling *RAPL* values at 9999Hz.

d) **strace**, where the server is profiled using the `strace` tool. `strace` is known for introducing relatively high overhead to the profiled software and will serve as additional context for this benchmark. In this benchmark the output of the `strace` tool is redirected to `/dev/null` to reduce the effect of printing text into the terminal window on the benchmark.

Table 7.1 shows the results of the different benchmarks where the results are the average of three sets of results for every benchmark. The results show that in applications with highly frequent system calls in both benchmarks *sysrapl-default* and *sysrapl-highfreq* `sysrapl` degrades the performance by about 20%. The small difference between *Performance ratio* of the *sysrapl-default* and *sysrapl-highfreq* benchmarks shows that the majority of the performance overhead comes from tracing system calls and not from sampling *RAPL* energy readings, making our method of sampling of the events quite efficient. The *strace* benchmarks confirms the general believe that the `strace` tool does introduce high overhead to the profiled software. The method of tracing system calls in `sysrapl` compared to the method of tracing in `strace` shows significantly better results and reaffirms our believe in the efficiency of the *eBPF* instrumentation framework.

Benchmark	Requests [req/sec]	Responses [res/sec]	Performance ratio [%]
clean	78,780	78,780	100.0%
sysrapl-default	65,249	65,249	82.8%
sysrapl-highfreq	62,790	62,790	79.7%
strace	29,316	29,316	37.2%

Table 7.1: Results of the benchmarks showing the runtime overhead of `sysrapl` with different configurations and strace.

7.2 Reporter

In this section, we'll describe the implementation and capabilities of a reporting tool called `sysrapl-report`. Statistical analysis and generation of graphs is commonly done using languages like R or Python. As stated in Section 7.1, Perun is written in Python, so using it for the implementation of `sysrapl-report` is a natural choice. In `sysrapl-report` we make use of a number of modules which are well-maintained, widely used, and already used by Perun. For data processing we used *Pandas* [35, 52], and for plotting *Seaborn* [56] (which uses *matplotlib* [27]).

`sysrapl-report` is implemented as a single `sysrapl-report.py` script. By default it outputs its results into a directory named based on the data from the interpreted profile file. `sysrapl-report` takes as its input a single profile data JSON file and it only uses this file to produce its results. This is to fulfil the **NFR_RP** requirement.

The number of system calls that a process can invoke in a matter of seconds can easily reach thousand digits. `sysrapl-report` mainly cares about the *energy scores* of the different

system calls whose values can differ largely. Presence of outliers is commonplace which can significantly affect the final outputs of `sysrapl-report`. To detect outliers we use the *interquartile range* (IQR) method which we then filter out from the datasets. The method separates the data into four quartiles (Q_1, Q_2, Q_3, Q_4) where the first quartile (Q_1) and the third quartile (Q_3) are used to calculate the IQR. Then the IQR value is multiplied by 1.5 and subtracted from Q_1 and added to Q_3. Then all values below $Q_1 - 1.5 * IQR$ and above $Q_3 + 1.5 * IQR$ are considered outliers.

7.2.1 Text report

Visualizations and graphs are a great way to convey complex data. For simple data a text report is often more suitable for its simplicity. `sysrapl-report` outputs a brief text report for an interpreted profile in the output directory in a file called `report.txt`.

A concrete example of the text report can be seen in Appendix A.

The text report contains the following sections:

1. **General information.** Contains the name and PID of the profiled process, time of profiling, its duration and the sample rate of the *RAPL* energy domains.
2. **System information.** Because energy profiling is hardware specific, a section about the profiled system is included based on JSON section described in Section 7.1.2.
3. **Ratio of energy consumption.** By mapping energy consumption to system calls it can be possible to estimate what percentage of the system energy consumption can be attributed to the profiled process.
4. **Top consuming system calls.** Shows the top five contributors among system calls to the process's energy consumption. The number of shown top contributors can be configured.

7.2.2 Visualizations

Every kind of data can be visualised using a different method. `sysrapl` does generate heterogeneous data and thus different visualizations are needed for accurate assessment of its findings. In `sysrapl-report` we implemented a number of graphs which we categorise into two groups (due to two categories of data being provided by `sysrapl`):

a) **Raw energy consumption** (Section 7.2.2), and,

b) **System calls as energy consumers** (Section 7.2.2).

We designed these visualizations to be easy to understand to fulfil the **NFR_CP** requirement.

The following images have been cropped to better show the form of the different graphs. The exact readings are not relevant in this section. An example report with full forms of the graphs can be seen in Appendix A.

Raw energy consumption

The first data category provided by `sysrapl` includes the raw energy consumption readings. The readings serve as a basis for further analysis but even their raw form provides interesting insights. The energy readings are separated into domains explained in Section 5.1.

Consumption at a certain point of time. The most basic graph plots the sampled energy readings in time using line plot. It shows the energy behaviour of the whole system and serves as a good entry point for getting a basic understanding of the system's energy behaviour. But due to no analysis being involved, it is highly susceptible to noise.

The X axis represents time and the Y axis represents the consumed energy.

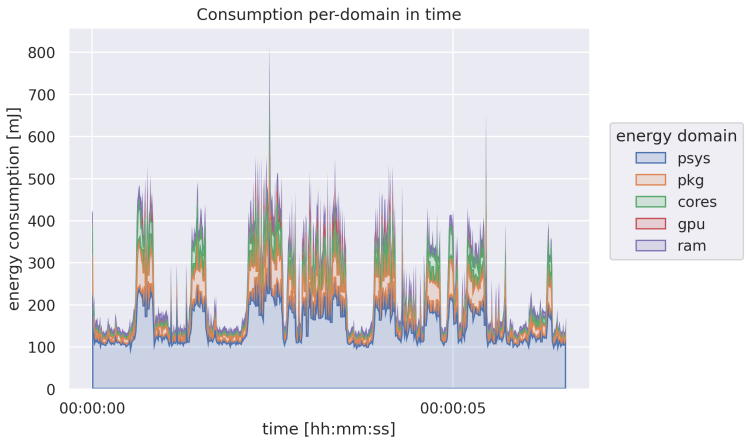


Figure 7.2: Line plot of per-energy domain energy consumption in time

Cumulative consumption An alternative representation of the above graph with the difference of showing the energy readings in a cumulative manner.

The X axis represents time and the Y axis represents the consumed energy.

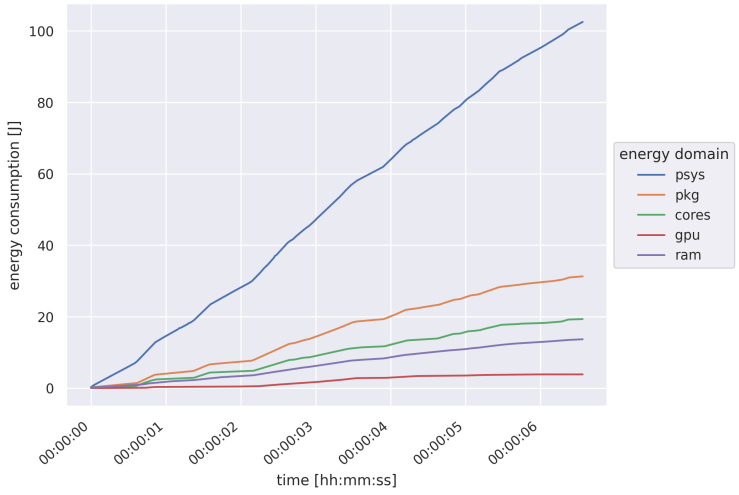


Figure 7.3: Line plot of cumulative per-energy domain energy consumption in time

System calls as energy consumers

The second data category contains all system calls called by the profiled process. Using the raw energy consumption data every single system call has been given an *energy score*. This process helps to reduce noise significantly. The scoring algorithm is described in Section 7.1.2.

