Discovering energy inefficiencies in Docker through tracing: A case study with Redis

Enrique Barba Roque

Discovering energy inefficiencies in Docker through tracing: A case study with Redis

by

Enrique Barba Roque

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 25, 2024 at 15:30 AM.

Student number:	5849152	
Project duration:	November 6, 2023 – June 25, 2024	
Thesis committee:	Arie van Deursen,	Thesis advisor
	Luís Miranda da Cruz	Supervisor
	Thomas Durieux	Supervisor
	Sicco Verwer	External examiner

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Preface

A large thanks to Luis and Thomas, for making time every week for our meetings and supporting me along the way. I'd also like to thank Diomidis Spinellis for his insight on our results, and Joseph P. De Veaugh-Geiss and the KDE Eco community for giving me the chance to share my work with them.

Enrique Barba Roque
Delft, June 2024

Summary

Containerization with Docker has become the standard for the deployment of software in recent years since it is a lightweight method to isolate applications. However, the selection of a Docker image brings different dependencies, which can introduce energy inefficiencies that are often not documented. Recent work performs a comparative study between different base images and finds certain patterns in which some images tend to be less energy efficient for certain tasks. In this thesis, we propose a methodology to find the reason behind observed energy inefficiencies in a Docker image by using tracing to study how a workload makes use of the dependencies provided by the image. We apply this methodology to test the energy efficiency of Redis when using different base images. This leads to finding and raising awareness of energy inefficiencies in some functions from musl, the C standard library implementation used by Alpine.

Contents

List of Figures

List of Tables

1 Introduction

In later years, the demand for computing power has grown, and data centers are increasing in number and size. An increase in energy consumption accompanies this increase in demand. It is estimated that by 2025 data centers will consume 20% of global electricity and account for 5.5% of global emissions [3]. Because of these reasons, the field of Sustainable Software Engineering tries to raise awareness about the energy consumption in software. Sustainable Software Engineering and the study of energy efficiency in software have been relevant in the field of mobile and smartphone development [10], given the need to optimize battery life in these devices, but the optimization of cloud deployments is still a relatively new area of study.

Traditionally, performance optimization is believed to be a task exclusively for compilers. However, some studies show that bloating and certain software engineering principles also affect energy consumption [35, 1]. Modern software relies on a large number of libraries and dependencies, which are built on top of other dependencies, with the objective of increasing abstraction and facilitating development.

Each dependency introduces its own data structures, functions, and classes, and this complexity increases the deeper we go on the dependency tree. This can introduce bloat and inefficiencies by using functionalities that are too complex for the intended functionality. For example, using a complex implementation of a Set that most of the time will contain just one or a very small amount of elements.

Due to the complexity of dependency trees of modern software, it is very difficult to locate and understand where energy inefficiencies come from, and which libraries perform better or worse in this sense. Therefore, developers often select their tools and libraries based on other, more immediate parameters, like runtime or size. However, a library being better in these measurements does not necessarily mean that it is more energy efficient.

In cloud environments, Docker is commonly used to deploy applications, since containerization provides a lightweight process to start and stop your software while providing robust isolation at a much lower cost than virtual machines. Some papers [30, 25] compared the energy efficiency of Docker containers against virtual machines, and showed that virtual machines incur in a much higher energy consumption than Docker containers.

However, Docker containers still provide a number of libraries and binaries over which the applications are run, and different versions of these dependencies might lead to different energy consumption. A recent thesis [32] considers this, and performs a comparative study in which the energy consumption of different workloads is compared when switching the base Docker image to build the dependencies.

One of the study's main findings is that, for most of the workloads, Alpine tends to have a higher energy consumption when compared to other base images. The main hypothesis for this observed inefficiencies in Alpine is a performance difference in the Alpine implementation of the C standard library (*libc*). Ubuntu and most other Linux-based operating systems use the GNU C Library implementation or *glibc*, while Alpine uses a custom implementation called *musl*, with the aim of reducing clutter and bloat compared to the GNU implementation.

This thesis will delve more into finding out the reasons why Alpine is less energy efficient than other images in certain workloads. We will research which are the differences in terms of libraries and binaries between Alpine and other Linux distributions, focusing on the C standard library implementation and its effects on energy consumption. For this, we will focus on one of the workloads from the previous study (Redis) and find out which functions are used the most in different points of execution, especially in the parts of the workload where the consumption difference is higher. To do this, we will define a formal approach, providing a methodology that other researchers or developers can apply to investigate energy inefficiencies and regressions in other workloads that use different libraries or technologies.

1.1. Research questions

This thesis will answer the following research questions to guide our research and design of energy experiments and validate hypotheses.

RQ1. What is the difference between glibc and musl in terms of energy consumption?

The first step in our research will be to verify the hypothesis made by previous work and confirm if there is an energy performance difference between different implementations of the C standard library. To do this, we will try to replicate the results of the previous thesis and define additional experiments that are oriented to verify the hypothesis.

RQ2. Can we use tracing to compare energy consumption differences in shared libraries?

Once we confirm the previous hypothesis, the main focus of this thesis is to explain the specific reasons for the energy difference between *libc* implementations. The objective is to find the concrete function or functions from *libc* that are the cause behind the observed difference in energy consumption. To do this, we will define a formal methodology using tracing tools to record the functions that are being used by the workload.

RQ3. Can we verify the origin of energy consumption differences by recreating the workload behavior closely?

Finally, to fully confirm the function that is causing the observed difference in energy consumption, we will try to emulate the original workload as much as possible while using only functions that we know do not have an impact on energy plus the function that introduces the impact. If we still observe a similar difference in energy, we know that function is the most probable cause. To do this, we will study the energy consumption of other functions, specifically those associated with the operating system kernel, which should be shared by the Docker containers and therefore, not present a difference in energy usage.

1.2. Contributions

The thesis provides the following contributions to the field of Sustainable Software Engineering.

Replication of previous work and verification of the proposed hypothesis about differences in *libc* implementation

We replicate and extend the experiments done for Redis in previous work so we can verify the hypothesis proposed, giving some insights into the energy performance of different *libc* implementations.

A methodology and pipeline to study energy regressions in software

We propose a methodology based on program tracing to detect the most used external functions of a workload and measure their impact on total energy consumption. We provide a detailed description of the steps to take, as well as a Jupyter Notebook for easier usage and replication of our results[1].

Set of energy benchmarks for some *libc* functions

We provide the set of energy benchmarks we used to test and confirm the energy differences of `memcpy` and other functions of the C standard library. They are provided as experiments in the *docker-energy* framework for easier reproducibility.

1.3. Outline

The rest of this thesis is structured as follows: Chapter 2 gives some background into the most important technologies for the Thesis. Chapter 3 presents an overview of literature related to this topic. Chapter 4 presents our case study with a timeline of our research into Redis energy consumption, and the results obtained, and presents the methodology to successfully find energy inefficiencies in shared libraries for a piece of software, as well as the requisites to do so. Chapter 5 discusses more in-depth the implications derived from the results of the Redis investigation and the methodology proposed. Finally, Chapter 6 presents some final conclusions and discusses possible future research on the topic.

[1]`https://github.com/enriquebarba97/energy-hotspots`

2 Background

This chapter presents an essential background to understand how we run and analyze our energy experiments. We will explain how energy consumption is measured on a computer, and different tools and alternatives to perform these measurements. We also explain the difference between different virtualization technologies and how each of them isolates their workloads.

2.1. Energy experiments and measurement

The awareness of climate change and sustainability has been growing in later years [4], and with it, multiple fields are trying to reduce their impact on carbon emissions. Among them, the field of software has to be considered, given how it permeates across all of society. Given the digital nature of software, it might not be obvious how this affects climate change. However, this software needs to run on machines and servers, which require large amounts of energy [3], and the source of this energy plays a major role in carbon emissions.

There are two main ways of measuring energy consumption: physical power meters and software profilers. Each has different advantages and limitations, and we will compare multiple options in this section.

2.1.1. Physical power meters

Physical power meters provide the most accurate measurements [6] and it has been used previously in other studies [25]. They are connected between the power socket and the computer to test, which can provide measures of instantaneous power usage through the device.

One device of this type is the *Watts Up Pro*[1] which provides instantaneous measures of voltage and current, which can be used to calculate power. Another device is the *Raritan*[2] power distribution unit, which computes the average of instant power over periods of time, which is known as active power in alternating current systems [19].

This kind of measurement device is often difficult to set up and needs calibration for the machine that it is going to be measured. Additionally, they only provide a power measurement for the whole machine, without the ability to measure individual components. This can be misleading since idling components still consume some power, and this would not represent the usage of the specific workload under study.

2.1.2. Software profilers

An alternative to physical power meters is to use software-based profilers. These tools can collect data on energy usage directly from the CPU registers dedicated to collecting energy usage data for other tools to use, like performance monitoring. These measurements are based on estimations, which makes them less reliable than those obtained with physical meters. However, they can report individual

[1]`https://www.powermeterstore.com/p1206/watts_up_pro.php`

[2]`https://www.raritan.com/eu/products/power/power-distribution/intelligent-rack-pdus`

energy usages of components, like memory and CPU packages, or even consumption of individual CPU cores.

The information that the software profiler can report heavily depends on the interface provided by the processor. The most common interface is Intel's Running Average Power Limit or RAPL [18]. This interface is used for measuring and limiting energy consumption for different power domains of the computer [14]. Concretely, it can provide measurements for the following domains:

- **Package (PKG)**: Energy consumption of the entire socket, including all cores, integrated graphics, caches, and memory controller.
- **Power Plane 0 (PP0)**: Energy consumption of all cores of the CPU.
- **Power Plane 1 (PP1)**: Energy consumption of integrated GPU
- **DRAM**: Energy consumption of the RAM attached to the system
- **PSys**: Reports the power specifications of the entire system-on-chip.

There are several tools that use RAPL to measure the energy consumption of different processes in Linux. The most common tool is the `perf` command [34], which is able to measure both computing performance in the form of total runtime and CPU cycles, as well as the total energy consumption of an application. Other options are PowerTOP[3] or Powerstat[4]. These tools provide power usage statistics after running a workload, including average, standard deviation, and minimum and maximum power usage, in Watts. Total energy consumption must be computed afterward from these statistics.

The RAPL interface is an implementation made by Intel, and available only on Intel processors. However, AMD provides its own implementation of RAPL [27], and shares certain registers with the original implementation. This means that the previously mentioned tools are partially compatible with AMD processors. AMD's implementation of this interface provides more fine-grained details, like being able to report the energy consumption of individual cores of the CPU.

For this reason, we will use an AMD machine to run our energy experiments, and we will use the software profiler tool *Energibridge* [24] to make energy measurements. *Energibridge* is a cross-platform energy measurement tool, supporting multiple operating systems (Windows, Linux, and Mac OS) and different CPU architectures like Intel, AMD, and Apple ARM. It does so seamlessly: you only need to call the tool following the instructions, with no need for specific configurations depending on the platform it is run.

Instead of relying on existing tools for each of the specific architectures, which would introduce overhead, *Energibridge* uses low-level system calls to collect energy data directly from the platform tools, the *System Management Controller* on Mac or the *Model-specific registers* on Intel and AMD. After collection, it can write the data to an output CSV file.

2.2. Operating systems and containerization

An operating system is a system software that acts as an intermediary between the hardware of a computer and the different applications that want to use that hardware, providing a unified interface to access and manipulate these resources [29]. There are multiple components to an operating system. For the purpose of our work, we will distinguish between kernel-level software and routines, (e.g. writing to a file), and user-level functionality, like library functions.

The user-level functionality is software that can be used directly by the user to work with the system. This includes, for example, the user interface, be it a command-line interface or a graphical user interface. Part of this user-level functionality is the binaries for shared libraries, or other binaries that can be used by developers to develop their own programs. One of these libraries is the C standard library, which provides basic functionality for programming in C.

However, this user-level software cannot contain any explicit implementation to access the hardware in the system directly. Accessing and directly handling hardware is the responsibility of the kernel, the core of the operating system. The kernel contains code that provides access to the different hardware

[3]PowerTOP GitHub repository `https://github.com/fenrus75/powertop` retrieved on Jun 17, 2024
[4]Powerstat GitHub repository `https://github.com/ColinIanKing/powerstat` retrieved on Jun 17, 2024

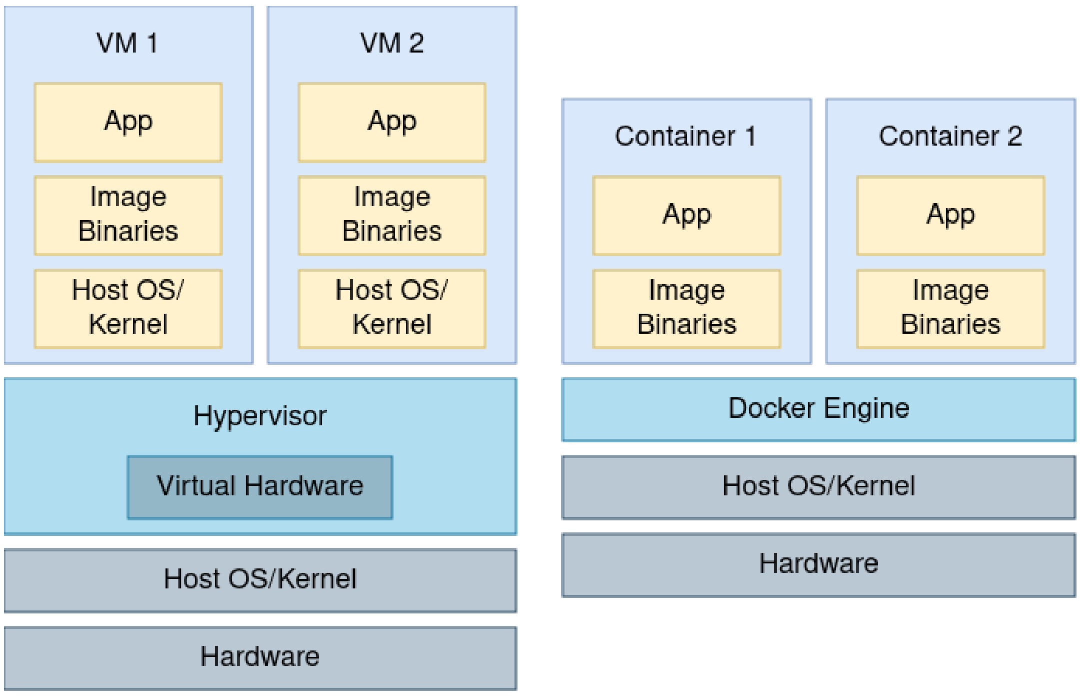


Figure 2.1: Comparison of hypervisor and container architecture

devices in the system, like hard drives, using their drivers. Access to these services is done through an API of system calls.