Energy consumption in time. Similarly to the graph in Section 7.2.2, the most basic plot is showing the energy consumption of system calls in time. But this data category is more challenging due to higher number of data categories and uneven number of data points. Every type of system call can have up to five *energy scores* and by default `sysrapl-report` reports more than one system call. For displaying high number of samples across many domains we chose two types of graphs: *a) heatmap*, and, *b) waterfall graph*.

a) **Heatmap**

Plotting values in time using heatmaps provides a great way of spotting exact points of time where a significant chunk of energy was consumed. To allow seeing the maximums reliably the dataset extremes are used as the border values. The graph has its limits as the number of plotted samples needs to be capped at a low enough number to prevent the cells from becoming too small to perceive their colour. In this work we achieve this by downsampling the data points and summing them up. The sample width is adjustable.

The X axis represents time, on the Y axis are different (*system call*, *energy domain*) combinations and the colour of the cells represents the consumed energy. The brighter the colour, the more energy was consumed by given pair at given time.

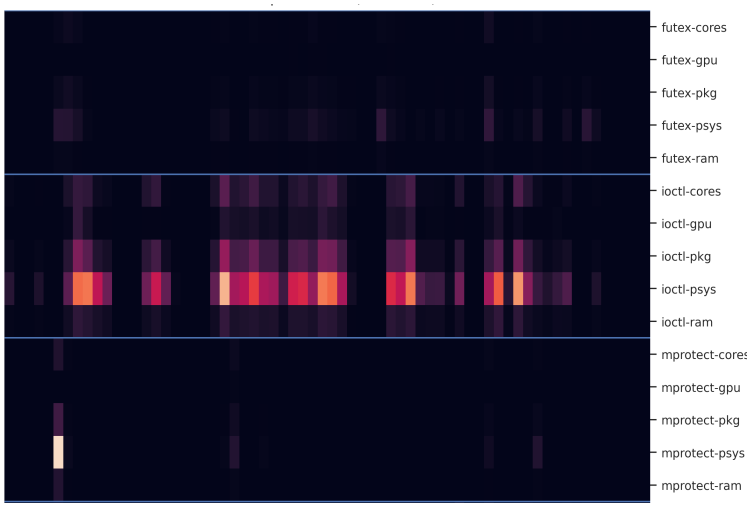


Figure 7.4: Heatmap of per-energy domain system call energy consumption in time

b) **Waterfall graph**

The graph is based on Brendan Gregg's *frequency trails* [16] which he designed to show the distribution of sampled data across many domains. In this work we adopted *frequency trails* into a so called *waterfall graph* displaying on a normalized scale the energy consumption of system calls in time. Similarly to the heatmap, every plot

represents a different (*system call*, *energy domain*) combinations. These combinations are sorted from the most expensive ones to the least expensive ones. This allows at a single glance to see both the highest and smallest consumers. Unlike the heatmap, the waterfall graph does not require the data points to be downsampled to fit into the graph. Still the data points are downsampled and then upsampled again to achieve a smoothing effect in the graph.

The X axis represents time and the Y axis represents the consumed energy on a normalised scale.

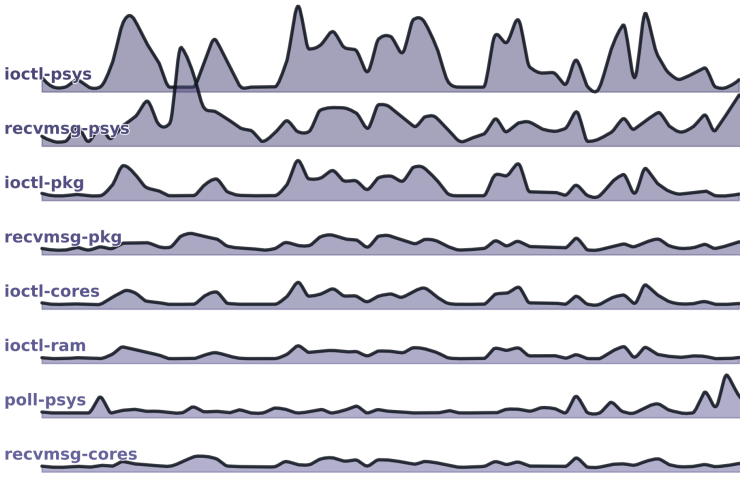


Figure 7.5: Waterfall graph of per-energy domain system call energy consumption in time

Cumulative consumption in time. The *tsunami graph* is a variant of the *waterfall graph* (*b*) where instead of plotting the values at a certain point of time we plot the cumulative values, i.e. the sum of all previous values. This allows us to show (*system call*, *energy domain*) combinations with the highest energy consumptions.

The X axis represents time and the Y axis represents the consumed energy on a normalised scale.

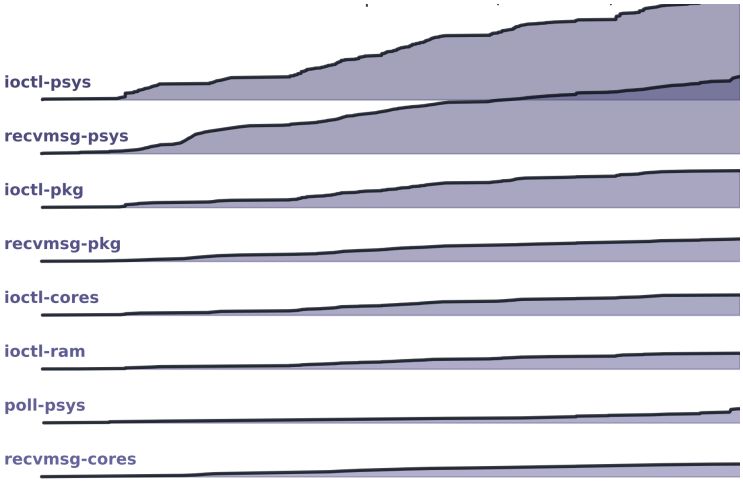


Figure 7.6: Tsunami graph of cumulative per-energy domain system call energy consumption

Distribution of energy consumption. Different system calls serve for different purposes and thus their energy behaviour can largely differ. Using a box plot we can show the distribution of the consumed energy.

The X axis shows different system calls across energy domains and the Y axis shows the energy consumption. The box represents three quartiles (first, median and third), the whiskers represent maximum and minimum. Values outside of the whiskers are outliers.

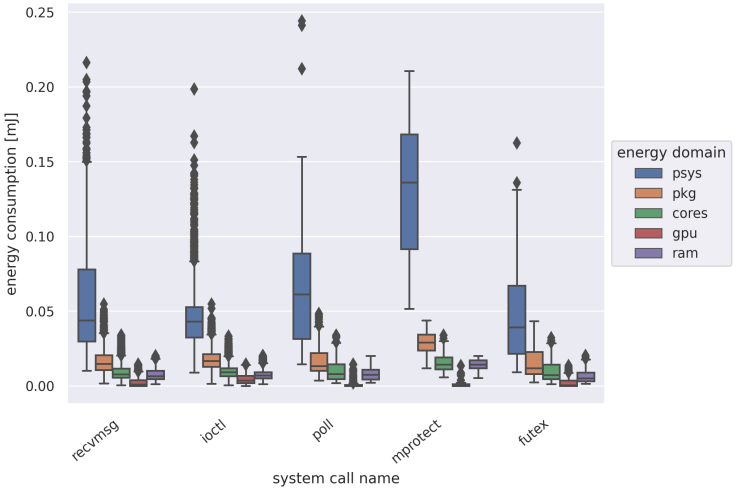


Figure 7.7: Box plot showing distribution of energy consumed by system calls across different energy domains

7.3 Limitations

Use of phases. During development we've focused mainly on the **FR_RO** requirement which lead us to design `sysrapl` in phases. This decision in some cases causes problems when the post-processing time (**FR_PPT**) becomes so high, that the use of the tool is no longer practical. Designing the post-processor capable of processing the raw data continuously and in batches could be viable compromise.

Number of profiled processes. At the moment `sysrapl` can profile only a single process. Removing this limitation could help with the **FR_NS** requirement because the energy consumption could be split between the different processes. With such a change the scoring algorithm (**FR_SCS**) would need to be adjusted as well.

Support for `perf_event` groups in *eBPF* programs. At the moment *eBPF* programs can read counter values of perf events only one at a time while User-space applications may make use of the capability to group different perf events and read their values in bulk. Having this capability available for *eBPF* programs could provide some performance boost.

Speed. Despite the **FR_IT** requirement for `sysrapl-report`, the average runtime of the tool is $\tilde{3}5$ seconds for a profile with >11000 sampled system calls on a Lenovo ThinkPad T460 laptop with Intel Core i5-6300U processor (used during experiments in Chapter 8). We believe this is due to *matplotlib* being a single-threaded CPU-bound module which does not scale well with multiple plots with a high number of samples. Use of plotting libraries

capable of utilizing multiple threads or GPU capabilities would significantly reduce the average runtime of the tool.

Separation of output. During the design of `sysrapl-report` we've focused greatly on the **NFR_RCP** requirement. We believe we met this requirement and can only further raise the bar. At the moment `sysrapl-report` outputs the text report and graphs as separate files. Bundling these into a single report would with interpreting the results and also archiving the results as only a single file would be needed.

Chapter 8

Experiments

This chapter will demonstrate the use of `sysrapl` and `sysrapl-report` (introduced in Chapter 7) in a series of experiments. The goal of these experiments is not only to show the capabilities of the created tooling but also to show its behaviour under different conditions.

8.1 Methodology

Instead of profiling a large number of projects across several experiments, we chose to profile a single project — GNOME Shell [12] — under different runtime conditions. Because `sysrapl` is an energy profiler, the runtime conditions can affect the results and we will hopefully see a high value in showing those effects.

GNOME Shell is a Graphical shell using Mutter [13] as a Display server. The fact that it is a non-trivial project with complex behaviour and that it makes even use of all basic hardware components (CPU, RAM, GPU) makes it a good choice for our experiments.

For the runtime of the experiments we define three variables which we use to create a matrix of test cases. The variables and their possible values are:

- **System installation.** The state of an operating system differs from user to user. Most users don't have complete control over their system's configuration. Moreover, this can be a potential source of energy consumption anomalies in the form of, e.g., software running continuously in background or software assuming a different mode of operation.

  - ***existing*** represents an existing Fedora Workstation 38 system with several years worth of upgrades, personalized selection of installed software and possible fine-tunings of the behaviour of the system. This environment is far from the ideal environment for continuous testing.
  - ***live*** represents a system as booted from a Live ISO on a USB flash drive. The booted system is Fedora Workstation 38. This environment is close to the ideal environment for continuous testing.

- **Workload.** The most visible cause of energy consumption is the software users run on their system. Peripheral devices like Bluetooth modules or Wi-Fi modules can also contribute significantly to the system's energy consumption. In the experiments we want to see how different workloads contribute to the system's consumption.

In both workloads a single terminal (GNOME Terminal) window is open and it is used to start the test session.

  - ***clean*** does not run any graphical applications with only the default set of background services running. The laptop is in *air-plane* mode (Wi-Fi and Bluetooth modules are disabled). To ensure the cleanliness of the environment the profiling was performed moments after the system's boot and login into the graphical session.
  - ***busy*** has a series of applications running including: (1) calendar app (GNOME Calendar), (2) maps application (GNOME Maps), and, (3) Firefox with 5+ tabs open and loaded, where one plays a video on YouTube in 720p quality. Both Wi-Fi and Bluetooth modules are enabled and the laptop is connected to a 5GHz access point.

- **Power mode.** Modern hardware devices (i.e., CPU or GPU) do not have a single mode of operation. The drivers of these devices often take into consideration external factors like temperature and energy consumption (e.g., using interfaces like *RAPL*), or configuration preference. The configuration preference is often presented to users in the form of *power modes*.

  Fedora Workstation uses the *power-profiles-daemon* [40] project since version 35 [41] to provide a way to set the system's mode of operation. The project (based on the used performance scaling driver) sets used scaling governors or energy biases. The project understands three power modes: a) ***balanced***, b) ***powersave***, and, c) ***performance*** (needs to be supported by the scaling driver).

All experiments share a single testing scenario. It involves basic interaction with the graphical shell simulating use by a user. To ensure consistency between the different test cases, we automated the testing scenario by implementing a test script. The script is written in the Bash scripting language. At the beginning it starts the `sysrapl` profiler and then start executing the test scenario where the steps are:

1. Open overview (press `Start`).
2. Open application menu (press `Start+A`).
3. Close overview (press `Start`).
4. Switch to workspace on the right (press `Ctrl+Alt+Right`).
5. Switch to workspace on the left (press `Ctrl+Alt+Left`).

Between every step we delay for 5 seconds. The keystrokes are send to the shell using the `ydotool` [42] tool. The tool requires a Daemon to be running[1]. The testing script needs to be run with root privileges as both `sysrapl` and the `ydotool` daemon require them.

During the testing we used an external power meter (Elektrobock EMF-1) for correlating the measured values with the values measured by `sysrapl` using the *RAPL* interface. The power meter has a display updating the power readings every second with values with one decimal place. The power meter was configured to display current power consumption in

[1] Which the test script ensures it does

Watts. Because the power meter does not offer a way to count the total energy consumed, we captured every experiment on a video and retrospectively summed the measured values into Joules.

8.2 Testing environment

All experiments were carried out on a single reference laptop:

Model	`Lenovo ThinkPad T460`
Arch	`x86_64`
CPU	`Intel Core i5-6300U CPU @ 2.40GHz`
GPU	`Intel HD Graphics 520`
RAM	`16GB DDR3 @ 1600MHz`
OS	`Fedora Workstation 38`
Kernel	`6.2.12`
GNOME Shell	`44.0`
Mutter	`44.0`

The laptop contains two battery packs: a) internal, and, b) external. During the experiments the external battery pack was removed and the laptop was plugged into the power meter which was plugged into a wall socket.

8.3 Experiments

Table 8.1 shows all test cases that were conducted as part of our experiment.

Test case	System installation	Workload	Power mode
ECB	existing	clean	balanced
ECS	existing	clean	powersave
ECP	existing	clean	performance
EBB	existing	busy	balanced
EBS	existing	busy	powersave
EBP	existing	busy	performance
LCB	live	clean	balanced
LCS	live	clean	powersave
LCP	live	clean	performance
LBB	live	busy	balanced
LBS	live	busy	powersave
LBP	live	busy	performance

Table 8.1: Test cases with specified runtime variables defined in Section 8.1

Table 8.2 shows all test cases with the total energy consumed sorted in descending order based on the total consumed energy. The order of the test cases shows that, as expected, the highest ranking test cases are in general the ones with *busy* workloads. These two groups are then ordered generally by the power mode with *performance* at the top, followed by *balanced* and *powersave*. The system installation has the lowest impact on the overall rankings but the results show significant difference in energy consumption in test cases with different

system installation variable but with all other variables equal (e.g, test cases EBP and LBP, or test cases EBB and LBB). Test cases EBS and LBS are an exception to this phenomenon which makes us believe that multiple runs of the same experiments are required to fully exaplain this phenomenon.

The energy consumption of the whole laptop measured by the hardware monitor (see Section 8.1) is uniformly higher than the values measured in the *psys* energy domain, which reflects the energy consumption of the whole SoC, by about 100 Joules. This suggests that while the SoC contributes the most to the reference system's energy consumption, there are other components that also contribute significantly. Such components possibly are the display, speakers, WiFi module, Bluetooth module and more. If we were to order the test cases in Table 8.2 in descending order based on the *hardware monitor* values, we would get almost the same order with the exception of the last four test cases. This suggest that the majority of variance in energy consumption across the different test cases comes from the SoC rather than other hardware components in the reference system.