To guarantee that the hardware is not tampered with by other code that is not part of the kernel, CPUs have different privilege modes, which limits what code can be executed at each time. We will explain this using a C program as an example. When a C program starts running, it is assigned a section of memory and runs in a low privilege mode, where it can run operations that require no privileges, like assigning variables or arithmetic computation. If at some point, the program wants to write into a file, it calls the `write` function from the C standard library. This function does not contain any specific implementation to directly manipulate the filesystem. Instead, it prepares the arguments and performs a system call to the kernel. Then, the program is interrupted, the privilege mode is increased, and the trusted code from the kernel takes over, performing the operation and returning the result to the program, de-escalating privileges again. This way, when a program wants to do some sensitive operation with the hardware, it has to go through the appropriate system call from the kernel, which provides trusted and secure access.

2.2.1. Virtualization and containerization

In computers, virtualization is used to create an abstraction between hardware resources and software processes. This way, software can be packaged with all its requirements and dependencies already configured, and hardware resources like CPU cores or memory can be individually assigned to the software.

There are two main approaches to virtualization: with virtual machines and a hypervisor or with containers. Figure 2.1 shows the architecture of both methods. In hypervisor-based virtualization, hardware is abstracted by the hypervisor, which emulates virtual hardware devices and acts as an interface with the physical devices [26].

A virtual machine contains a full software system, including the guest operating system kernel, filesystem, and its binaries, besides the application to isolate. The guest OS treats the virtual hardware as actual hardware, not knowing they are emulated devices. When the application wants to access a hardware resource, it calls the guest kernel, which accesses the virtual hardware. This request actually arrives at the hypervisor, which maps the operation to the underlying physical hardware using the host OS kernel.

This technology offers a large grade of isolation, given that the guest systems have no access to information about the machine they are hosted in. However, they are usually heavy, since a virtual machine has to include a full operating system, and they have a large performance overhead due to the additional layers of translation introduced by the hypervisor [28, 20].

A second technology that has grown in popularity over the last few years is containerization. Instead of performing complete hardware emulation to achieve isolation, containers use technologies available in the operating system kernel to isolate certain resources from one another. The most popular container platform in Linux is the Docker engine. This technology uses a couple of Linux kernel functionalities to manage the isolation and resource requests of containers to the kernel called namespaces and control groups.

Namespaces are a way of limiting the visibility of processes and resources and their interaction with other processes [17]. When a process is assigned a namespace, it can only interact with other processes and resources that have the same namespace assigned. Namespaces can also have parent-child hierarchical relations, so processes in a parent namespace can see resources in children's namespaces but not the other way around. The Docker engine uses namespaces to start processes and containers, so they cannot interact with each other or with other critical parts of the host system.

Control groups, or *cgroups* are a kernel feature that can allocate specific amounts of hardware resources like CPU utilization, network or disk usage, to different processes in the system. It can also manage the contention of resources by scheduling access for the processes. With this feature, resources can be distributed evenly between containers, and we can assign individual physical cores to a container, which will help in energy experiments.

To package the necessary information to build a container, Docker uses images. These images are read-only templates that contain the instructions to build and start a container [16], equivalent to a virtual machine snapshot. Images are built and saved in a layered filesystem, following instructions from a script-like file called Dockerfile. The building starts from a base image (e.g. *ubuntu*, which contains the binaries and libraries of the Ubuntu distribution) and each instruction adds or modifies the specified files in a new layer. This makes layers reusable, which can cache operations for future builds or be reused in other images that follow some of the same instructions, which optimizes space usage.

When a container is started from an image, a writable layer is created, which contains the modifications done by the container during runtime. This means that containers are often lighter than virtual machines since multiple containers from the same image can reuse the read-only layers, and only need a writable layer for each instance.

Because of these features, containers are actually very isolated processes and filesystems, rather than virtual machines. This means that they do not need a fully-fledged guest operating system kernel, which makes them even lighter. When a process in a container needs to make use of a hardware resource through the kernel, the system call is done directly to the host kernel. This is an important detail for our work. Since we are trying to locate which parts of a base image are responsible for the difference in energy usage, we know that the kernel functions should not be part of the reason.

3 Related work

This chapter gives an overview of work in different fields related to this study. We look into several studies that look into the optimization of Dockerfiles and Docker images, as well as studies that compare Docker energy performance with other virtualization technologies and bare metal. We also look into the energy performance of different programming languages and technologies, and the effect of compile-time optimizations on energy performance.

3.1. Energy optimization for Docker images

There have been a number of studies that focus on optimizing Docker images and containers. One way of optimizing Docker containers is by studying the impact of Dockerfiles and optimizing the building process of the image. For this area, there have been multiple studies that try to improve the quality of Dockerfiles to identify common pitfalls or Docker smells that tend to affect the performance of a Docker container. Some tools use AST parsing of the Dockerfile to provide base image suggestions based on the instructions used in the file through the use of neural networks [38] or to detect and warn the developers about bad practices that appear in the Dockerfile [11]. Other studies propose techniques and approaches to assess image quality, size, and build times, based on the evolutionary trajectory of the artifacts [37] or Docker smells [23]. Although all of these studies bring important contributions to their fields, none of them study Docker from an energy perspective.

There have been other studies that compare energy consumption for virtualization and containerization. Researchers like Morabito et al. [19] perform an empirical comparison of different virtualization technologies where they compare KVM and Xen as hypervisor-based virtualization and Docker and LXC as container-based technologies, measuring power usage through a physical meter. They find that, for CPU-heavy workloads, power usage is similar for all technologies, while container-based technologies show better performance for network tasks. However, this study is limited to comparing both technologies, without researching if different Docker configurations or base images could show different performance.

Another comparison between virtualization technologies is provided by Tadesse et al. [30]. In this paper, they only compare VirtualBox and Docker, and test both synthetic workloads and real-world applications. Like in the previous paper, they find that CPU usage is similar for CPU-intensive tasks, but they show that the hypervisor-based virtualization gets less job done in the same amount of time, indicating that a large part of CPU usage is due to the translation layer introduced by the hypervisor, decreasing the performance of the task. For memory and other I/O workloads, containers are again more efficient than hypervisors. As with the previous paper, this work is limited to comparing hypervisors and containerization and does not delve into comparing the performance of different container configurations.

Santos et al. [25] also performed benchmarks to compare Docker energy efficiency to bare-metal Linux. For their study, they used realistic benchmarks with real-life applications like Redis, PostgreSQL, or WordPress, and found that using containers introduces a small energy overhead compared to a bare-metal Linux server. However, they show that this energy overhead is generally negligible, and the improved isolation granted by containers is a considerable advantage in improving development efforts.

In all cases, Docker shows better performance than virtual machines with respect to energy consumption, as well as lower CPU usage across the board. However, all of these studies focus on comparing bare metal, virtual machines, and Docker, and none of them look into how Docker energy performance varies according to configuration and or base images.

The work by Tjiong et al. [32] is one of the first studies to look into how base image selection affects the energy consumption of Docker containers for certain workloads. They perform multiple benchmarks for different common workloads, like databases, web servers, and CPU-intensive tasks such as video transcoding. They tested multiple popular base images like Ubuntu, Debian, and Alpine among others, and provide a framework to run and create your own tests.

One of their main findings is that Alpine, known and commonly used for being the slim and light alternative to other images, tends to have a worse energy performance while showcasing similar runtimes for the workloads in question. Some of the workloads that present this trend are Redis and Postgres.

However, this study is limited to reporting the results of the energy experiments, but it does not delve into why this difference exists. The authors suggest that it might stem from a difference in the implementation of the C standard library, but there is no evidence of that. Alpine is a popular image because it is smaller than other images and it is considered less bloated and more efficient. This work sheds light on understanding the unexpected differences in energy efficiency and provides insightful observations that can be useful to operative system developers or developers who use Alpine and care about energy efficiency.

3.2. Energy efficiency of programming languages and compilers

While this thesis uses Docker for its experiments, the energy inefficiencies observed can be the result of the technologies and programming languages used inside the images. Therefore, we look into studies that research the energy efficiency of different programming languages, as well as how compilers can optimize time and/or energy usage and possible trade-offs when compiling.

Pereira et al. [22] perform a study to compare the energy efficiency of a large variety of programming languages, using benchmarks for well-known and diverse programming problems, like computing digits of π. They find that the most energy-efficient languages are compiled languages like C, C++, and Rust, and the least energy-efficient are interpreted languages like Python or Lua. These results are reasonable, given that an interpreter introduces an additional layer of complexity and latency.

For compiled languages, there has also been some research on optimizing energy consumption. In general, the main focus for compilers during their development is on producing more time-efficient and faster code. Although this generally leads to a more energy-efficient program, Zambreno et al. [36] show that this is not always the case, especially for memory performance.

The paper by Pallister et al. [21] makes an exploratory study of different combinations of optimization flags for the GCC compiler and their impact on the energy consumption of a program. The study is limited to embedded software in ARM architectures. They find that, in most cases, compiler optimizations bring a better performance and energy efficiency, but some combinations of optimizations and workloads increase energy usage without an improvement in runtime performance and conclude that there is not a universally optimal set of optimization options. While these are relevant observations, the study only focuses on optimizations introduced by the compiler itself. The software can depend on other shared libraries which might introduce their own optimization (like the C standard library). The different versions or implementations of these dependencies can introduce differences in energy performance.

3.3. Energy efficiency in other software fields

Although this thesis is focused on energy inefficiencies introduced by Docker, given the architecture of container virtualization, it is possible that the energy inefficiencies are a product of the software included with the container rather than the virtualization method itself. Because of this, we explore studies of energy efficiency in other fields of Software Engineering.

There has been some discussion and research into the inefficiencies introduced by the commonly used design patterns of reusability and object-oriented programming. The work by Xu et al. [35] presents some challenges in this area for the Java language. It shows how in certain cases when the data to store has a simple structure, using Java collection objects like `HashMap` introduces unnecessary complexity and performance issues due to the potentially unnecessary features and its dependencies on other classes and routines.

The paper by Bhattacharya et al. [1] makes a similar point. Modern large-scale applications are built over deeply-layered frameworks (e.g. a Javascript or Typescript framework) with modules designed for a high degree of interoperability and flexibility. However, most systems only use a subset of these functionalities, while the unused combinations can be an energy burden for the system. They present several case studies where simple applications accidentally create temporary objects larger than necessary or perform additional, unnecessary operations, which decreases performance and increases energy consumption. In general, both this paper and the previous one agree that bloated software is a problem that should be tackled, and the problem is aggravated by the diminishing returns from Moore's law [12].

A field where energy efficiency is important is in mobile software development, given the importance of optimizing battery life. Dornauer et al. [10] present a survey on which of the different elements has the greater impact on energy consumption for mobile devices. They also show how the main research focus to improve energy efficiency in mobile devices has been the optimization of CPU cycles through efficient scheduling, although hardware improvements for wireless communication were also relevant.

In this same field, Cruz et al. [8] study the effect of best practices for mobile software performance on energy consumption. These guidelines are designed to improve the performance of applications, but they were not proposed with energy performance in mind. The paper performs an experimental study to find the energy impact of these guidelines and find that following these recommended practices leads to an improvement in energy performance. Later, Cruz et al. expanded their research into a catalog of energy patterns for mobile applications [7]. Here, they analyze which energy patterns are commonly applied in Android and iOS applications by collecting commits in open-source applications and building a catalog of the most effective energy patterns to apply to a mobile application.

4 Study

In the previous Chapters of this thesis, we show how different Docker base images can introduce energy inefficiencies due to the shared libraries included in them. Despite having some previous literature that uncovers these differences in performance, there are still no methods or tools to pinpoint the origin of energy inefficiencies. The objective of this Chapter is to present a methodology that can help explain some of the energy inefficiencies observed in previous work. We will apply this methodology to a case study using Redis to prove its effectiveness and uncover the reason behind some performance losses in Alpine and its C standard library.

The previous work by Tjiong et al. [32] does a good job of comparing the energy impact of different base images in Docker, providing a framework to run energy experiments, and giving general guidelines, like avoiding Alpine for certain workloads. The work also provides a hypothesis, suggesting that the main culprit is the C standard library implementation.

However, the thesis does not confirm the hypothesis, it does not focus on explaining the reasons for this behavior, and it does not provide techniques to locate the origin of the energy performance differences. Being able to explain why some dependencies are performing worse than others is a fundamental first step to fixing it and improving its performance. For example, in the case of Alpine, if we can explain and report which part of the distribution is performing worse, we can give Alpine developers the opportunity to fix it.

We start by replicating and expanding the Redis workload tested in previous work, to identify the potential root causes for the energy usage difference observed. We define a methodology to locate energy performance differences in shared libraries, which we will apply to Redis, but can be applied to other workloads. This process will be guided by our Research Questions:

RQ1 What is the difference between *glibc* and *musl* in terms of energy consumption?

RQ2 Can we use tracing to compare energy consumption differences in shared libraries?

RQ3 Can we verify the origin of energy consumption differences by recreating the workload behavior closely?

4.1. Experimental setup

4.1.1. Profiler selection

In this study, we isolate the measurements of energy to account only for the impact of the workload being tested. To do this, we use the software-based energy profiler *EnergiBridge* [24]. As we explained in Section 2.1.2, these profilers are not as accurate as physical power meters, but they are easier to use and provide accurate enough measurements of individual devices on the machine.

To isolate the measurements of the workload, we configure the energy profiler to measure the power usage of individual CPU cores. This way, we can assign an individual physical core to the workload and measure usage of this core, which limits the impact of other tasks on the core usage and energy consumption. Besides measuring total energy consumption over a period of time, we also need measurements of energy in fixed intervals of time, so we can calculate other useful data like power usage across the experiment.

For these reasons, we chose to use the profiler Energibridge provided by Sallou et al. [24]. The tool is multiplatform (Intel, AMD, Mac ARM) and easy to set up, without needing complex permissions or configurations that are platform-dependent. It makes energy measurements in fixed intervals by extracting information such as total energy used from the *Model-specific registers* (MSR) in AMD, performing minimal processing to affect energy measurements as little as possible.

4.1.2. Energy measurements

When performing energy experiments, there is a series of guidelines that must be followed to guarantee the quality of the results [5]. This is because, when measuring the energy consumption of a specific workload, there are multiple factors that can introduce bias in the measurements, like package temperature or operating system routines. Therefore, we need to follow certain steps to guarantee the integrity of results

First, we must warm up the machine. This is done because there is a correlation between CPU temperature and energy usage. A higher temperature increases the resistance of the internal components of the CPU, which leads to a larger energy consumption. Without proper warmup, the first runs of a workload will use a cooler CPU and will seem more energy-efficient than they truly are [5]. To achieve this, we first run each of the images of the workload without measuring any energy usage. Afterward, it is recommended to run a CPU-intensive task for at least 5 minutes. We do this using the Sysbench[1] prime number test on every thread of the CPU for around 6 minutes. This guarantees that the later measurements will not be affected by different CPU package temperatures.