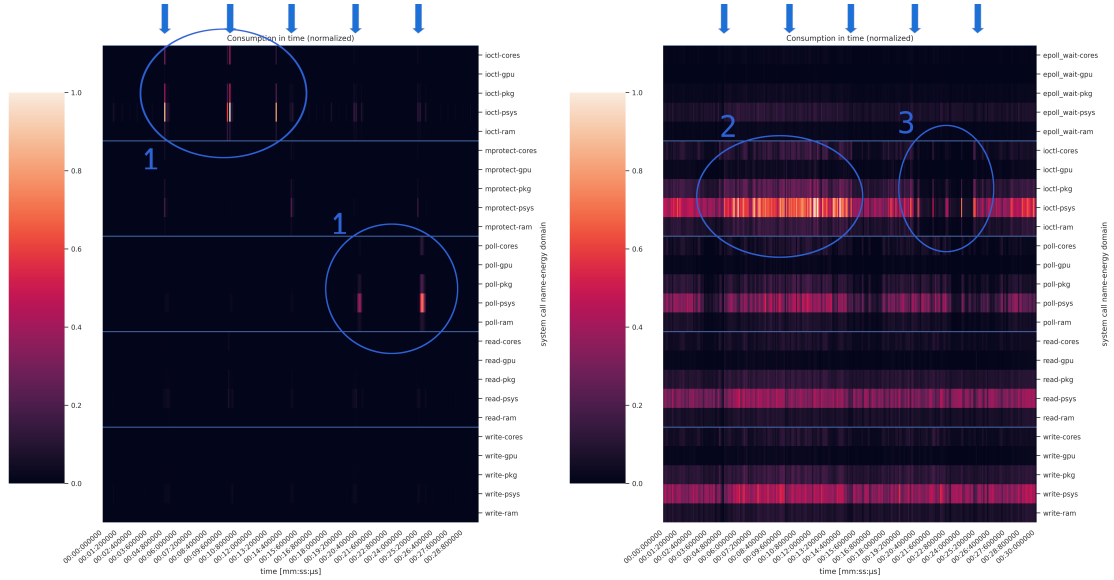
Test case	**psys [J]**	pkg [J]	cores [J]	ram [J]	gpu [J]	**hardware monitor [J]**
EBP	**496.551**	229.442	162.999	71.836	27.852	**600.0**
LBO	**439.894**	184.968	134.792	57.929	14.889	**542.4**
EBB	**353.226**	120.749	62.753	72.870	19.238	**443.6**
LBB	**335.378**	101.770	52.966	59.050	12.379	**415.8**
LBS	**292.539**	72.099	26.069	55.302	9.734	**379.1**
EBS	**283.364**	70.898	22.761	63.006	10.913	**357.3**
ECP	**201.514**	60.171	35.206	27.641	0.855	**304.9**
ECB	**185.740**	43.225	18,342	26.384	0.684	**285.5**
ECS	**167.512**	30.951	6.989	24.457	0.439	**266.7**
LCP	**166.271**	35.657	12.077	24.610	0.895	**284.5**
LCB	**155.632**	28.195	4.929	23.971	0.839	**272.0**
LCS	**150.826**	25.794	3.050	22.983	0.623	**267.6**

Table 8.2: Total energy consumed during test cases. The test cases are sorted in descending order based on the *psys* energy domain

Figure 8.1 shows two heatmaps from two experiments showing clear difference in the energy profile of the test cases.

In Figure 8.1a most of the „hot spots“ in energy consumption across all system calls are at the times when the different steps (marked by arrows) in the test scenario are carried out. Areas marked by number 1 in Figure 8.1a highlight how the steps working with the „Overview“ and „Application Menu“ show „hot spots“ mainly for the `ioctl` system call (i.e., input/output control; for configuring (mainly) hardware devices) while the steps switching workspaces show „hot spots“ mainly for the `poll` system call (i.e., for waiting for file descriptor to perform I/O operations). This suggests a difference in workloads and the capability of `sysrapl` to detect these differences.

But in Figure 8.1b the test scenario steps are not as clear. That suggest that Compositing windows is a significantly more expensive activity than the short-lived actions of interacting with the shell. In the figure there are two notable areas. The first one is marked by number 2 where the energy consumption of the `ioctl` system call has increased. In this case the Shell renders additional graphical elements while compositing scaled windows on the screen leading to probable higher energy consumption. In the are marked by number 3 the energy



(a) Test case ECB

(b) Test case EBB

Figure 8.1: Heatmaps from test cases **ECB** and **EBB** showing energy consumption of system calls across energy domains in time with highlighted steps of the test scenario and highlighted noteworthy areas in the heatmaps.

consumption of the `ioctl` system call was lowered significantly. We believe this is due to an optimization in Shell where an invisible window (Firefox playing a YouTube video), hidden after switching a workspace, is no longer composited as it is no longer visible on the screen.

Table 8.3 shows the top consuming system calls across all test cases. The results show that while there is some small variation in the ranking of the system calls, the lists always contain the *ioctl*, *poll*, *read* and *write* system calls. Their presence is expected as they are all used for I/O operations which Shell does use a lot in order to draw content on the display.

The results also show clear difference in energy consumption between test cases with the *clean* and *busy* workloads. This shows the capability of the profiler to recognize differences in workloads.

From the results it is apparent that the scoring algorithm described in Section 7.1.2 is indeed quite trivial and large number of system calls and the length of their runtime affect the energy score significantly. This is very noticeable in test cases EBB, EBS, and EBP where the total consumption of the different system calls is the highest for the *powersave* performance mode.

In test cases with *busy* workload it is unclear what is the extent of the effect of the noise (i.e., Firefox window playing a YouTube video) on the resulting energy scores of system calls. A mechanism for filtering out noise could increase the quality of the results. But for the purposes of this thesis the results are adequate because the scores managed to properly locate spots in the profiles with higher/lower energy consumption.

Test cases	ECB	ECS	ECP	EBB	EBS	EBP
Top 5 consuming system calls	ioctl	poll	poll	ioctl	ioctl	ioctl
	poll	ioctl	ioctl	write	write	poll
	read	read	read	read	read	write
	mprotect	write	write	poll	poll	read
	write	getpid	getpid	epoll_wait	epoll_wait	recvmsg
Top 5 consuming system calls values (*psys*) [mJ]	100.641	91.593	73.750	831.667	1,137.011	687.751
	70.060	72.700	28.805	694.437	869.479	541.777
	19.717	28.861	12.380	607.970	828.454	495.398
	14.982	28.364	10.272	482.460	463.981	426.071
	14.554	11.331	5.480	152.292	197.497	124.903

Test cases	LCB	LCS	LCP	LBB	LBS	LBP
Top 5 consuming system calls	poll	poll	poll	write	write	poll
	ioctl	ioctl	ioctl	read	read	write
	read	read	read	poll	ioctl	ioctl
	write	write	write	ioctl	poll	read
	mprotect	getpid	getpid	epoll_wait	epoll_wait	recvmsg
Top 5 consuming system calls values [mJ]	58.396	88.072	83.324	700.995	902.970	463.572
	57.762	76.289	24.350	508.048	644.264	362.542
	24.863	34.246	11.899	398.288	516.286	265.104
	21.064	30.144	10.709	373.766	240.162	253.515
	20.820	13.512	5.459	174.855	229.607	142.546

Table 8.3: Top consuming system calls and the values consumed (*psys*) in all test cases.

Chapter 9

Conclusion

The goal of this thesis was to implement a novel energy profiler for the Perun project and together with a way to visualize the results it creates. The profiler is capable of sampling at high frequencies the energy consumption of a system and put that consumed energy into the context of system calls. The profiler operates with low overhead, requires minimum number of dependencies and is publicly available under the GPLv3 license. Moreover we proposed a reporting tool capable of using the profiler's output data to create both text reports and visualizations of the profiled data providing a comprehensive and easy-to-use way to interpret the profiler's results.

We evaluated the profiler on a series of experiments where the GNOME Shell software was profiled in different runtime environments. The experimental evaluation showed that the profiler a) is capable of locating „hot spots“ in the runtime of applications, and, b) is capable of picking up differences in the profiling runtime conditions.

Future work. The next step is to fully integrate the energy profiler in the Perun project as a new collector. Because the implementation of the profiler was designed with this goal in mind, this step should be straightforward. Adding support to the profiler to trace multiple processes could help in reducing the effect of noise on the results of the profiler by using the additional system call traces in the scoring algorithm. The informational value of the profiler could also be increased significantly by adding support for scoring functions in Callgraphs. The performance of the reporting tool could be improved by leveraging a plotting module making use of multiple threads or even GPU APIs.

Bibliography

[1] *Bpftrace* [IO Visor Project]. Available at: https://github.com/iovisor/bpftrace.

[2] *Chipwhisperer* [NewAE Technology Inc.]. Available at: https://github.com/newaetech/chipwhisperer.

[3] *Ctypes — A Foreign Function Library for Python*. Available at: https://docs.python.org/3/library/ctypes.html.

[4] *HID-BPF — The Linux Kernel Documentation*. Available at: https://docs.kernel.org/hid/hid-bpf.html.

[5] *Linux Kernel Developers' Bpfconf 2019*. Available at: http://vger.kernel.org/bpfconf2019.html#session-2.

[6] *Pin - A Dynamic Binary Instrumentation Tool*. Available at: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[7] *Side Channel Analysis Tool | eShard*. Available at: https://eshard.com/side-channel.

[8] *What Is eBPF? An Introduction and Deep Dive into the eBPF Technology*. Available at: https://www.ebpf.io/what-is-ebpf.

[9] *BPF Compiler Collection (BCC)* [IO Visor Project]. September 2022. Available at: https://github.com/iovisor/bcc.

[10] *Wachy* [Rubrikinc]. December 2022. Available at: https://github.com/rubrikinc/wachy.

[11] *Ebpf_exporter* [Cloudflare]. March 2023. Available at: https://github.com/cloudflare/ebpf_exporter.

[12] *GNOME Shell* [GNOME]. April 2023. Available at: https://gitlab.gnome.org/GNOME/gnome-shell.

[13] *Mutter* [GNOME]. April 2023. Available at: https://gitlab.gnome.org/GNOME/mutter.

[14] Aggarwal, K., Hindle, A. and Stroulia, E. GreenAdvisor: A Tool for Analyzing the Impact of Software Evolution on Energy Consumption. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, September 2015, p. 311–320. DOI: 10.1109/ICSM.2015.7332477. ISBN 978-1-4673-7532-0. Available at: http://ieeexplore.ieee.org/document/7332477/.

[15] Aggarwal, K., Zhang, C., Campbell, J. C., Hindle, A. and Stroulia, E. The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes. In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*. USA: IBM Corp., 2014, p. 219–233. CASCON '14.

[16] Brendan, G. *Frequency Trails*. February 2014. Available at: https://brendangregg.com/frequencytrails.html.

[17] Brendan, G. *BPF Binaries: BTF, CO-RE, and the Future of BPF Perf Tools*. November 2020. Available at: https://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html.

[18] Cees-Bart Breunesse, Ilya Kizhvatov, Ruben Muijrers and Riscure. *Jlsca* [Riscure]. March 2023. Available at: https://github.com/Riscure/Jlsca.

[19] Corbet, J. *BPF Comes to Firewalls*. February 2018. Available at: https://lwn.net/Articles/747551/.

[20] Fiedor, T. and Pavela, J. *Perun Documentation*. June 2022. Available at: https://raw.githubusercontent.com/tfiedor/perun/master/docs/pdf/perun.pdf.

[21] Fiedor, T. and Pavela, J. *Perun: Lightweight Performance Version System*. July 2022. Available at: https://github.com/tfiedor/perun.

[22] Garcia, J. A. Exploration of Energy Consumption Using the Intel Running Average Power Limit Interface. In: *2019 IEEE Space Computing Conference (SCC)*. July 2019, p. 1–10. DOI: 10.1109/SpaceComp.2019.00005.

[23] Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J. et al. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, p. 896–904. DOI: 10.1109/IPDPSW.2015.70.

[24] Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B. et al. Energy Profiles of Java Collections Classes. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, p. 225–236. DOI: 10.1145/2884781.2884869. ISBN 978-1-4503-3900-1. Available at: https://dl.acm.org/doi/10.1145/2884781.2884869.

[25] Hielke de Vries. *Io_uring-Echo-Server*. Available at: https://github.com/frevib/io_uring-echo-server.

[26] Hoyer, H. *Rust_echo_bench*. Available at: https://github.com/haraldh/rust_echo_bench.

[27] Hunter, J. D. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*. 2007, vol. 9, no. 3, p. 90–95. DOI: 10.1109/MCSE.2007.55. ISSN 1521-9615. Available at: http://ieeexplore.ieee.org/document/4160265/.

[28] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K. and Ou, Z. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*. march 2018, vol. 3, no. 2, p. 9:1–9:26. DOI: 10.1145/3177754. ISSN 2376-3639. Available at: https://doi.org/10.1145/3177754.

[29] Kizhvatov, I. *Pysca*. April 2023. Available at: https://github.com/ikizhvatov/pysca.

[30] Kocher, P., Jaffe, J. and Jun, B. *Introduction to Differential Power Analysis and Related Attacks*. August 1998. Available at: https://www.rambus.com/introduction-to-differential-power-analysis-and-related-attacks/.

[31] Kocher, P. C., Jaffe, J. and Jun, B. Differential Power Analysis. In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 1999, p. 388–397. CRYPTO '99. ISBN 3-540-66347-9.

[32] Li, D., Hao, S., Halfond, W. G. J. and Govindan, R. Calculating Source Line Level Energy Information for Android Applications. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. Lugano Switzerland: ACM, July 2013, p. 78–89. DOI: 10.1145/2483760.2483780. ISBN 978-1-4503-2159-4. Available at: https://dl.acm.org/doi/10.1145/2483760.2483780.

[33] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G. et al. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita: IEEE, March 2016, p. 517–528. DOI: 10.1109/SANER.2016.85. ISBN 978-1-5090-1855-0. Available at: http://ieeexplore.ieee.org/document/7476671/.

[34] Liu, K., Pinto, G. and Liu, Y. D. Data-Oriented Characterization of Application-Level Energy Optimization. In: Egyed, A. and Schaefer, I., ed. *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, vol. 9033, p. 316–331. DOI: 10.1007/978-3-662-46675-9_21. ISBN 978-3-662-46674-2 978-3-662-46675-9. Available at: http://link.springer.com/10.1007/978-3-662-46675-9_21.

[35] McKinney, W. Data Structures for Statistical Computing in Python. In: *Python in Science Conference*. Austin, Texas: [b.n.], 2010, p. 56–61. DOI: 10.25080/Majora-92bf1922-00a. Available at: https://conference.scipy.org/proceedings/scipy2010/mckinney.html.

[36] Močáry, P. *Performance Analysis of Programs Based on PIN Framework*. Brno, CZ, 2022. Bakalářská Práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23847/.

[37] Nakryiko, A. *BCC to Libbpf Conversion Guide*. February 2020. Available at: https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/#bpf-skeleton-and-bpf-app-lifecycle.

[38] Nakryiko, A. *BPF CO-RE (Compile Once – Run Everywhere)*. February 2020. Available at: https://nakryiko.com/posts/bpf-portability-and-co-re/.

[39] Nakryiko, A. *Journey to Libbpf 1.0*. August 2022. Available at: https://nakryiko.com/posts/libbpf-v1/.

[40] Nocera, B. *Power-Profiles-Daemon*. April 2023. Available at: https://gitlab.freedesktop.org/hadess/power-profiles-daemon.

[41] Nocera, B., Catanzaro, M. and Gompa, N. *Changes/Power Profiles Daemon*. July 2021. Available at: https://fedoraproject.org/wiki/Changes/Power_Profiles_Daemon.

[42] NotMoe, R. *Ydotool*. April 2023. Available at: https://github.com/ReimuNotMoe/ydotool.

[43] Pang, C., Hindle, A., Adams, B. and Hassan, A. E. What Do Programmers Know about Software Energy Consumption? *IEEE Software*. may 2016, vol. 33, no. 3, p. 83–89. DOI: 10.1109/MS.2015.83. ISSN 0740-7459, 1937-4194. Available at: https://ieeexplore.ieee.org/document/7155416/.

[44] Pathak, A., Hu, Y. C. and Zhang, M. Where Is the Energy Spent inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In: *Proceedings of the 7th ACM European Conference on Computer Systems - EuroSys '12*. Bern, Switzerland: ACM Press, 2012, p. 29. DOI: 10.1145/2168836.2168841. ISBN 978-1-4503-1223-3. Available at: http://dl.acm.org/citation.cfm?doid=2168836.2168841.

[45] Pavela, J. *Efficient Techniques for Program Performance Analysis*. Brno, CZ, 2020. Master's thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/19092/.