The next step is to run the energy experiment properly. For this each of the Dockefiles is run 30 times in a randomized order, with a short pause between them, and energy usage data is measured for each of them. This way, the impact of possible variations introduced by unexpected variables, like services in the system, can be reduced. The recommended number of 30 comes from the Central Limit Theorem [5]. The accumulated data from the 30 runs should follow a normal distribution in which some runs were slightly more favorable than others due to unaccounted factors but together represent the general energy behavior of the workload.

The images are also run in a random order to avoid possible biases introduced by systematic and uniform variations in the setup. For example, if an experiment runs through dusk, there will be a room temperature decrease while the experiment is running, which will make the later runs more efficient than the first runs. By running the images in a random order, the effect of these possible variations will impact the different images equally.

Additionally, between each of the runs of an experiment, there is a short pause of 20 seconds. This pause is long enough to guarantee that there are no residual measurements from the previous run that leak to the next one, and it is also short enough to avoid the CPU cooling down and making the next run seem more efficient.

To make the energy data collected as accurate as possible, we isolate the workload to a single CPU physical core, and measure the energy used by that core. If the workload uses other containers (e.g. to measure a Redis server we need a client), they are run on other physical cores, so they do not affect the performance of the actual workload under test.

We use the *docker-energy* framework[2] [32] to set up and orchestrate the experiments. This framework uses Docker to set up and run the experiments, and it implements the previously mentioned guidelines for accurate energy measurements. It provides the ability to define different configurations and distributions for our experiments by setting them up in a container, using different base images. Another option would be to test distributions on native operating system installations. While this might provide more realistic results, having to install and uninstall different operating systems and packages is a challenging task, and difficult or costly to automate.

[1]`https://wiki.gentoo.org/wiki/Sysbench`

[2]`https://github.com/btjiong/docker-energy` retrieved on June 17 2024

Besides the previous guidelines about how to run the experiment, we also need to configure the machine itself to reduce noise in the energy measurements. We turn off any unnecessary services from the operating system and connect only the necessary hardware. This way, we minimize the impact of other routines on energy usage.

The experiments are run in a machine equipped with an AMD Ryzen 9 7900X processor, 64 GB of RAM, an MSI Geforce RTX 4090, and a 1000W power supply. The AMD CPU is selected because, unlike Intel CPUs, it can report the energy usage of individual cores, rather than reporting only the consumption of the whole CPU package. This CPU has an x86_64 architecture, and the machine runs Ubuntu 22.04.3 as the operating system, with Linux kernel version 6.2.0. Version 24.0.5 of Docker Engine is used for setting up and running the containers.

Additionally, the CPU frequency on the machine is fixed to 4700 MHz. When running experiments without a fixed CPU frequency, workloads will run at different paces depending on the demands of the processor, but using more or less energy. With a fixed CPU frequency, total runtime is more stable, and energy measurements are easier to compare.

4.2. RQ1. Comparison of C standard libraries through Redis

As explained in Chapter 3, previous work [32] shows evidence that Alpine is unexpectedly less energy efficient in some specific cases. Despite there being no concrete evidence, the main hypothesis we will propose is that there is a performance difference between the C standard library implementations used by Alpine and other Linux distributions. This hypothesis is sound, given that the experiments are run in Docker. In Docker, containers provide packaged software and dependencies (in the form of libraries and binaries). When containers need to access kernel functions, they rely on the host operating system kernel.

This means that when switching base images based on Linux distributions such as Ubuntu or Alpine, we are not switching the Linux kernel version to the one used by those distributions – we are only switching the binaries included with them. One of these binaries, and the most fundamental for running programs in a Linux system, is the C Standard Library or *libc*.

The API for *libc* is defined by the International Organization for Standardization, but different implementations of its functions exist. The most popular one is the GNU C library or *glibc*, which was created in 1987, and it is part of most Linux distributions, including Ubuntu. However, *glibc* has been criticized as bloated and slow by several engineers, with Linus Torvalds among them[3].

To face this problem, other alternative implementations of the C standard library started to appear. One of them is *musl*[4], started in 2011. According to its creators, *musl* is "lightweight, fast, simple, free, and strives to be correct in the sense of standards-conformance and safety". Because of these characteristics, it is the implementation included with Alpine. As a downside, *musl* is known to have some compatibility problems when running binaries compiled against *glibc*.

To answer RQ1 and verify the hypothesis that points out to the *libc* implementation, we will replicate the energy experiments performed for Redis. We chose Redis[5] as a focus for several reasons. First, it is a popular in-memory database written in C, which can also serve as a cache or message broker among other uses.

During previous energy experiments, Redis was one of the workloads that showed a significant energy usage difference between Alpine and Ubuntu. According to Redis documentation, the only dependencies needed to run Redis are a C compiler and a *libc* implementation. Given that the hypothesis we want to verify is that the difference comes from *libc* implementations, this workload is a good fit. If we compile under the same compiler version and configurations, the differences in performance observed between Alpine and Ubuntu should come from the standard library.

[3]Mail archive for the *glibc* project mailing list `http://ecos.sourceware.org/ml/libc-alpha/2002-01/msg00079.html` retrieved on June 17 2024

[4]*musl* home page `https://musl.libc.org/` retrieved on June 17 2024

[5]Redis home page `https://redis.io/docs/latest/get-started/` retrieved on June 17 2024

Table 4.1: Configurations for Redis energy experiments

Label	Distro	libc	Redis version	Allocator
ubuntupack	Ubuntu	*glibc*	6.0.16	jemalloc
ubuntu	Ubuntu	*glibc*	7.2.4	jemalloc
alpineglibc	Alpine	*glibc*	7.2.4	jemalloc
alpinejem	Alpine	*musl*	7.2.4	jemalloc
alpinemusl	Alpine	*musl*	7.2.4	musl allocator
alpinepack	Alpine	*musl*	7.0.15	musl allocator

4.2.1. Redis configurations

To verify the hypothesis, we replicate the Redis experiments from the previous work [32], and add some additional scenarios that allow us to compare *libc* implementations.

The workload used to test the energy usage of Redis is the *redis-benchmark*[6] command provided by the application itself. This benchmark simulates running a certain set of Redis commands done by N clients to a total of M requests. The benchmark executes these M requests for each of the tested commands. To execute these commands, Redis relies on some functions from the C standard library. We use a client container that executes this benchmark against the server, which we configure according to our scenarios and measure its energy usage.

To properly test if *libc* is the reason behind the observed energy differences, we need to freeze the rest of the Redis configuration as much as possible. Therefore, we will define different Redis setups to test the impact of different configurations on energy consumption.

The first setup we test is installing using the distribution's respective package managers (`apt install redis` in Ubuntu and `apk add redis` in Alpine) with the default configurations. Given the differences in versions, this comparison does not guarantee that the difference in energy consumption observed comes exclusively from *libc*. However, this is valuable for two reasons. First, this is a common installation method for most developers, so the difference in energy consumption observed is closer to what we would expect in a real-world setup. Second, this is the method used by previous work to test Redis energy consumption. It is possible that the energy differences they observed came from this discrepancy in versions, rather than from *libc*, so we need to test this case more thoroughly.

To define the next scenarios, we download a fixed version of the Redis source code (7.2.4) and compile it in the respective distributions keeping the same default configurations. These scenarios still present a discrepancy in terms of dependencies. Redis includes a custom memory allocator with its source code called *jemalloc*. With default configurations, this allocator is enabled by default in Ubuntu, but disabled in Alpine. Since this allocator can also affect energy efficiency, we define two Alpine scenarios with it enabled and disabled, to evaluate its impact on performance.

We define a final scenario to confirm the hypothesis that the C standard library implementation is the main reason for the energy difference. In this scenario, we configure Alpine and add the *glibc* binaries using a *glibc* binary package[7] for compatibility, and run Redis compiled against *glibc* instead of compiling it against *musl*. If we find a similar energy consumption between this version of Alpine and Ubuntu, we can confirm that the *libc* implementation is the main cause for the difference, instead of other included libraries or binaries.

Table 4.1 shows the different configurations that are tested. For each of our scenarios, we detail the Redis version used, the *libc* implementation, and the memory allocator used by Redis. *ubuntupack* and *alpinepack* are the versions installed through package managers, *ubuntu* and *alpinemusl* are the fixed versions compiled with default configurations, *alpinejem* is the Alpine version with the improved allocator, and *alpineglibc* is the Alpine version with the improved allocator using *glibc*.

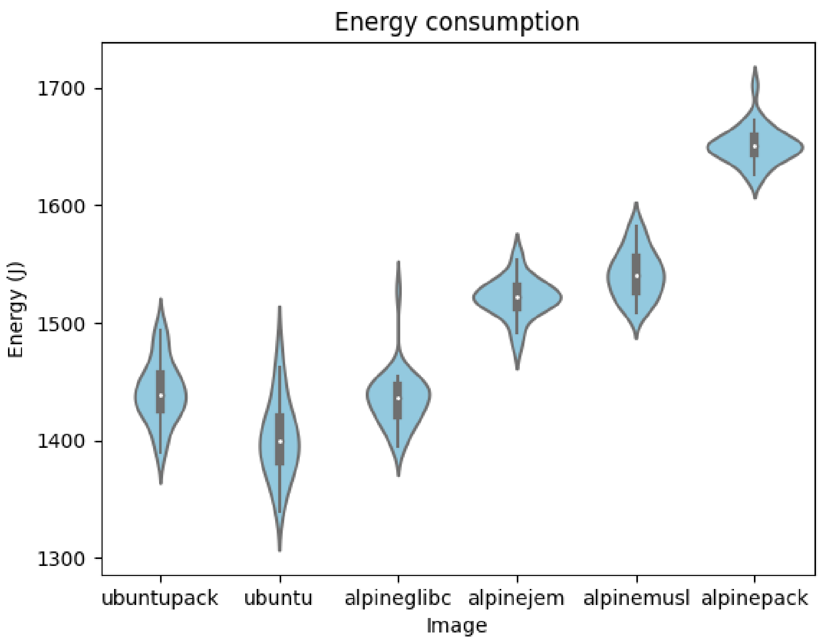


Figure 4.1: Energy consumption of Redis for the different configurations

Table 4.2: Average completion time and energy consumption for the different Redis configurations

Image	Time (s)	Energy (J)
ubuntupack	281.62	1441.42
ubuntu	295.27	1401.54
alpineglibc	284.16	1435.28
alpinejem	284.15	1521.59
alpinemusl	284.94	1541.85
alpinepack	286.53	1651.11

4.2.2. Data and Results

Figure 4.1 shows the violin plots for the total energy consumption of each image and Table 4.2 shows average energy usage. A violin plot is similar to a box plot in the sense that it aggregates the 30 measurements done, showing mean and quartiles. However, instead of having a box shape, it takes the form of the probability distribution of the data. We chose this because it lets us check easily if the experiments are correct. If the shape is symmetrical in the shape of a Gauss bell, we see that the results follow a normal distribution, as it should be for an energy experiment of this type.

The first thing we notice in this graph of Figure 4.1 is the energy difference between the packaged versions of Redis in Ubuntu and Alpine. The Alpine version uses around 14.5% more energy than the Ubuntu version while taking roughly the same time to complete. These are similar results to the one obtained in [32], which validates their experiments.

When fixing and using the latest version of Redis (*ubuntu* and *alpinemusl* images) instead of default package manager installations (*ubuntupack* and *alpinepack*), the energy gap gets smaller, but it is still there. Average consumption improves slightly in Ubuntu between both versions (2.76% better) and more considerably in Alpine (6.6% better). This might indicate that there were some performance improvements in Redis between versions. However, there is still a significant gap between Alpine and Ubuntu, with Alpine using around 10% more energy than Ubuntu. This indicates that the discrepancies in Redis versions from the original experiment are not the only reason for the energy performance

[6] *redis-benchmark* documentation `https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/` retrieved on June 17 2024

[7] *glibc* package for Alpine by S. Gerrand `https://github.com/sgerrand/alpine-pkg-glibc` retrieved on June 17 2024

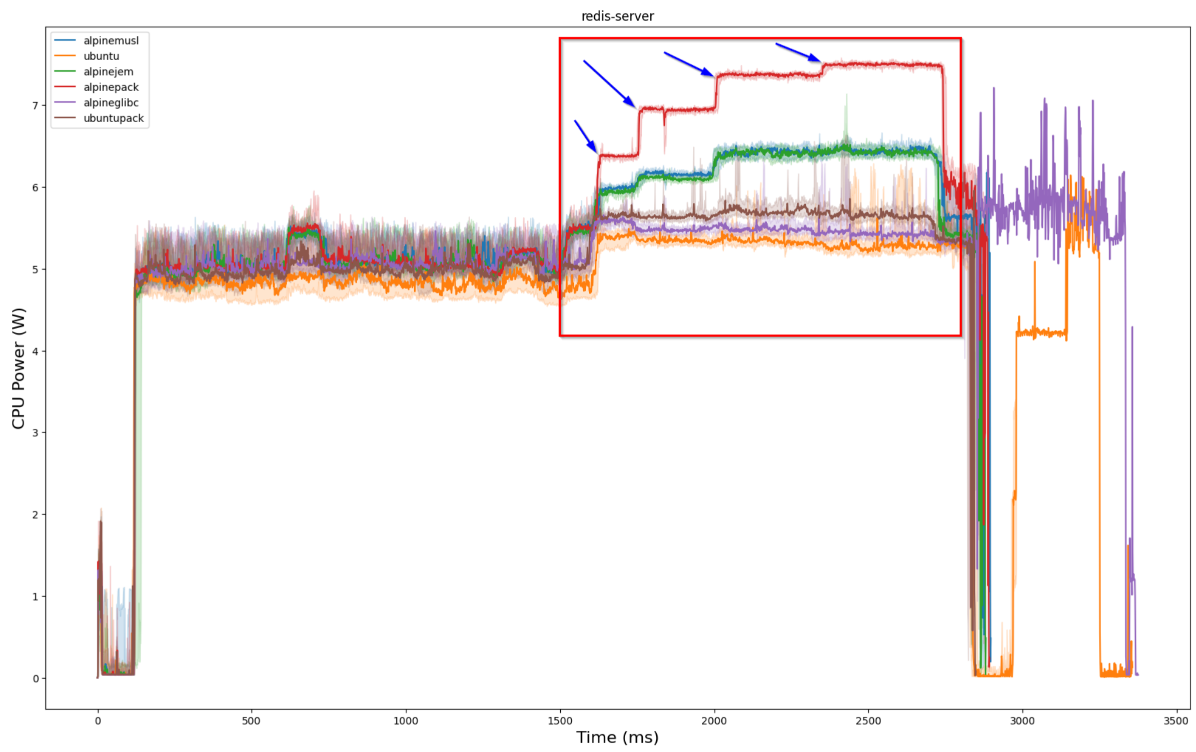


Figure 4.2: CPU Power usage (W) against runtime of Redis for the different configurations

difference observed.