[46] Pinto, G. and Castor, F. Energy Efficiency: A New Concern for Application Software Developers. *Communications of the ACM*. november 2017, vol. 60, no. 12, p. 68–75. DOI: 10.1145/3154384. ISSN 0001-0782. Available at: https://doi.org/10.1145/3154384.

[47] Pinto, G., Castor, F. and Liu, Y. D. Understanding Energy Behaviors of Thread Management Constructs. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. Portland Oregon USA: ACM, October 2014, p. 345–360. DOI: 10.1145/2660193.2660235. ISBN 978-1-4503-2585-1. Available at: https://dl.acm.org/doi/10.1145/2660193.2660235.

[48] Pinto, G., Liu, K., Castor, F. and Liu, Y. D. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, October 2016, p. 20–31. DOI: 10.1109/ICSME.2016.34. ISBN 978-1-5090-3806-0. Available at: http://ieeexplore.ieee.org/document/7816451/.

[49] Prometheus. *Prometheus - Monitoring System & Time Series Database*. Available at: https://prometheus.io/.

[50] Rambus Press. *Side-Channel Attacks Explained: Everything You Need to Know*. October 2021. Available at: https://www.rambus.com/blogs/side-channel-attacks/.

[51] Silberschatz, A., Galvin, P. B. and Gagne, G. *Operating System Concepts*. Ninth editionth ed. Hoboken, NJ: Wiley, 2013. ISBN 978-1-118-06333-0.

[52] Team, T. P. D. *Pandas-Dev/Pandas: Pandas* [Zenodo]. April 2023. DOI: 10.5281/ZENODO.3509134. Available at: https://zenodo.org/record/3509134.

[53] Torvalds, L. *Torvalds/Linux*. April 2023. Available at: https://github.com/torvalds/linux.

[54] Ven, A. van de. *PowerTOP*. April 2023. Available at: https://github.com/fenrus75/powertop.

[55] Wang, Y., Paccagnella, R., He, E., Shacham, H., Fletcher, C. W. et al. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on X86. In: *Proceedings of the USENIX Security Symposium (USENIX)*. 2022.

[56] Waskom, M. Seaborn: Statistical Data Visualization. *Journal of Open Source Software*. april 2021, vol. 6, no. 60, p. 3021. DOI: 10.21105/joss.03021. ISSN 2475-9066. Available at: https://joss.theoj.org/papers/10.21105/joss.03021.

[57] Weaver, V. *Linux Support for Power Measurement Interfaces*. Available at: https://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html.

[58] Yingchen Wang. *Hertzbleed* [FPSG-UIUC]. April 2023. Available at: https://github.com/FPSG-UIUC/hertzbleed.

Appendices

List of Appendices

Appendix A

Full report of profile data from a test case

```
# General information:
Process                   gnome-shell
Profile time              2023-04-28T05:08:52
Profile duration          0:00:30.170295
Energy profile frequency  99 Hz

# System information:
system     Linux
node       localhost-live
release    6.2.9-300.fc38.x86_64
version    #1 SMP PREEMPT_DYNAMIC Thu Mar 30 22:32:58 UTC 2023
machine    x86_64
processor  Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz

# Ratio of energy consumption (total/process):
energy domain      system [J]    process [J]  ratio
psys                  335.378          2.510  0.75%
pkg                   101.770          0.767  0.75%
cores                  52.966          0.383  0.72%
gpu                    12.379          0.107  0.86%
ram                    59.050          0.452  0.77%

# Top consuming system calls:
system call      psys [mJ]    pkg [mJ]    cores [mJ]    gpu [mJ]    ram [mJ]
write              700.995     209.101        97.413      32.683     128.304
read               508.048     151.906        71.885      23.059      93.057
poll               398.288     127.629        68.833      16.072      69.498
ioctl              373.766     115.554        62.071      12.568      66.696
epoll_wait         174.855      51.404        22.856       8.637      32.112
```

Listing A.1: Text report from the LBB test case (see Chapter 8)

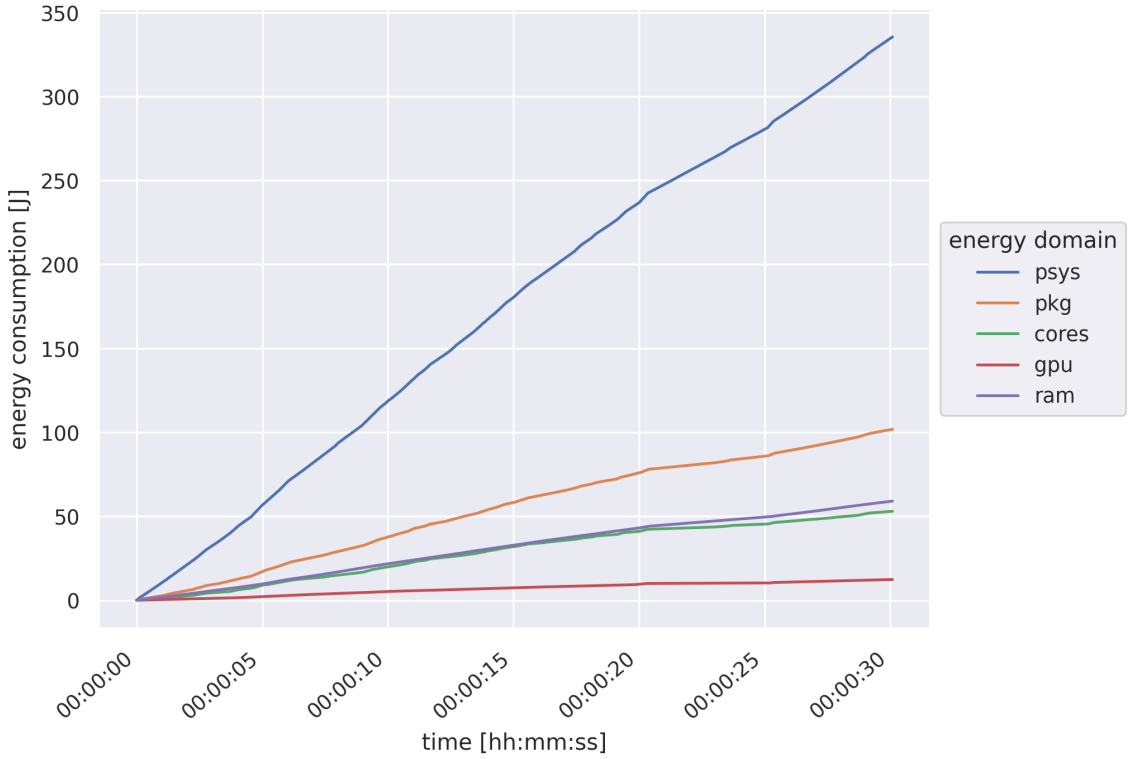


Figure A.1: Line plot of per-energy domain energy consumption in time from the LBB test case (see Chapter 8)