We can also see how using the custom allocator (*alpinejem*), which was disabled in the Alpine packaged installation, slightly improves energy usage with respect to the standard library allocator from *alpinemusl* ($\sim 1\%$ better). However, there is still a significant performance difference of 8.6% between *alpinejem* and *ubuntu*, meaning that this configuration difference is also not the main reason for the energy performance difference.

Finally, we look at the energy consumption of Alpine with *glibc* (*alpineglibc* image). For this image, results showcase a similar performance between *alpineglibc* and the Ubuntu images. Except for some outliers, the image performs at around the same level as the old Ubuntu version, with an observed 0.4% improvement, and slightly worse than the most recent Ubuntu, using 2.4% more energy. However, the image performs considerably better than any of the Alpine images with *musl*.

We further inspect these differences by comparing the power consumption throughout the different executions of the benchmark. Figure 4.2 shows the median CPU power usage for the 30 runs of each image across time during the execution. In this image, we can see that power usage is similar in all experiments for most of the Redis benchmarks, hovering at around 5W. However, after a certain point, power usage goes up for all images (boxed region). However, this increase is not uniform, and it affects *musl* images the most. If we compare power usage between *ubuntu* and *alpinejem*, images where all configurations are equal except for *libc*, we obtain a difference of up to 1.17W in the larger gap (around 20.2% difference) This indicates that the main reason for the total energy difference is happening in this part.

We further investigate this pattern and manually check the logs. We can observe that the operations that are running at this point are LRANGE operations from Redis. The LRANGE command [8] is a simple instruction that, given a key and a start and end indices, recovers the elements between those two indices. In the benchmark run during the experiment, elements from the database have a default size of 3 bytes, and the LRANGE instruction is tested for 4 different numbers of elements: 100, 300, 500, and 600 elements.

[8]LRANGE docs `https://redis.io/docs/latest/commands/lrange` retrieved on June 17 2024

The power usage graph actually shows step increases (blue arrows) that correspond to the increase in the size of the responses from Redis. This correlation, and the fact that energy consumption is similar for the rest of the benchmark, suggest that power consumption depends on workload size. The energy demands of the functions behind the difference depend linearly either on argument size or the number of calls.

4.2.3. Conclusions

With the data uncovered with these experiments, we can answer RQ1:

RQ1: What is the difference between glibc and musl regarding energy consumption?

With our experiments, we not only managed to replicate the results from [32] and confirm the difference but with the additional scenarios we managed to confirm their proposed hypothesis: There is an actual difference in energy performance that seems to be introduced by the implementation of the C standard library. We can observe a difference of up to 8.6% in total energy consumption between *musl* and *glibc* image, and a difference of up to 20.2% in instant power in certain parts of the benchmark. This difference is lost when we modify Alpine to add *glibc*, which confirms that the cause is limited to the standard library.

Answer to RQ1. There is a substantial difference in both power usage and total energy consumption between Alpine and Ubuntu. We confirm that this difference has its origin in the different *libc* implementations provided by the distributions by introducing *glibc* into the Alpine distribution. With every other Redis configuration except for *libc* fixed, we observe a difference of 8.6% in total energy consumption and up to 20.2% in instant power usage.

4.3. RQ2. Digging deeper with tracing

Now that we have managed to confirm that there is a difference in performance between *glibc* and *musl*, we want to find out the reason behind the difference. In this case, we want to find the specific functions or features from *libc* that differ in implementation and performance.

To determine which functions are being used during the execution of the workload, we will use tracing. In software engineering, tracing is a debugging technique used to monitor the execution of a program by capturing and recording events like function calls [15]. We can use this technique to collect data of the *libc* functions being called during the workload and synchronize this data with the power measurements, finding which functions are more heavily used when the workload shows a larger power usage difference between configurations.

We will use this tracing data to know which *libc* functions are being used at different points of the benchmark. However, due to limitations on how the tracing technique works, and the size of the data it generates, we need to be intelligent in the analysis.

For this, we will design a full methodology that we will apply to Redis, but that can be generalized to other software, so other researchers and developers can find and study energy hotspots. This methodology will include instructions on how to run energy experiments, how to trace software, and automatically identify relevant checkpoints of the execution (avoiding the manual check we had to do in Redis with LRANGE). Finally, we can use this accumulated data to identify suspect functions that are used the most during execution, so we can further study and benchmark them.

To support the application of this methodology, a Jupyter Notebook with useful code for each of the steps is available in Github.[9]

4.3.1. Methodology

Energy tests

The first step to studying the performance of a workload and finding energy inefficiencies is to run energy experiments and locate relevant differences as we did with Redis. For a workload, multiple Dockerfiles

[9]`https://github.com/enriquebarba97/energy-hotspots` retrieved on June 17 2024

should be created, with varying dependencies. In our example, we changed the base image, but other dependencies can be changed. *uftrace* can trace calls to not only *libc* but to other shared libraries, as well as calls to Python libraries. This means that some other dependencies, like library versions, can be changed.

We adapt the existing *docker-energy* framework by Tjiong, B.[10] to print timestamps to the logs that we can use to synchronize energy data and tracing data. Our version of the framework is available on GitHub[11], which also includes the additional experiments ran during this study.

Tracing test

The next step is to execute a run of the workload while using tracing to record calls to shared libraries. As we explained, tracing is a technique to capture and record certain points during the execution of a program. In our case, we want to use it to record the calls made to *libc* functions, saving which functions are called, how many times, and the runtimes for each of the calls. With this data, we can find which are the most commonly used functions, or the functions that accumulate a longer runtime.

To apply this technique we use a tracing tool called *uftrace*[12]. *Uftrace* is a tool to trace and analyze the execution of a C/C++ program. It can track both user space functions and calls to dynamically linked shared libraries like *libc*. The tool reports each call to a function and the duration of said call. The program under study normally needs to be compiled using specific options like instrumentation, but the tool also provides the option of dynamically patching certain functions during runtime. This patching approach works well for simple shared libraries like the standard library, but it has more problems when trying to trace internal function calls of complex software.

We execute our workload once more while capturing tracing data with *uftrace*. Any base image can work for this run since library APIs should be consistent across Linux distributions. A Docker container with *uftrace* installed can be used for this. A Dockerfile that installs *uftrace* in Ubuntu is available with the Jupyter Notebook. The tracing data from *uftrace* is saved into a directory that can be recovered with volume binding or the `docker cp` command.[13]

To have timestamps with a consistent format in the tracing logs, you can use docker-energy also to run this experiment, or run using docker-compose and the *ts* tool from *moreutils*. Detailed instructions and commands are available in the Jupyter Notebook.

For *uftrace* to properly collect data, it is important that the workload shuts down gracefully. It must either stop by itself when it finishes running (e.g. returning from main or using exit if it is a C program), or it must shut down properly when receiving a SIGINT signal sent by Docker to shut down.

Another important recommendation is to reduce the size of the workload. The tracing data can get very large fast, and become unmanageable in the following tests. We can reduce the size of tests e.g. reduce the total number of requests to a server by some factor. If the previous energy tests revealed an energy inefficiency, one can also try running only the test cases that occur during that part of the workload.

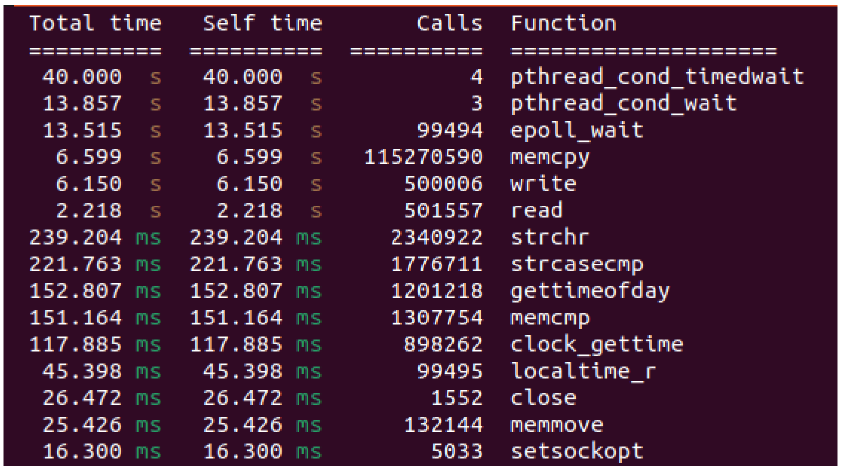
Analyze tracing data

The next step is to analyze the tracing data obtained from the workload. To do this, *uftrace* needs to be installed in your development machine. A fork from the original tool is available on Github[14]. This fork adds the option to print in nanoseconds without time units, which is necessary for later analysis.

A general summary of called functions and runtimes can be obtained with *uftrace report*. This will show an aggregate of all function calls, with a total count of calls as well as the total runtime for the functions. Figure 4.3 shows an example output of this command for the Redis benchmark.

This report will already provide a nice overview of which functions are running the most time, which can make a good suspect functions list to check. However, your workload might not be uniform and perform

[10]*docker-energy* framework by Tjiong `https://github.com/btjiong/docker-energy` retrieved on Jun 17, 2024
[11]Adapted version of *docker-energy* `https://github.com/enriquebarba97/docker-energy` retrieved on June 17, 2024
[12]*uftrace* repository by N. Kim `https://github.com/namhyung/uftrace` retrieved on June 17 2024
[13]`https://docs.docker.com/reference/cli/docker/container/cp/`
[14]Adapted version of *uftrace* `https://github.com/enriquebarba97/uftrace` retrieved on June 17 2024



```
 Total time   Self time       Calls  Function
 ==========  ==========  ==========  ====================
   40.000  s   40.000  s           4  pthread_cond_timedwait
   13.857  s   13.857  s           3  pthread_cond_wait
   13.515  s   13.515  s       99494  epoll_wait
    6.599  s    6.599  s   115270590  memcpy
    6.150  s    6.150  s      500006  write
    2.218  s    2.218  s      501557  read
  239.204 ms  239.204 ms     2340922  strchr
  221.763 ms  221.763 ms     1776711  strcasecmp
  152.807 ms  152.807 ms     1201218  gettimeofday
  151.164 ms  151.164 ms     1307754  memcmp
  117.885 ms  117.885 ms      898262  clock_gettime
   45.398 ms   45.398 ms       99495  localtime_r
   26.472 ms   26.472 ms        1552  close
   25.426 ms   25.426 ms      132144  memmove
   16.300 ms   16.300 ms        5033  setsockopt
```

Figure 4.3: Output of *uftrace report* for Redis

different operations at different points in time, showing different power usage and hotspot areas (like we observe in Redis). In this case, we need some deeper analysis, like function runtime distribution across time.

The command *uftrace replay* will show the sequence of function calls as they were captured during execution. Additional fields can be shown, like call duration or elapsed time. The result of this command with elapsed time and function duration fields is used for later analysis with Python. To get the appropriate time format for the timestamp, the fields *rawElapsed* and *rawDuration* should be used. More information is available in the Jupyter Notebook.

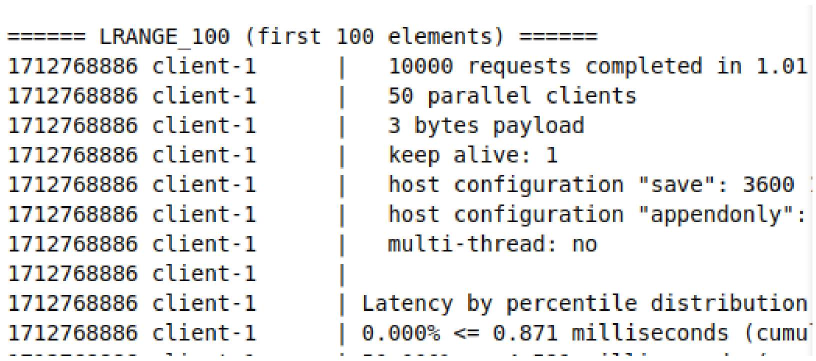
With this replay log, further temporal analysis of the tracing can be applied. However, the tracing utility introduces latency and larger energy requirements, making the benchmark take longer to complete. This means that we cannot run energy experiments and the tracing tool together because we would not get reliable energy data. We need to do a separate tracing run and relate the collected data with the previous energy data. This is not straightforward. Function calls take slightly longer to complete due to the latency, and, as suggested previously, the size of the benchmark should be reduced to avoid huge data files. This means the time points of the tracing run are not equivalent to the time points in the energy experiment, and we cannot directly associate a time region in the power measurement runs with the equivalent tracing region.

To solve this, we define a log alignment method using the logs provided by the workload under study to try to align logically equivalent points during execution. This means finding points during the execution where, no matter how long it took to complete, the same logical operations have been executed in both runs.

To relate the time points in energy data to time points in tracing data we find checkpoints in the logs. We define a checkpoint as a relevant and unique or almost unique line in the logs that marks the beginning or end of a section of the workload. In other words, checkpoints are progress marks for the benchmark. An example of a checkpoint from Redis is shown in Figure 4.4. Redis marks the end of each tested command with some statistics of the execution, preceded by a header that is unique in the logs.

We design a simple strategy to find checkpoints that assume no knowledge of the log structure, so we can reuse this for different workloads. The strategy has two steps:

- Clean log lines by removing numbers and measurement expressions. We remove dates, times, number of requests, etc. This way we don't consider lines that report certain measurements periodically as unique.
- We select as checkpoints lines that appear only once on the logs. Optionally, we can increase this to lines that appear less than X times if the log does not have enough variety. We also discard lines that are too close to the beginning and end of the file.



```
====== LRANGE_100 (first 100 elements) ======
1712768886 client-1      |   10000 requests completed in 1.01
1712768886 client-1      |   50 parallel clients
1712768886 client-1      |   3 bytes payload
1712768886 client-1      |   keep alive: 1
1712768886 client-1      |   host configuration "save": 3600
1712768886 client-1      |   host configuration "appendonly":
1712768886 client-1      |   multi-thread: no
1712768886 client-1      |
1712768886 client-1      | Latency by percentile distribution
1712768886 client-1      | 0.000% <= 0.871 milliseconds (cumu
```

Figure 4.4: Example of a checkpoint line in Redis logs

Once we find a set of checkpoints in energy and tracing logs, we can intersect both and treat the time regions between them as equivalent in terms of execution. It does not matter if the tracing run took more or less time to finish compared to the energy measurement run, the distribution of calls is the same in both. This way, we can aggregate the tracing data on this region, and show it together with the energy data in the form of a histogram.

Benchmark suspect functions

Once a set of functions likely to explain the difference in energy consumption is defined, we can isolate and individually test them with new benchmarks. Depending on the popularity of the library under test, it is possible to find some benchmarks already available online. These benchmarks usually test raw performance and runtime, but they can be used to measure energy. The benchmark should be long enough to conform to standards in energy testing. It is also beneficial if the benchmark for these individual functions replicates to an extent the way they are used in the original workload.