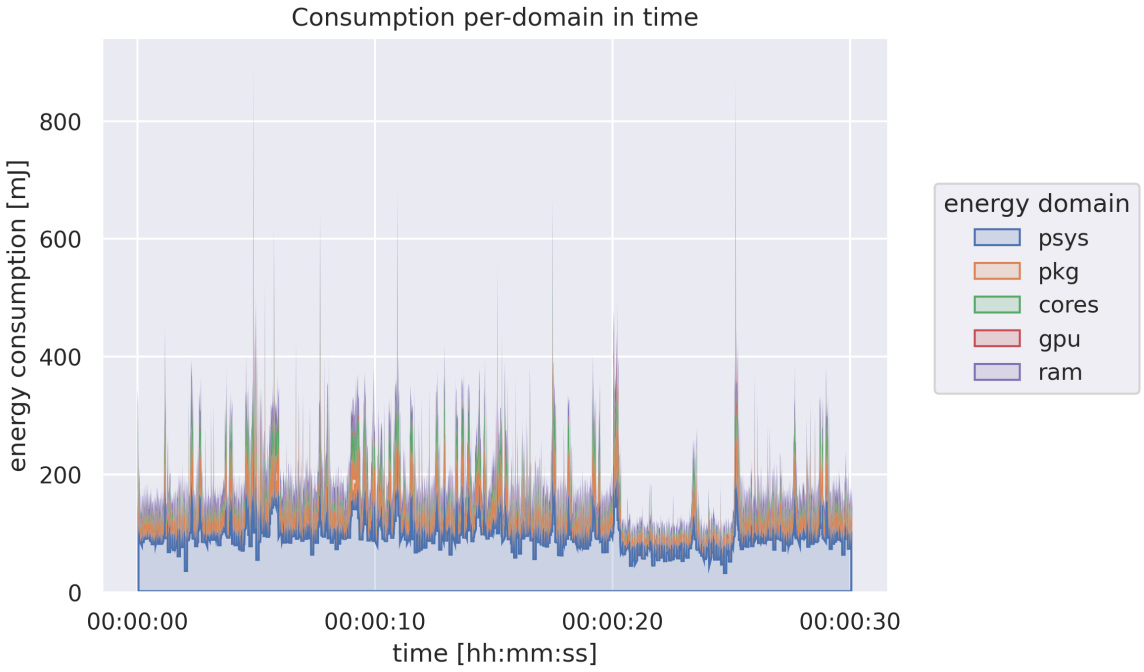


Figure A.2: Line plot of cumulative per-energy domain energy consumption in time from the LBB test case (see Chapter 8)

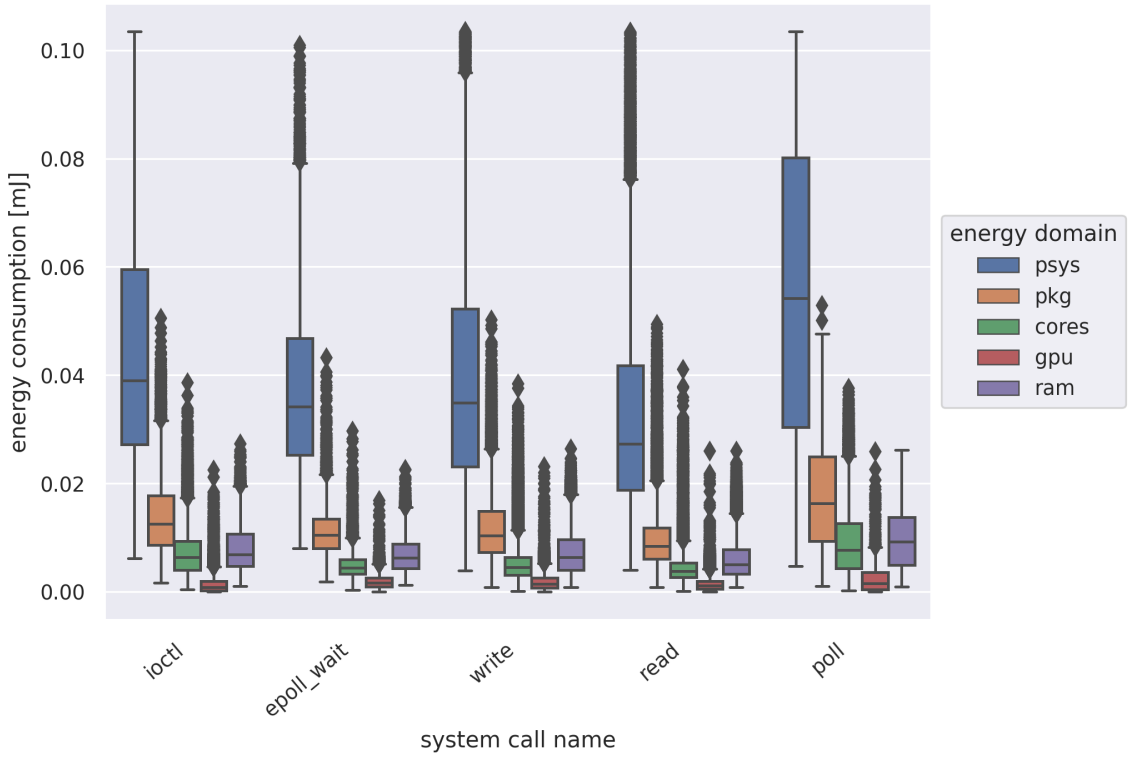


Figure A.3: Box plot showing distribution of energy consumed by system calls across different energy domains from the LBB test case (see Chapter 8)

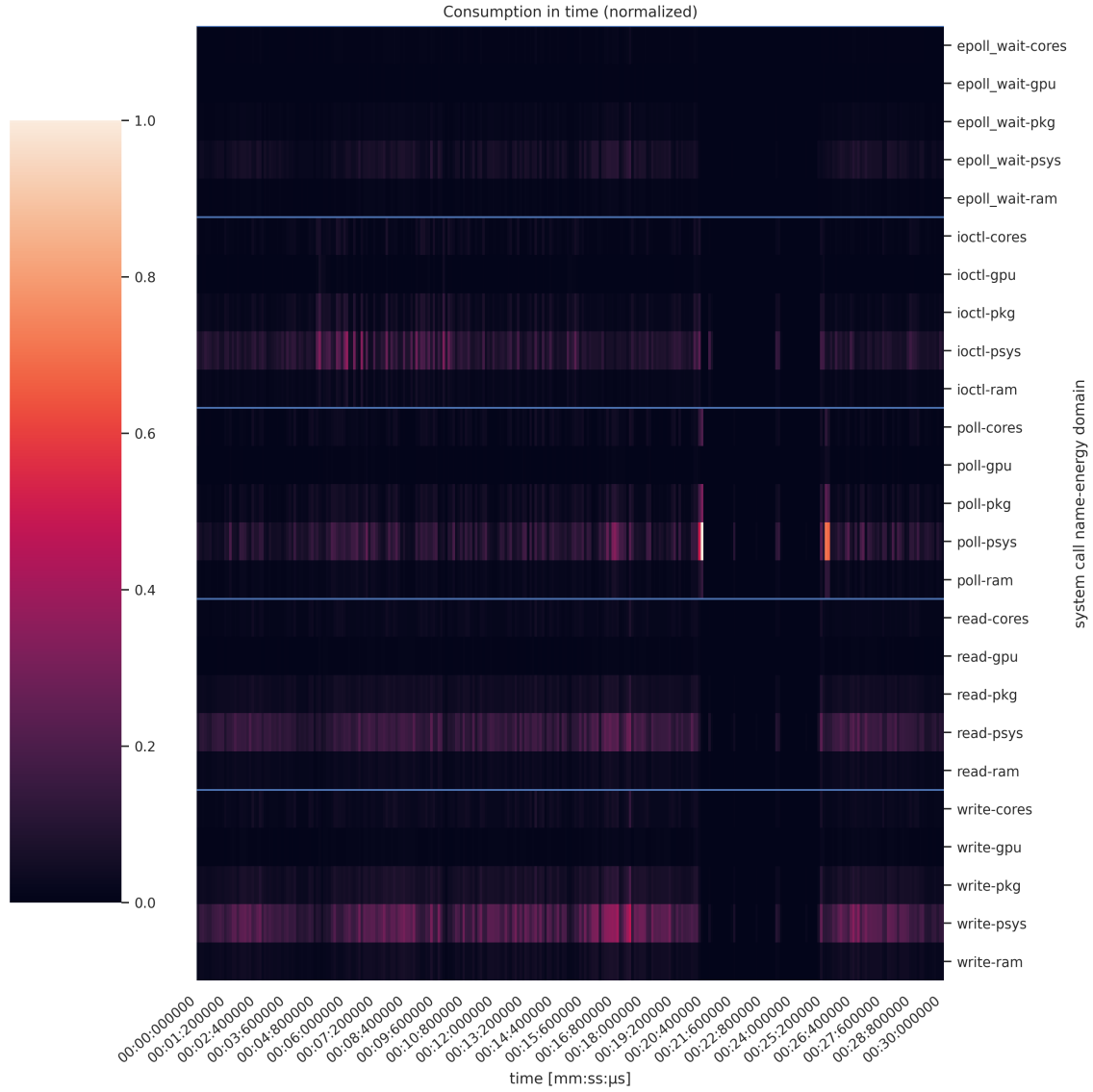


Figure A.4: Heatmap of per-energy domain system call energy consumption in time from the LBB test case (see Chapter 8)