The benchmark can be run with docker-energy in the same way as the original workload, switching the same base images/dependencies as in the first experiment. This way, isolated parts of the dependencies are tested, helping to locate the parts where energy consumption is greater.

The energy and power usage plots can be plotted again. If the variations in energy consumption observed are similar to the ones observed in the original workload, then we know that the function under study is at least partially responsible for the original difference.

It is likely that the energy consumption difference is not exactly mirrored from the original experiment, and it might be smaller. This is because the benchmark will only represent a fraction of the original workload, and the original efficiency difference observed may come from the combination of several smaller differences.

Once the function is confirmed to be behind the difference, the last thing would be to figure out why and potentially fix it. The next steps here would potentially vary according to the library or technology in use. This methodology provides a way of partially pruning the dependency tree in search of the potential causes, without having to inspect all of the dependencies.

If we suspect that the function or functions found with this method are still high-level, and the actual cause of the difference could still be in some of the lower-level libraries, we could repeat the process, tracing the new benchmark and finding lower-level functions that could explain the inefficiencies.

Limitations to the methodology

The methodology presents some limitations due to the tools used and techniques applied. The first limitation is the large size of the tracing data. The methodology gives recommendations to avoid the tracing data becoming unmanageable, but if the workload depends on too many libraries, you could end up with too large data files.

Another limitation is that to apply the normalization methods based on checkpoints in the logs, the workload needs to have decently formatted logs, and there should be some variety in the features tested or run during the workload.

Finally, the tool selected *uftrace* is limited to tracing C, C++, Rust, and Python programs. This means that tracing analysis can only be done on programs of this kind. With other technologies, energy experiments can be run. If the workload shows power usage differences only in certain regions (like the Redis example), you could pinpoint the differences to a certain part of your source code just by looking at the logs. Still, it would not be possible to pinpoint it to underlying libraries. This limitation could be circumvented by using a different tool that supports other languages, and this methodology would still be valid.

4.3.2. Redis data and results

We apply the previous methodology to our Redis benchmark. This workload is a good fit for this methodology, since we have a specific region of the benchmark that shows a higher energy difference, and we want to figure out what functions are being called in that region of interest.

First, we execute the traced run to collect the data as defined in the methodology and obtain a first summary of the whole benchmark. As explained before, we reduce the size of the workload to have manageable data sizes. To do this, we reduce the number of requests that the benchmark does for each of the commands, from $1\,000\,000$ to $10\,000$.

Table 4.3 shows the summary of the trace, with the top 10 functions that accumulate the longest runtime across the whole benchmark. We see that the most called function by a wide margin is `memcpy`. However, the software spends a similar time in the `write` and `read` functions, with a much lower number of calls.

Table 4.3: Summary of *libc* function calls and runtimes for Redis

Function	Runtime [ns]	Calls	Runtime [%]	Call [%]
`memcpy`	6594371865	115270590	39.562145	92.637077
`write`	6100079082	500006	36.596694	0.401829
`read`	2204989695	501557	13.228572	0.403076
`epoll_wait`	701065124	99494	4.205956	0.079958
`strchr`	238221535	2340922	1.429182	1.881279
`strcasecmp`	221697527	1776711	1.330048	1.427852
`gettimeofday`	152283682	1201218	0.913608	0.965357
`memcmp`	150901874	1307754	0.905318	1.050975
`clock_gettime`	117885050	898262	0.707237	0.721887
`localtime_r`	45398704	99495	0.272364	0.079959

Next, we run the log alignment method to find checkpoints and be able to accurately plot a histogram of function runtimes, so we can study the LRANGE portion of the benchmark with more detail. Figure 4.5 shows the result of applying this process to the Redis experiment, and Figure 4.6 shows the power usage of Redis for the two relevant Ubuntu and Alpine images, with the same checkpoints included. Now, we can easily look at them side by side to find differences in runtimes of the functions called, despite the two runs taking different times to complete.

In the figure, we can see how, except for the first region, which comprises server startup, the runtime distributions of the different functions look more or less uniform. However, when we get into the LRANGE portion of the benchmark, we can see how the total runtime of the `memcpy` function starts to go up drastically, compared to the other functions, taking up to 2 seconds of the total runtime of the LRANGE.

The purpose of this function is to copy a certain number of bytes from one pointer to another pointer in memory. Researching this function further, we figure out that the *glibc* implementation of the function contains some assembly code that can be inlined in place of the function[15] to improve efficiency, while *musl* does not[16]. This difference in *memcpy* implementation between *glibc* and *musl* could explain the

[15] *glibc* `memcpy` implementation `https://github.com/lattera/glibc/blob/master/string/memcpy.c` retrieved on June 17 2024

[16] *musl* `memcpy` implementation `https://github.com/esmil/musl/blob/master/src/string/memcpy.c` retrieved on June 17 2024

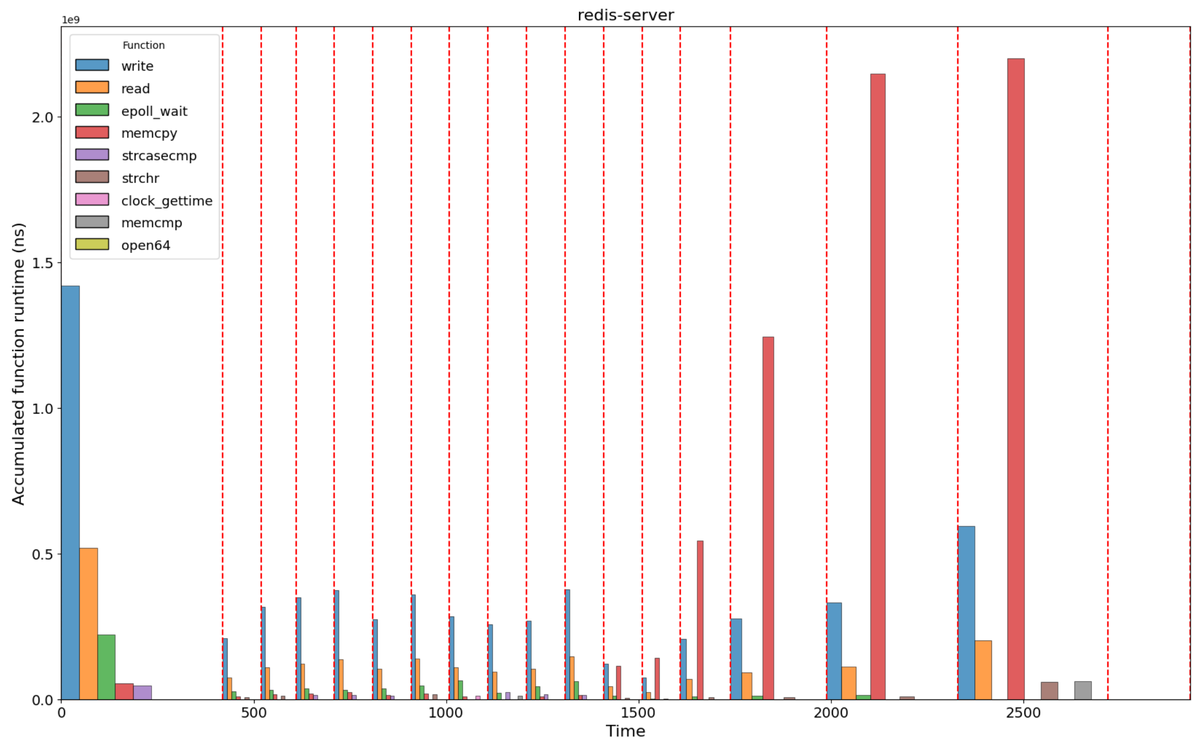


Figure 4.5: Histogram showing accumulated function runtime in each region between checkpoints

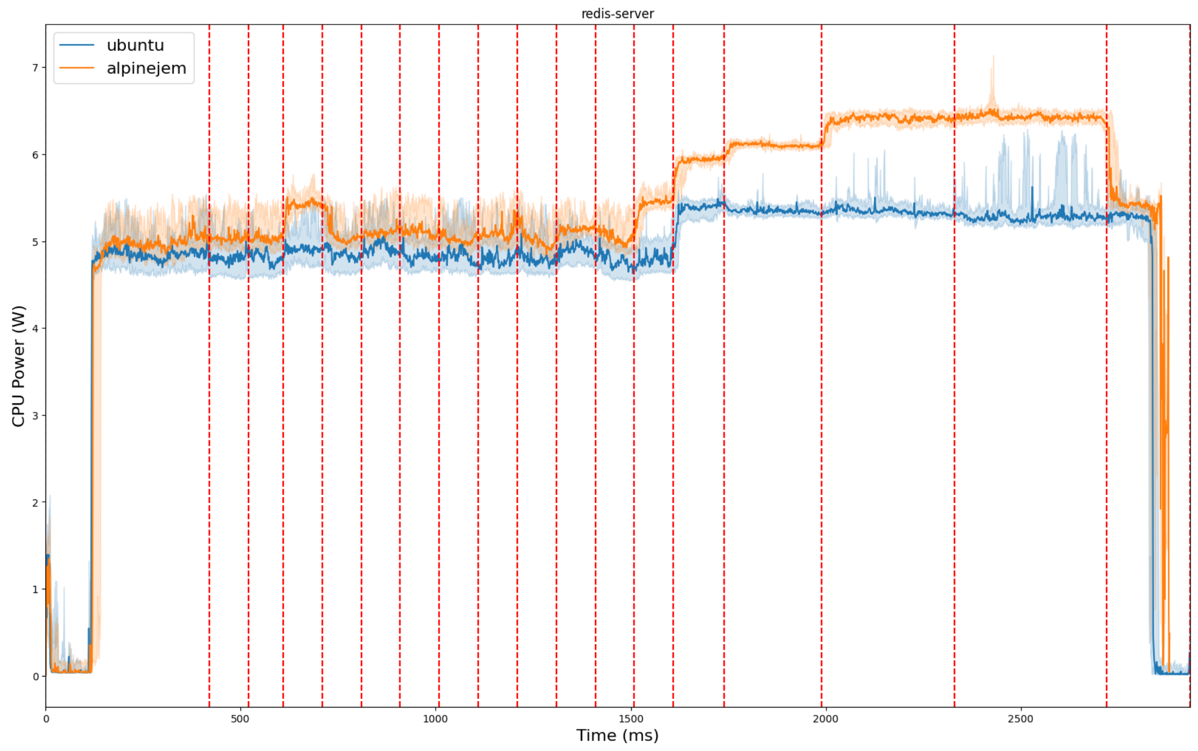


Figure 4.6: Power usage of Redis for Ubuntu and Alpine, with the obtained checkpoints

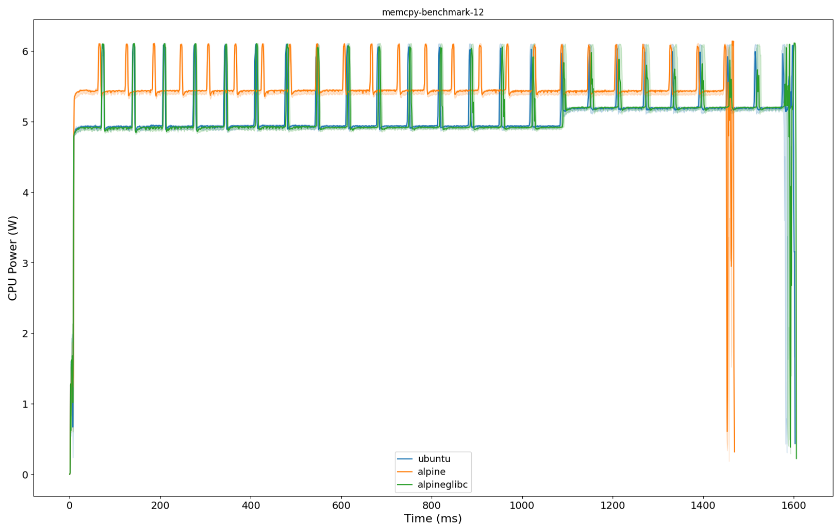


Figure 4.7: Power usage of the *memcpy* benchmark with 12 GB

difference in energy consumption, considering the prevalence of this function in the Redis workload. Therefore, this is the first suspect function we will benchmark further.

By carefully studying Redis source code, we can see how Redis makes use of this function. When recovering an element from some list from memory, Redis uses `memcpy` to move that element to a memory buffer that is later sent to the client through TCP. Most of the functions tested in the benchmark only recover one element. However, LRANGE has multiple tests that recover up to 600 elements, which is done through an iterator that copies the elements to the buffer one by one. This means that a single LRANGE request has up to 600 more `memcpy` calls than other commands from the benchmark, increasing the usage of the function.

4.3.3. Benchmarking `memcpy`

To benchmark the *memcpy* function, we design and run several energy experiments. We do so, according to source code review and consulting experts, *glibc* defines different processor-specific assembly code for this function to handle certain edge cases in the most efficient way possible, based on what is known about the usage of the function in compile time. Musl only provides a generic C implementation to cover all cases.

Now that we have concluded that the difference in performance is introduced by *libc*, the base images of these experiments will be limited to Ubuntu, Alpine, and Alpine with *glibc*. This will provide more visual clarity in the results. The first experiment consists of a simple *memcpy* benchmark written in C, adapted from an existing benchmark[17]. This benchmark allocates a random memory buffer of 12 GB and performs the following operations:

- Copy all 12 GB to another point in memory with a single `memcpy` call
- Copy all 12 GB with multithreading, to measure multithreaded performance.
- Copy all 12 GB in small sequential batches using 2^{20} calls to *memcpy*. This is more similar to the behavior observed in Redis

Each of the operations is repeated 8 times to elongate the duration of the benchmark and be able to take proper energy measurements. Before starting a repetition the buffers get allocated with `malloc` and they are freed after each repetition.

Figure 4.7 shows the power usage of this experiment for Alpine and Ubuntu, and table 4.4 shows average time and consumption. We can recognize each repetition of an operation by an energy spike that is produced when freeing and allocating memory. We can appreciate a slight difference in power

[17] *Memcpy* benchmark by Rodorigo L. avaialable on Github: `https://gist.github.com/lrodorigo/280bd4db453210cee6e1610648d937a9` retrieved on 17 June, 2024

Table 4.4: Average completion time and energy consumption for the `memcpy` benchmark with 12 GB

Image	Time (s)	Energy (J)
alpine	145.39	783.48
ubuntu	158.38	795.13
alpineglibc	158.81	793.55

usage, with Alpine using slightly more power than Ubuntu, with a difference of around 0.5W or 9.5% for most of the benchmark. At the final step of the benchmark, where we divide the `memcpy` into 2^20 sequential calls, energy usage for the *glibc* based images goes up a bit, while Alpine usage remains the same.

Despite the difference in power usage, this behavior is not exactly the same as the one observed in Redis. As observed in the table, the higher power usage in Alpine is translated into a faster runtime and the Alpine image ends up taking slightly less overall energy to complete the task.

This previous experiment does not confirm that the reason for the energy consumption difference comes from `memcpy`. However, as explained before, there is a multitude of `memcpy` edge cases to cover. Concretely for Redis, as we explained before, it uses `memcpy` to move elements 1 by 1. If we study the benchmark default configuration, we can see that the default data type and size used are strings of size 3. This means that most of `memcpy` calls in Redis and LRANGE contain 4 bytes (3 bytes of data + 1 separator that is used in the format for the response).

This detail is important because of memory alignment. In 64-bit architecture, data in memory is required to be 8-byte aligned [2]. This means that moving multiples of 8 bytes is usually less expensive than moving less than 8 bytes since the latter requires additional instructions to guarantee proper alignment. In our first experiment, we copied 12 GB ($2^{30} \cdot 12$) with a single call, divided into 8 multithreaded calls ($2^{27} \cdot 12$ size for each) and in 2^{20} calls ($2^{10} \cdot 12$ size). All of these calls are multiples of 8 bytes, so it is easy to process while keeping alignment.

Hence, we design a new experiment to mimic Redis' usage of the function. In this experiment, we initialize a memory buffer *destination* of 3000 bytes and perform the `memcpy` operation with small sized elements. For each of the sizes defined in the LRANGE benchmark (100, 300, 500 and 600), we copy that number of elements of size 4 bytes, and repeat for a large number of times (40 000 000) to simulate the high number of requests in Redis and to have a long enough benchmark so we can measure the energy accurately.

The dummy element we use for this experiment is "VKX,", the same dummy data that Redis uses. We provide this data to `memcpy` in two different ways: in one of the experiments we initialize a second buffer of the same size *source* with the literal repeated over and over. Then, we call `memcpy` using two moving pointers, one to *destination* and another to *source*. In the second experiment, we use a cached literal, and copy it over and over until filling the requested number of elements, with a single moving pointer to the destination buffer. While these two experiments might look functionally the same, they change the information that is known on compile time, which might change the version of `memcpy` that is used for *glibc*.

Table 4.5: Average completion time and energy consumption for the `memcpy` benchmark from memory to memory

Image	Time (s)	Energy (J)
alpine	95.05	1065.34
ubuntu	95.24	1075.88
alpineglibc	94.96	1075.36

Figure 4.8 shows power usage over time of the memory-to-memory `memcpy` experiment, and table 4.5 shows average runtimes and energy usage. In this experiment, we can appreciate a slightly different behavior from the first experiment. We removed the repeating allocations and freeing of memory to have a more uniform measurement of `memcpy`. In this benchmark, we observe how the power usage of all images escalates to the approximately 11 W. We can also see how, in this case, the *glibc* images

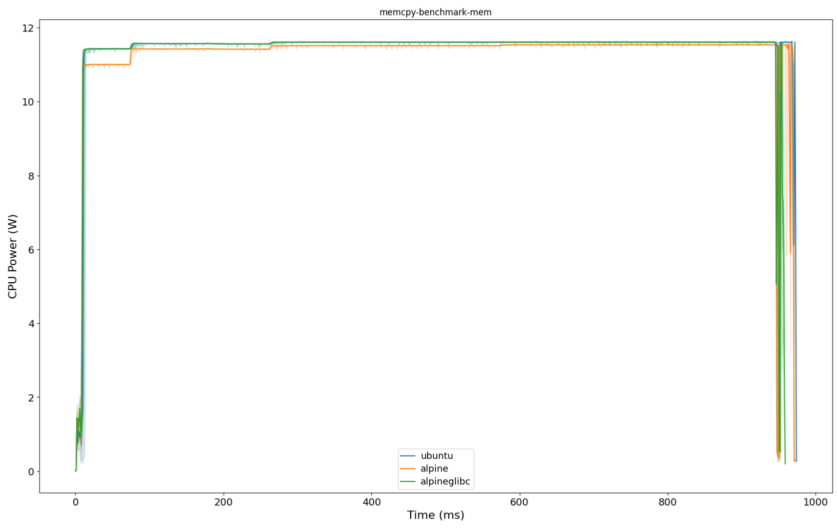


Figure 4.8: Power usage against time of the *memcpy* benchmark from memory to memory

are using more power than *musl* and, unlike the previous experiment, this is not translated to a shorter runtime. Indeed, now all images take a similar time to complete, and the Alpine image uses $10J$ less to complete the task. We can also appreciate small step-ups in power usage as the number of elements to copy goes up. This is similar to the behavior observed in Redis.

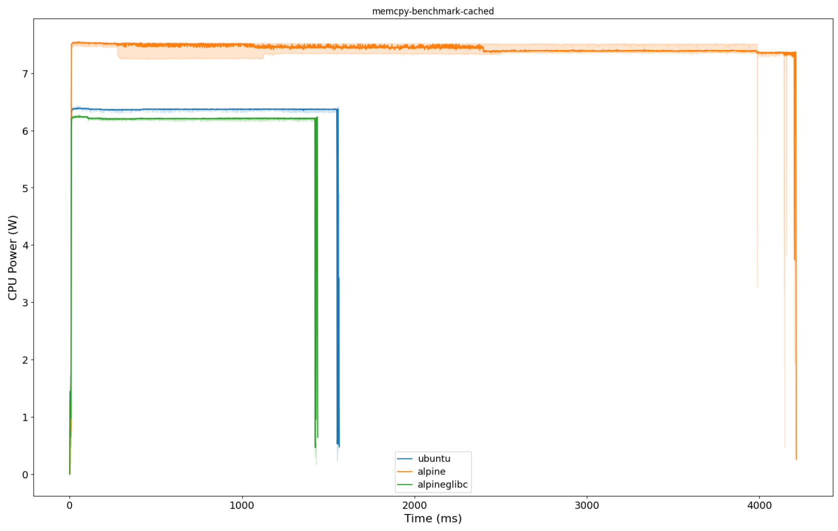


Figure 4.9: Power usage against of the *memcpy* benchmark from cached to memory

Figure 4.9 shows the power usage of the cached experiment, and Table 4.6 compares the runtime and total energy consumption of this experiment. For this benchmark, we obtain completely different results from the previous two benchmarks. Here, *musl* is much more power hungry than *glibc*, with a difference of around $1.1W$ or 15.8% difference, closer to the difference observed in Redis. What is more, this difference does not translate into more performance, with Alpine taking almost $3X$ more time to complete the task, resulting in a much higher total consumption.

These benchmarks show that the behavior of `memcpy` can vary wildly depending on what information is available in compile time. We also notice how some of the benchmarks move in different power usage ranges to Redis. This can be explained because Redis has network communication features. In this benchmark, all `memcpy` calls are done without pause. However, in Redis, once the `memcpy` calls for a request are finished, the response has to be sent through TCP, an I/O operation in which the CPU does not have to be used as heavily.

Table 4.6: Average completion time and energy consumption for the `memcpy` benchmark from cached to memory

Image	Time (s)	Energy (J)
alpine	406.10	2977.45
ubuntu	155.37	972.76
alpineglibc	142.53	869.53

4.3.4. Conclusions

From the results of the benchmarks and tests performed for Redis, as well as the experiments used to study the behavior of `memcpy` in different scenarios, we can answer our second research question.

RQ2: Can we use tracing to compare energy consumption differences in shared libraries?

By running the previous experiments, not only did we confirm that we can use tracing to locate the origin of energy inefficiencies, but we also managed to actually locate some energy inefficiencies and differences between Alpine/*musl* and Ubuntu/*glibc* with respect to the energy performance of the `memcpy` function. While it is known among the Linux and Alpine communities that there are performance differences for certain *musl* functions, we conclude that there is a lack of awareness of the impact on energy consumption that these differences can make.

Additionally, we propose a methodology to measure and identify energy inefficiencies in software and trace the inefficiencies to the specific dependencies that are causing them with a log alignment method. We also provide a pipeline to apply this methodology and properly analyze the data in the form of a Jupyter Notebook, so that other researchers and developers can study their own workloads.

Answer to RQ2. We defined a methodology around tracing designed to locate and explain energy inefficiencies, and successfully applied it to Redis to narrow down the energy consumption difference observed to the `memcpy` function. We benchmarked this function, finding that usage scenarios in which memory alignment has to be enforced expose performance differences between *musl* and *glibc*.

4.4. RQ3. Recreating Redis behavior

Through different benchmarks, we confirm that, in some cases, the `memcpy` function shows a significant difference in performance between *musl* and *glibc*, depending on the optimizations available and the information known on compile time. However, our results do not reflect the energy behavior observed in Redis, since we don't observe similar gaps in energy usage. We want to obtain more concrete results and recreate the Redis observed inefficiencies as much as we can. In this section, we improve our benchmark, so that it reflects more closely the frequency in which Redis uses the `memcpy` function.

The first step is to confirm if there is some other function that might be affecting performance in some way. In the tracing data we obtained for Redis, we observed that `memcpy` was the function with the most runtime, and helped identify this function as a point of inefficiency in *libc*. However, as a close second, we observe the function `write`. We want to investigate if this function introduces any difference in energy consumption when using *glibc* or *musl*.

The `write` function is used to write a certain amount of bytes from a memory pointer into a file descriptor. However, Redis is an in-memory database. These calls to the `write` function are not used to write into the file system but to write responses back to the clients through TCP sockets, which are treated as file descriptors on Linux. In a similar way, the third function with more runtime, `read`, is used to read incoming data from a TCP socket. The next function in the list, `epoll_wait` serves to monitor the writing and reading events taking place in file descriptors. In this case, it monitors the open TCP sockets to see if there is incoming data. Therefore, these three functions represent the network usage of the workload.

To test if this TCP communication creates a difference in power usage between *glibc* and *musl*, we create a new energy experiment to test it. For this experiment, we build a basic TCP server in C that

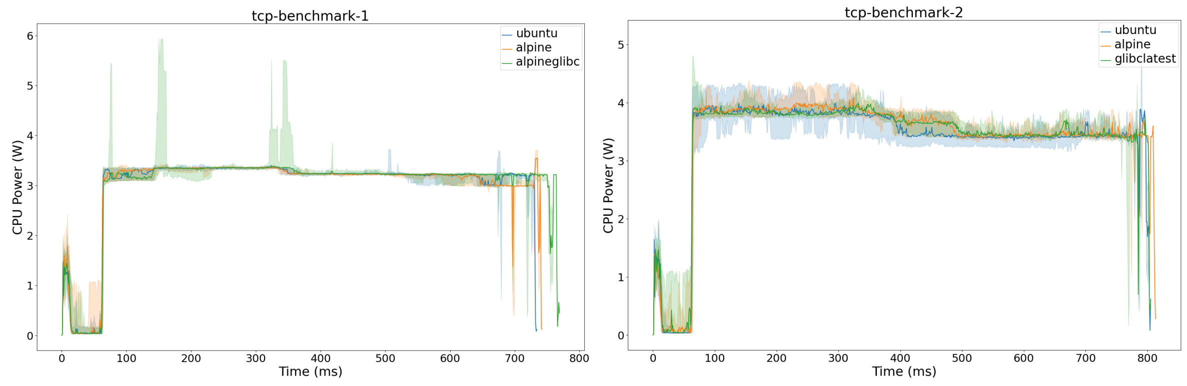


Figure 4.10: TCP server median power consumption for 1 and 2 clients against time

simulates the network behavior of Redis during the benchmark. Concretely we simulate the LRANGE part of the benchmark with the following behaviour:

1. A client sends an integer *size* as a request (e.g., "100")
2. The server returns *size*4* bytes (3 data bytes as the default Redis benchmark + 1 separator)

We also try to avoid using any *libc* function except for the strictly necessary for network communication, so we do not accidentally introduce a performance difference through one of these functions that taints the energy data. For this, we allocate and fill a memory buffer with dummy data at startup, and the server will just send the content of this buffer.

Similarly to Redis, the server can accept multiple clients and monitors if their sockets are ready to read or write using the epoll API provided by the kernel.

With this setup, we run a new energy benchmark with one and two simultaneous clients connecting to the server. To emulate the Redis benchmark behavior, one million requests are made among all clients for each of the sizes 100, 300, 500, and 600.

Table 4.7: Average completion time and energy consumption for the TCP benchmarks

Image	1 client		2 clients	
	Time (s)	**Energy (J)**	**Time (s)**	**Energy (J)**
alpine	67.25	211.36	73.34	255.37
ubuntu	67.59	215.16	72.88	249.37
alpineglibc	68.08	221.02	74.90	261.47

Figure 4.10 shows the median power usage for the TCP server in Ubuntu, Alpine, and Alpine with *glibc*, and Table 4.7 shows average total time and consumption. From these graphs, we can observe that there is no significant difference in performance between the 3 images. There is a slight difference for Alpine with *glibc* which tends to take slightly more time to complete, but its power usage is similar to the other two images.

This behavior is expected if we consider the architecture of Docker containers. As explained in section 2.2, a Docker container only brings the filesystem and shared binaries from a distribution, but uses the host kernel for any necessary system calls. In this case, the library calls `write`, `read` and `epoll_wait`, as well as the calls to open and listen to TCP sockets, are system calls and do not contain any complex or expensive logic. They are just wrappers whose only function is to place the necessary arguments in their correspondent registers and call the kernel to perform the function for them. Therefore, unlike with `memcpy`, all three images are actually using the same code.

We can derive a couple of conclusions from these results. First, when analyzing the functions used by a workload in a Docker environment, we can ignore those that are known system calls, since they should not change between scenarios, and cannot be the reason for the energy differences.

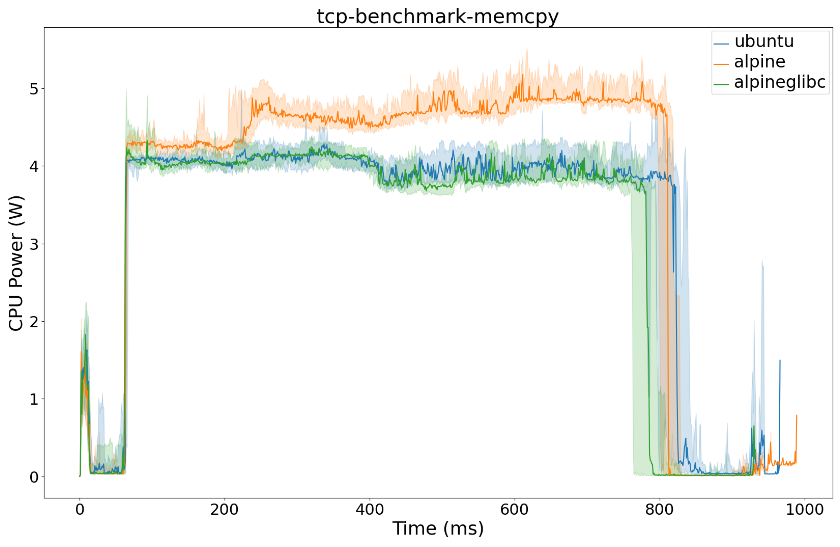


Figure 4.11: TCP server with `memcpy` median power usage for 2 clients

A second conclusion we can make is that, since system calls do not change between images, we can create more complex and accurate benchmarks for other *libc* functions by using only that function and known system calls. In this case, we will expand the TCP benchmark to create a new test scenario for `memcpy`. Instead of returning the pre-initialized memory buffer to the client, we will call `memcpy` to fill the buffer as Redis does, once to add each of the requested elements and send the result through TCP.