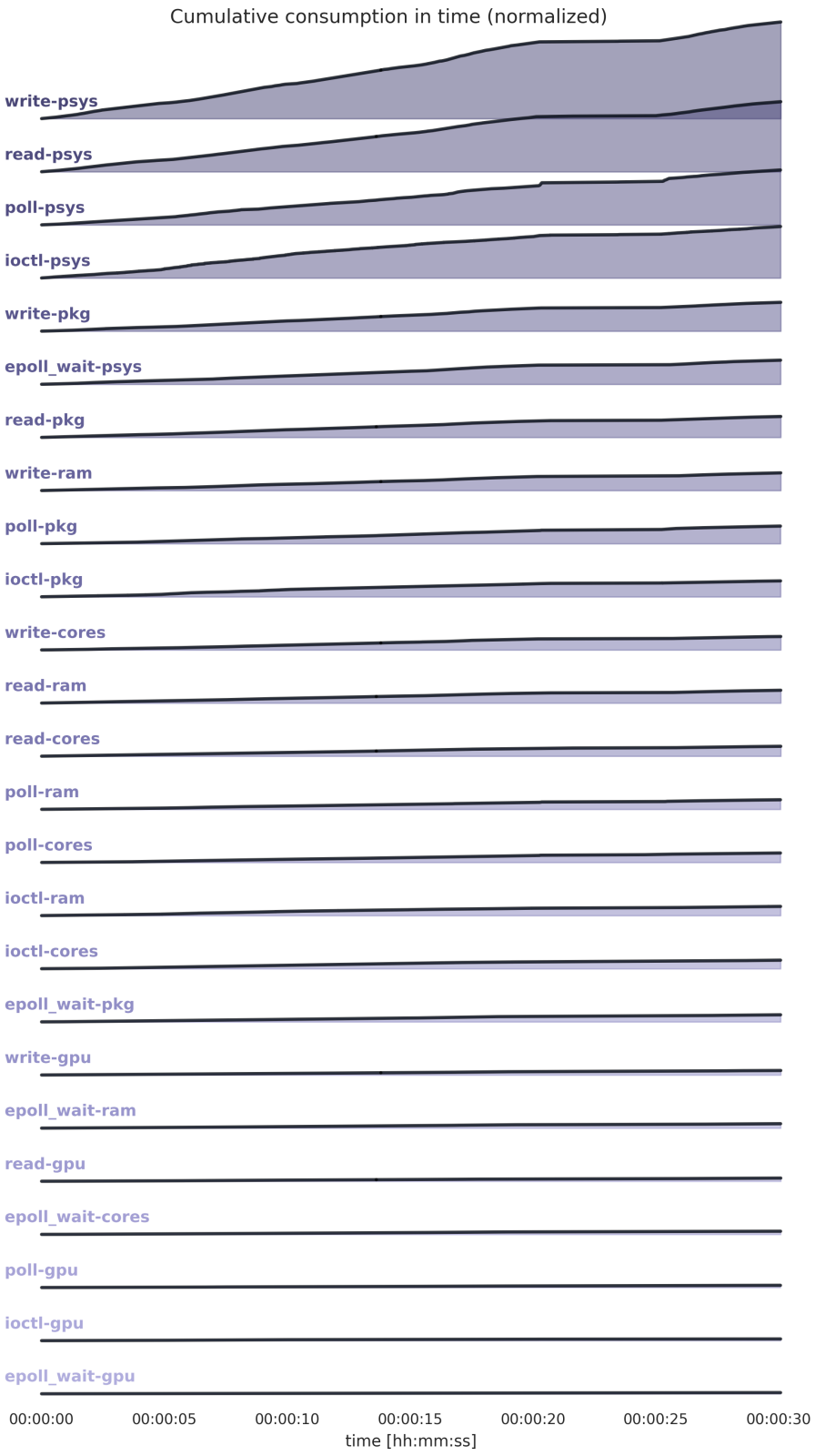


Figure A.5: Tsunami graph of cumulative per-energy domain system call energy consumption from the LBB test case (see Chapter 8)

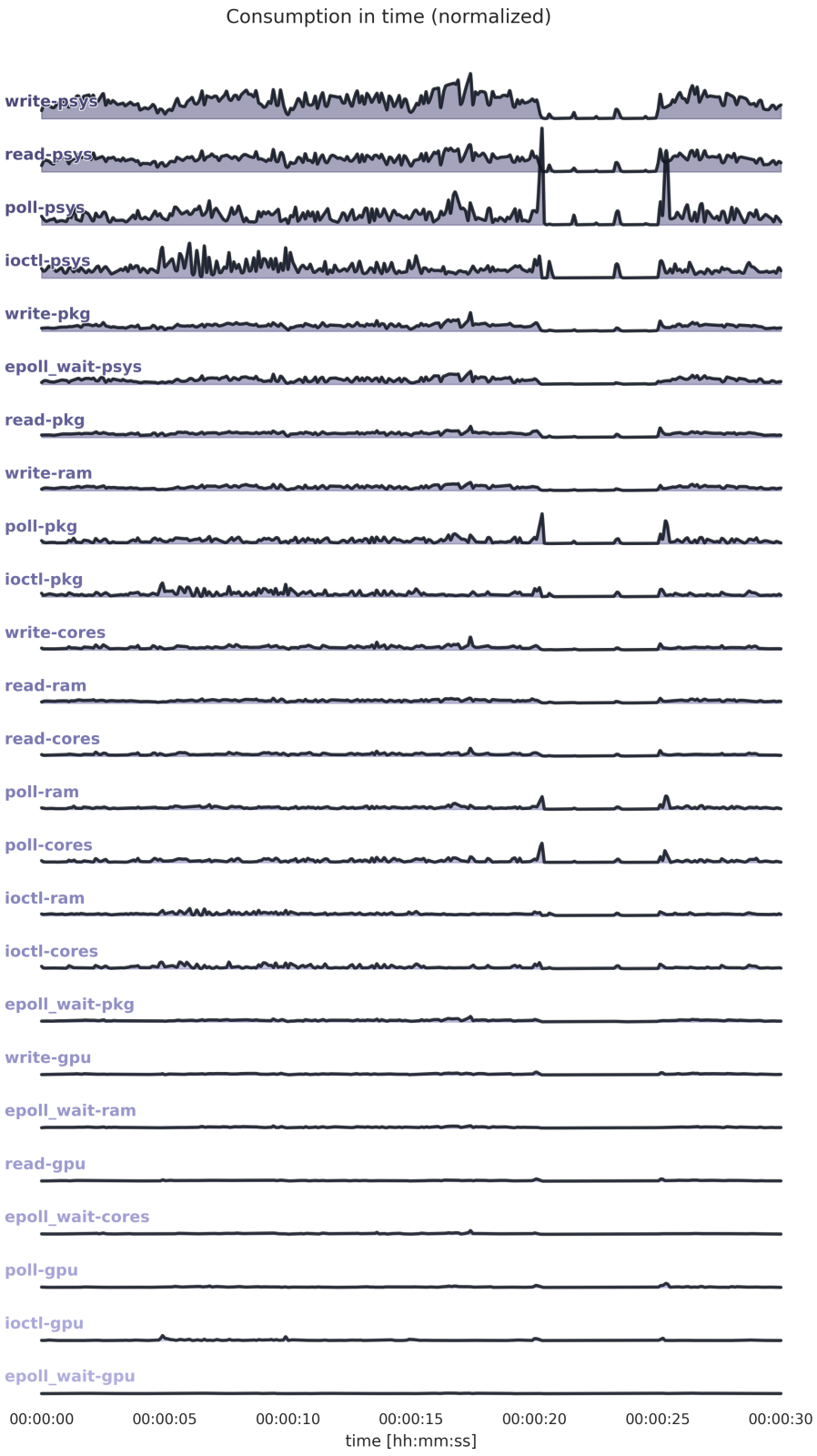


Figure A.6: Waterfall graph of per-energy domain system call energy consumption in time from the LBB test case (see Chapter 8)