Table 4.8: Average completion time and energy consumption for the TCP benchmark with `memcpy`

Image	Time (s)	Energy (J)
alpine	91.34	352.64
ubuntu	89.83	312.18
alpineglibc	88.25	296.04

Figure 4.11 shows the median power usage of this experiment and Table 4.8 shows average times and energy. Finally, we can see that in this benchmark, the behavior observed is similar to the power usage and behavior from Redis. The *alpine* takes slightly longer to complete and uses around 12% more energy than Ubuntu while using almost $1W$ or 13% more power.

4.4.1. Conclusions

From the results obtained for the benchmarks with TCP communications, we can answer our final research question.

RQ3: Can we verify the origin of energy consumption differences by recreating the workload behavior closely?

We ran a benchmark involving only the functions related to TCP communication. With this benchmark, we verified that C library functions that perform system calls are not affected by base image selection. This was to be expected, based on the Docker containers architecture, but we managed to confirm the behavior experimentally.

This behavior allowed us to define a strategy to define more complex benchmarks. We can recreate the behavior of a workload as closely as we can if we only use functions that are system calls, plus the function we want to study. With this, we created a benchmark that combined `memcpy` with TCP communication that emulated Redis behavior more closely. With this benchmark, we confirmed that `memcpy` is the main cause for the energy inefficiency in Redis between Alpine and Ubuntu.

Answer to RQ3. We managed to exploit the behavior of system calls in Docker to create a more accurate benchmark for `memcpy` when used in Redis, observing a difference of 12% in energy usage, similar to the difference observed in Redis

5 Discussion

5.1. Implications

The results obtained in this case study indicate that there is a significant difference in energy performance between *glibc* and *musl*. Concretely, we found that the `memcpy` function is one of the functions that explains this difference in energy performance, which has a significant impact on Redis performance when deployed in Alpine compared to Ubuntu.

As we explained, the *glibc* implementation of `memcpy` is generally better than the implementation from *musl* because, according to consulted experts and community forums, *glibc* implements architecture-specific assembly code to cover certain edge cases in a more performant way than with the generic implementation. On the other hand, *musl* tries to avoid this to have a cleaner and less bloated codebase.

Our results shed light on these inefficiencies for both Alpine maintainers and general developers. For the first group, it is true that the difference in performance between both implementations is a known fact among the C standard library and Linux development community. However, this performance is measured with benchmarks that are limited to collecting raw performance data, such as total runtime or data transfer speed. The consequences of these differences in implementation on energy usage have never been measured, and they can be relevant for Alpine and *musl* maintainers to consider if it can be worth adding more coverage of edge cases to improve energy performance.

For any other developers that have to work with Docker, our results provide an insight into the true cost that the base image can have. The `memcpy` and other performance differences are common knowledge in the C community [33, 13], as well as other incompatibilities. Due to *glibc*'s popularity, most binaries in package managers are dynamically compiled against this implementation. When using these binaries in *musl* systems like Alpine, problems in performance might arise, as is the case for some Python libraries [33]. However, we could not find any official documentation that properly informs about these decisions or differences between C standard libraries. Considering how almost everything in a Linux distribution has to go through *libc* at some point, this case study is a good example for developers to understand how it can have an impact, even if they are not working directly with *libc* or even in C.

Based on the results observed in previous work [32], where workloads of a different nature to Redis also show a significant energy performance difference between Alpine and Ubuntu, it is very likely that the performance difference between libraries is not limited only to `memcpy`, but also to other functions. Appendix A shows another case study with the PostgreSQL. We apply the methodology and find that the most used function that is not a system call is `memset`. We did not create and perform benchmarks for this function, but it is likely that its energy performance is worse in *musl* than in *glibc*.

In short, our study has unveiled significant energy inefficiencies in *musl* which we think should be taken into account by the Alpine team. From the Alpine page: “Alpine Linux is an independent, non-commercial, general purpose Linux distribution designed for power users who appreciate security, simplicity and **resource efficiency**."[1] We believe that, as part of being resource efficient, the energy efficiency of the distribution should also be taken into account. Considering that Alpine is mainly used for Docker containers, and how the layered filesystem of Docker already optimizes storage, requiring little extra resources for more containers, some image size could be sacrificed for better energy performance.

Another contribution of this research is the methodology to find functions that are introducing energy inefficiencies. We applied this methodology to the Redis case study, finding inefficiencies in `memcpy`, but it can also be applied to other contexts. There are many tools and methods to debug and fix different problems on software, but the strategies and tools to measure the energy efficiency of your software are almost non-existent. This methodology aims to be the first step towards easing the research and fixing of energy inefficiencies.

Developers can use this technique to study different dependency options for their applications. A developer can test energy performance in different base images or library versions and, based on which functions are required and used the most by their software, select the dependencies that show better performance. It can also be used to study the performance of their own libraries.

The technique is also valuable for Linux distribution maintainers, and especially to maintainers of well-used Docker images. The methodology can serve to compare the performance among different versions of your image, and the effect of the low-level system libraries included with it.

The methodology can also serve to monitor and fix energy regressions between versions. An energy regression is defined as a worsened energy performance between a newer version of a piece of software and its older version. For example, while working on this thesis and contacting experts, someone from Green Coding[2] informed us about an energy regression between two versions of Alpine[3], where a CPU stress test performs 1% slower using 5% more energy. Our methodology could be applied to this test to find out which functions are being used during the workload and potentially locate the origin of the energy regression.

This thesis also provided energy benchmarks for `memcpy` and a TCP server. These benchmarks show how classic benchmarks, usually oriented towards measuring runtime performance, can be adapted and used to measure energy performance. However, when applying our methodology to research and study specific functions used in a workload, the benchmarks might not properly represent the true usage of the function.

For example, for `memcpy`, the benchmarks found online usually focus on measuring raw performance. They make continuous use of the function for large amounts of data, which is the best way of measuring raw performance and is strenuous for the CPU and memory. However, a complex application like Redis will also use other functions and logic besides a single function, usually combined with I/O or network operations, in which the CPU does not have high usage. This means that the energy measurements taken for a benchmark of this type might be misleading, and not representative of the energy impact of the function in a more complete workload.

To palliate this, we also show how, when working on energy benchmarks for Docker, the different binaries provided by an image do not have an impact on the performance of system calls. This allows for the build of a more "mixed" benchmark for a function. We propose that a more accurate method of measuring the energy impact of a function inside a workload is to create a suitable benchmark that is comparable to the original workload. To do this without unknowingly impacting the energy usage through other functions system calls can be used since they do not change between Docker containers. This gives an ample range of functionalities to recreate, like TCP communication. This kind of benchmark should yield more accurate and truthful results about the energy inefficiencies of a particular function for a specific environment or workload type.

[1] `https://alpinelinux.org/about/`
[2] `https://www.green-coding.io/`
[3] `https://github.com/alpinelinux/docker-alpine/issues/385`

5.2. Threats to validity

There are several threats to the validity of the results and conclusions obtained from this thesis that need to be addressed. First, all of these experiments have been conducted in a single machine with an AMD x86_64 CPU. Given the low-level implementation of the *libc* functions, especially for the architecture-specific assembly code that *glibc* contains, it is possible that the energy difference results obtained for `memcpy` do not translate directly to other CPU architectures. Still, given the relevance of the x86_64 architecture in both personal and cloud computing, these still are strong results to consider.

One of the main contributions of this thesis is the methodology to analyze the function usage of a workload and locate potential reasons for energy inefficiencies. We applied this methodology to our case study of Redis, and successfully found the reason for the inefficiencies. However, we did not have time to perform a proper validation process of the methodology. Ideally, we would test this methodology with other workloads that show a difference in energy consumption and verify if we can also find the inefficient functions in these workloads. Additionally, we only tested this for software written in C. While defining our methodology, we say that it can work with C, C++, Rust, and Python programs, since these are the technologies supported by *uftrace*. Given how the tracing tool has worked well for us in our case study, we are confident that it can be generalizable to other workloads (limited to the previous technologies). However, we cannot make this affirmation with total certainty without more validation.

Finally, the alignment of time between energy and tracing results relies heavily on the log quality and variety of the workload under study. If the workload does not have a comprehensive breakdown of the steps taking place during execution, our simple alignment method based on relevant lines will not be able to find many normalization bins. While it is safe to assume a certain minimum for the quality of the logs, some applications might not follow proper practices. For these cases, we added the possibility of normalizing along the entire duration of the workload. Suppose the workload performs a single type of logic, and the power usage difference observed is relatively constant. In that case, this method should be good enough, as showcased in Appendix A with PostgreSQL. Despite this, a more complex method to analyze and align the logs could be beneficial. Something similar to the work by Tan et al. [31] where they analyze logs as state machines could work well for this method.

6 Conclusion

The Linux kernel and the C standard library specification were created more than 30 years ago. Since then, different technologies and libraries have been built on top of it, with the objective of abstracting and facilitating the development of new tools and programs. This combination of languages and libraries adds unexpected interactions and inefficiencies, which are difficult to notice or test for due to their complexity. Given the rapid growth of computing power over the years, these inefficiencies were not considered significant.

However, computer efficiency is not scaling as fast anymore [12], and the demands for computing power have increased significantly, with more and larger data centers being built each year. These data centers demand a significant amount of energy [9], which means more energy costs and largely contributes to pollution and carbon footprint. Due to these factors, the energy inefficiencies introduced in different libraries and implementations that were not significant before have to be addressed.

This thesis provides a methodology that can be used as a first step to help developers and researchers find energy hotspots and inefficiencies in their software and dependencies. The methodology helps to effectively prune the libraries under study, and locate exactly which functions or sections of the dependencies have a greater effect on the energy hotspots. This way, the amount of code that needs to be investigated is narrowed down, making the task of answering inefficiencies easier.

We also proved the effectiveness of the technique by applying it to a case study with Redis and discovered a significant energy inefficiency in Alpine and *musl*, which causes a higher energy consumption for certain uses of the `memcpy` function, compared to the energy consumption in Ubuntu with *glibc*. This difference is reflected as a difference of up to 20.2% of power usage in Redis, and a 13% difference in a custom benchmark, without showing improvements in runtime.

Although the Linux and C communities are aware of performance differences between these two libraries, this is limited to the study of runtime performance. While working on this thesis, we reached out and presented our work to a group of experts in a KDE community meetup [1], who found our conclusions relevant and agreed that there is little awareness on energy performance for the *musl* library. While the difference in performance shown here might seem small, it becomes relevant when scaled not to one, but to thousands of Redis instances running in one or multiple data centers.

6.1. Future work

This thesis leaves open several avenues for future work. In our case study, we find that the *musl* implementation of `memcpy` displays some energy inefficiencies for certain usages of the function. However, we do not go into low-level code comparison between both libraries. To properly compare both implementations of the function, future work could perform more benchmarks and micro-benchmarks, and analyze the compiled assembly code. If the differences can be properly explained, it could make

[1] KDE meetup minutes `https://invent.kde.org/teams/eco/opt-green/-/blob/master/community-meetups/2024-06-12_community-meetup.txt` retrieved on June 17, 2024

it easier to locate similar problems in other functions of the C Standard library, and potentially make them easier to fix. Similarly, if certain patterns are found in both the assembly code from *glibc* and the compiled code, they could be use to locate energy inefficiencies in these libraries without having to perform benchmarks.

Another path for future work would be to apply our methodology to other workloads that show a difference in energy consumption. This would allow us to find more energy inefficiencies in other functions of *libc* or other shared libraries. Additionally, it would also serve to validate the usefulness of the methodology proposed and find and correct shortcomings that may appear when applied to other workloads.

References

[1] Suparna Bhattacharya et al. “Software bloat and wasted joules: Is modularity a hurdle to green software?” In: *Computer* 44.09 (2011), pp. 97–101.

[2] Randal E. Bryant and David R. O’Hallaron. *Computer Systems: A Programmer’s Perspective*. 3rd. Pearson, 2015. ISBN: 013409266X.

[3] Rajkumar Buyya, Shashikant Ilager, and Patricia Arroba. “Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads”. In: *Software: Practice and Experience* 54.1 (2024), pp. 24–38.

[4] “Climate belief and issue salience: Comparing two dimensions of public opinion on climate change in the eu”. In: *Social Indicators Research* 162.1 (2022), pp. 307–325. DOI: `10.1007/s11205-021-02842-0`.

[5] Luís Cruz. *Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments*. 2021. URL: `http://luiscruz.github.io/2021/10/10/scientific-guide.html` (visited on 05/27/2024).

[6] Luís Cruz. *Tools to Measure Software Energy Consumption from your Computer*. 2021. URL: `https://luiscruz.github.io/2021/07/20/measuring-energy.html` (visited on 05/27/2024).

[7] Luis Cruz and Rui Abreu. “Catalog of Energy Patterns for Mobile Applications”. In: *CoRR* abs/1901.03302 (2019). arXiv: `1901.03302`. URL: `http://arxiv.org/abs/1901.03302`.

[8] Luis Cruz and Rui Abreu. “Performance-Based Guidelines for Energy Efficient Mobile Applications”. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 46–57. DOI: `10.1109/MOBILESoft.2017.19`.

[9] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. “Data center energy consumption modeling: A survey”. In: *IEEE Communications surveys & tutorials* 18.1 (2015), pp. 732–794.

[10] Benedikt Dornauer and Michael Felderer. “Energy-saving strategies for mobile web apps and their measurement: Results from a decade of research”. In: *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2023, pp. 75–86.

[11] T. Durieux. “Empirical Study of the Docker Smells Impact on the Image Size”. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 2568–2579.

[12] Lieven Eeckhout. “Is moore’s law slowing down? what’s next?” In: *IEEE Micro* 37.04 (2017), pp. 4–5.

[13] Martin Heinz. *Why I Will Never Use Alpine Linux Ever Again*. 2023. URL: `https://martinheinz.dev/blog/92` (visited on 05/21/2024).

[14] Intel. *Reading and Writing Model Specific Registers (MSRs) in Linux*. 2024. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/reading-writing-msrs-in-linux.html` (visited on 02/23/2024).

[15] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. “Open tracing tools: Overview and critical comparison”. In: *Journal of Systems and Software* 204 (2023), p. 111793. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2023.111793`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121223001887`.

[16] Scott van Kalken. *Docker overview*. 2024. URL: `https://docs.docker.com/get-started/overview/` (visited on 06/05/2024).

[17] Scott van Kalken. *What Are Namespaces and cgroups, and How Do They Work?* 2021. URL: `https://blog.nginx.org/blog/what-are-namespaces-cgroups-how-do-they-work` (visited on 06/05/2024).

[18] Kashif Nizam Khan et al. “RAPL in Action: Experiences in Using RAPL for Power Measurements”. In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[19] Roberto Morabito. “Power Consumption of Virtualization Technologies: An Empirical Investigation”. In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 2015, pp. 522–527. DOI: 10.1109/UCC.2015.93.

[20] Roberto Morabito, Jimmy Kjällman, and Miika Komu. “Hypervisors vs. Lightweight Virtualization: A Performance Comparison”. In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[21] James Pallister, Simon J Hollis, and Jeremy Bennett. “Identifying compiler options to minimize energy consumption for embedded platforms”. In: *The Computer Journal* 58.1 (2015), pp. 95–109.

[22] Rui Pereira et al. “Ranking programming languages by energy efficiency”. In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[23] Giovanni Rosa, Simone Scalabrino, and Rocco Oliveto. “Assessing and Improving the Quality of Docker Artifacts”. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2022, pp. 592–596. DOI: 10.1109/ICSME55016.2022.00081.

[24] June Sallou, Luís Cruz, and Thomas Durieux. *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*. 2023. arXiv: 2312.13897 [cs.SE].

[25] Eddie Antonio Santos et al. “How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers”. In: *Journal of Systems and Software* 146 (2018), pp. 14–25.

[26] Mathijs Jeroen Scheepers. “Virtualization and Containerization of Application Infrastructure : A Comparison”. In: 2014. URL: https://api.semanticscholar.org/CorpusID:18129086.

[27] Robert Schone et al. “Energy Efficiency Aspects of the AMD Zen 2 Architecture”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2021. DOI: 10.1109/cluster48925.2021.00087. URL: http://dx.doi.org/10.1109/Cluster48925.2021.00087.

[28] Prateek Sharma et al. “Containers and Virtual Machines at Scale: A Comparative Study”. In: *Proceedings of the 17th International Middleware Conference*. Middleware ’16. Trento, Italy: Association for Computing Machinery, 2016. ISBN: 9781450343008. DOI: 10.1145/2988336.2988337. URL: https://doi.org/10.1145/2988336.2988337.

[29] William Stallings. *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011, pp. 1–788. ISBN: 978-0-273-75150-2.

[30] Senay Semu Tadesse, Carla Fabiana Chiasserini, and Francesco Malandrino. “Characterizing the power cost of virtualization environments”. In: *Transactions on Emerging Telecommunications Technologies* 29.8 (2018), e3462.

[31] Jiaqi Tan et al. “SALSA: Analyzing Logs as StAte Machines.” In: *WASL* 8 (2008), pp. 6–6.

[32] Bailey Tjiong. *The impact of base image selection on the energy efficiency of containerized applications in Docker*. Master’s thesis. Available at http://resolver.tudelft.nl/uuid:1166da2a-a62d-4b53-baa3-08e6e107053b. Delft, NL, Dec. 2023.

[33] I. Turner-Trauring. *Using Alpine can make Python Docker builds 50× slower*. 2023. URL: https://pythonspeed.com/articles/alpine-docker-python (visited on 05/21/2024).

[34] Perf Wiki. *perf: Linux profiling with performance counters*. 2021. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 05/27/2024).

[35] Guoqing Xu et al. “Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER ’10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 421–426. ISBN: 9781450304276. DOI: 10.1145/1882362.1882448. URL: https://doi.org/10.1145/1882362.1882448.

[36] Joseph Zambreno, Mahmut Taylan Kandemir, and Alok Choudhary. “Enhancing compiler techniques for memory energy optimizations”. In: *Embedded Software: Second International Conference, EMSOFT 2002 Grenoble, France, October 7–9, 2002 Proceedings 2*. Springer. 2002, pp. 364–381.

[37] Yang Zhang et al. “An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency”. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. 2018, pp. 138–143. DOI: `10.1109/COMPSAC.2018.00026`.

[38] Yinyuan Zhang et al. “Recommending base image for docker containers based on deep configuration comprehension”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2022, pp. 449–453.

A

Tracing PostgreSQL execution

As another experiment to test the tracing capabilities of our methodology, we studied the energy consumption and tracing of PostgreSQL. This workload is tested with the *pgbench*[1] utility, and previous work also showed a significant difference in energy consumption between Alpine and Ubuntu. PostgreSQL is also written in C, but it has some extra dependencies compared to Redis, like OpenSSL for encrypted communication [2]. Our experimental method and tracing utility should still work.

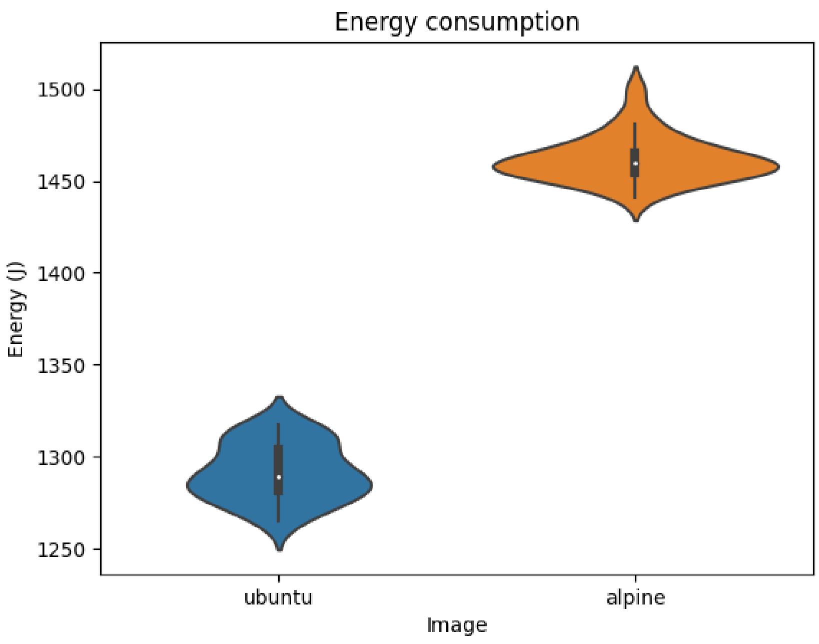


Figure A.1: Energy consumption for PostgreSQL with different base images

Figure A.1 shows the total energy consumption for the PostgreSQL benchmark in Alpine and Ubuntu, using the same PostgreSQL version. Figure A.2 shows the power usage of the benchmark across time.

From these, we can see that the pattern of this benchmark is different from Redis. The benchmark and operations realized seem to be much more uniform than in the case of Redis. From consulting the documentation we can verify that this is the case. With the default configuration, the benchmark performs the given number of transactions, each involving a combination of five *INSERT*, *SELECT*, and *UPDATE*.

[1] `https://www.postgresql.org/docs/current/pgbench.html`

[2] `https://www.postgresql.org/docs/current/install-requirements.html`

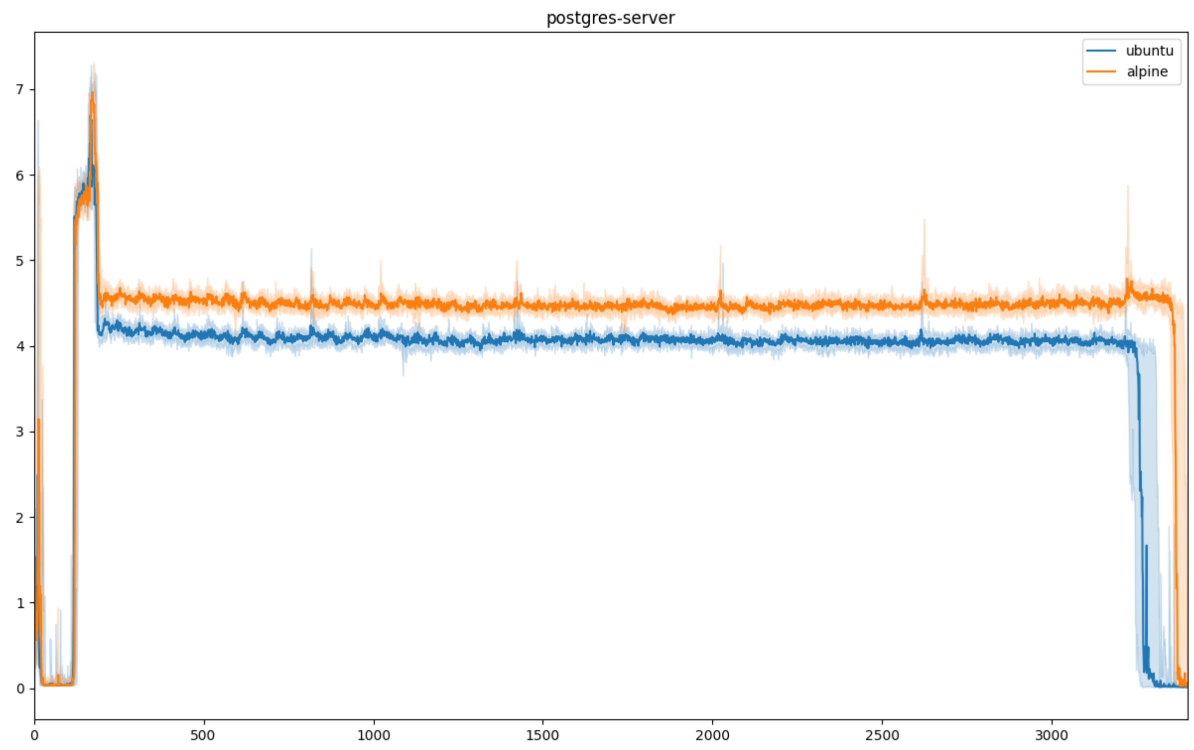


Figure A.2: Power usage of PostgreSQL in Alpine and Ubuntu

From the power usage, we can also see that Alpine uniformly uses slightly more power than Ubuntu, and it also takes longer to complete, which means that this higher usage does not translate to faster operation, which could end up saving energy.

Next, we perform a traced run with *uftrace* as we did with Redis. Table A.1 shows the summary of calls for PostgreSQL. From here, we can see that the first function with most runtime is `epoll_wait`, followed by `send` and `recv` which, according to POSIX documentation[3], are aliases to `write` and `read` for TCP sockets. Once again, we can see that TCP communication is a significant part of the workload.

The next step would be to perform log analysis to plot a histogram of function usage normalized by time. However, the logs for this tool do not have a specially good format, and they do not indicate the start and/or stop of functions properly, so applying the checkpoint method directly is not possible.

From benchmark code and documentation, we can see that the benchmark is divided into two parts,

[3]`https://pubs.opengroup.org/onlinepubs/007904975/functions/recv.html`

Function	Runtime (ns)	Calls	Runtime %	Call %
`epoll_wait`	11668930119	357797	27.965881	0.404581
`send`	6766784365	350356	16.217347	0.396167
`recv`	4675643546	1061822	11.205697	1.200661
`memset`	3655976982	39949235	8.761953	45.172796
`pwrite`	2019083997	180199	4.838958	0.203761
`sem_wait`	1860548575	56483	4.459010	0.063882
`lseek`	1804199688	493557	4.323964	0.558092
`pread`	1102366718	128930	2.641944	0.145788
`strlen`	1036313226	10088285	2.483639	11.407378
`fdatasync`	915658688	27699	2.194477	0.031321

Table A.1: Summary of *libc* function calls and runtimes for PostgreSQL

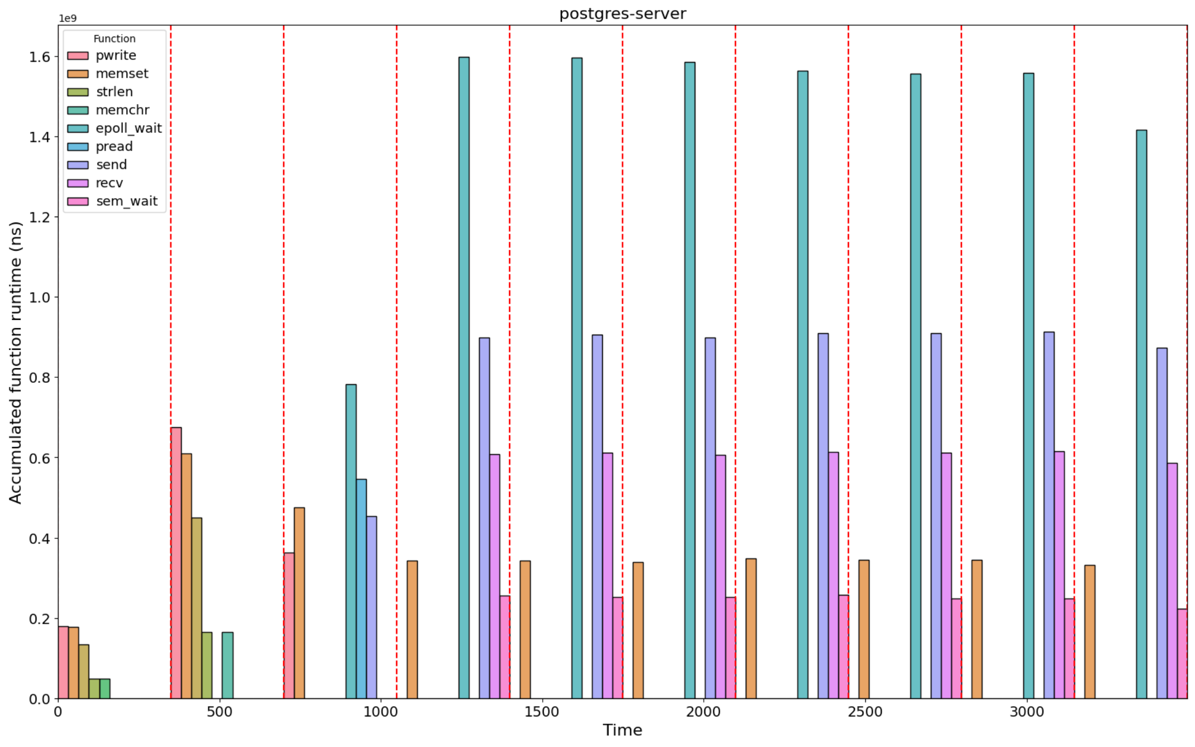


Figure A.3: Histogram showing accumulated function runtime in each region between checkpoints for Postgres

each of them uniform: database and tables initialization and the actual benchmark. We also see how the power usage difference is uniform except for the initialization at the beginning. In this case, a simple uniform division for the histogram would be enough to properly analyze the logs.