Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

GreenErl

Green computing for Erlang

Supervisors:
Dr. Melinda Tóth, Dr. István Bozó
Associate Professor, Assistant Professor

Author:
Youssef Gharbi
Computer Science MSc
2. year

Budapest, 2023

Abstract

Energy efficiency is the production of the same result with less energy consumption. Having the same result from software with less energy consumption not only helps the companies reduce their bills but might lead to faster software outcomes. Thus, the aim of this thesis is to try to measure energy consumption and find some patterns for Erlang and try to improve new possible findings on the previous results. Previous work focused on measuring Erlang under the Unix-like OS. Thus the aim of this thesis is to focus on the Erlang measurements for Windows OS. Finding a tool for Windows OS that fits our requirements is one of the main challenges since there exist many tools to measure CPU and Memory energy consumption for Windows but most of them are not able to measure the Erlang functions specifically. Thus, I tried many available tools such as Intel Power Gadget, Windows Energy Estimation Engine (E3), PowerCfg, CPU-Z, and Scaphandre. I choose Scaphandre since it was the most convenient tool and integrated it with the previous GreenErl framework. I first analyzed the results of running Erlang on Windows. After, I reproduced the previous measurements under the Windows environment and tried to spot some differences between running Erlang code under Windows OS and Unix-like environments which were behaving more similarly than differently except in some cases like dictionaries.

Contents

1 Introduction **4**

1.1 Contribution 5

1.2 Structure of the paper 6

2 Background **8**

2.1 Green computing 8

2.2 Erlang 10

2.3 GreenErl framework for Linux 11

2.4 RefactorErl 13

3 Tools for Windows **14**

3.1 Trying available tools 15

4 Scaphandre based tool-chain **20**

4.1 A deep dive into Scaphandre 20

4.2 Why Scaphandre 28

4.3 GreenErl for Windows 30

5 Measuring energy consumption **40**

5.1 Methodology 40

5.2 Previous measurements 41

5.3 Data structures 42

5.4 Higher-order functions 56

5.5 Parallel language constructs 62

5.6 Algorithmic skeletons 64

6 Comparing energy consumption **75**

6.1 Data structures 76

6.2 Higher-order functions 77

6.3 Parallel language constructs 78

6.4 Algorithmic skeletons 79

7 Related work **81**

7.1 Windows versus Linux operating systems 81

7.2 Programming languages efficiency 83

8 Conclusion **85**

Bibliography **86**

Appendix A Comparison **93**

A.1 Data structures 93

A.2 Higher-order functions 113

A.3 Parallel language constructs 121

A.4 Algorithmic skeletons 128

Chapter 1

Introduction

Nowadays, environmental consciousness is growing in popularity [1], today it is important to create software that is energy efficient. Cisco says that 90% of internet traffic goes through Erlang-controlled nodes[2]. With 5.07 billion internet users in the world growing by half a million daily[3], today comes the need to have efficient Erlang code patterns more than ever.

Energy use is a major concern in every aspect of our life. Being energy efficient is a crucial component of the modern manufacturing process for the majority of items[4]. This also covers the production of computer components. In addition to producing hardware components effectively, we also require efficient computers.

Computer performance is mostly determined by two factors. The hardware we employ and the applications we use on our PCs. For decades, people have been designing and producing ever-more energy-conscious hardware, but it has only recently become common practice to write software that is energy-efficient. Computers are being employed in more energy-critical systems as a result of their ubiquitous use and the Internet of Things (IoT).

Indeed building a sophisticated deep learning model can cost billions of dollars

and produce Co2 emissions equal to a month in a big city such as New York[5]. Mobile phones are an excellent illustration of it since we want to extend the battery's life as much as possible.

The numerous factors already mentioned make energy efficiency in computers necessary. Although efficiency has always been a key component of software technology. We first need to look into and identify language constructions and language components that make programs spend less energy so that programmers can design more energy-efficient code. Refactoring existing code bases to utilize more energy-conscious patterns is another technique to create software that is more energy-efficient. A previous tool was created in the last years called GreenErl. The tool was mainly developed for Unix-like Operating Systems. Thus, the aim of this work is to create, edit and develop a similar tool for Windows. We reproduced the same measurements from previous work and compared the results by trying to spot several valuable trends between the two operating systems and if there exists a more energy efficient one.

1.1 Contribution

During this section, I will try to explain the main steps taken to complete this thesis.

The first crucial step was an extensive search for a tool that is able to produce the energy consumption measurements and gather the results in files in order to generate the graphs. During the laboratory course, I was able to find, install and test many of those tools and decided on the final one to be used. The decision was not random since we had several criteria that needed to be fulfilled.

The second step after finding the correct tool was experimenting and editing the previous GreenErl framework [6] in order to make that tool work coherently with the new version under Windows.

After having an MVP of GreenErl on Windows, I started measuring the previous functions and trying to improve the results. Once decided on the final version of the new GreenErl for Windows I started measuring all of the functions.

The last step was to analyze the findings. The analysis was in two parts. The first part was purely analyzing the behavior of Erlang on Windows. The second part was about comparing the trends and the manners that differ between these two systems.

1.2 Structure of the paper

In this section, I will talk briefly about each chapter to give an understandable overview of the thesis.

As all thesis works should start, I started my thesis with an introduction which this section is part of. During this Chapter, I am giving an overview of the topic of the research and the main steps followed to achieve the results.

The next Chapter serves as an investigation of the background and the main motivation behind this work. I also dived deeper into the understanding of the different components that the previous framework was built from.

The third Chapter is a detailed experiment on all the tools I tried in order to decide which one I should move on with. I explained the constraints needed for that tool so it can be integrated with the Windows GreenErl framework.

In the fourth Chapter, I started with an inspection of the chosen tool Scaphandre [7, 8], I explained the reason why I choose it over the other tools. Followed by building and editing the new GreenErl framework for Windows, how I integrated it and the way the framework is working.

The fifth Chapter is where the analyses started. Here I inspected all of the

previously measured functions. But before that, I gave a detailed explanation of the methodology and the environment I used for the measurement which consisted of four main areas. The first is data structures (map, list, dictionary) where I compared some basic operations like deleting or updating an element. Next were the higher-order functions followed by parallel language constructs and algorithmic skeletons.

The sixth Chapter is a straightforward comparison of the energy consumption and runtime between the previous results [6] and the new results. In order for the measurements to be fair, I repeated the exact same measurements and the same inputs from the previous results.

The seventh Chapter is where I looked into some of the published research related to my work. It is composed of two parts. First I tried to understand the key differences between Linux and Windows. The second section was about understanding the differences between some of the programming languages.

The last chapter is a conclusion about this work.

Chapter 2

Background

The previously mentioned attributes were the initiating building blocks for this work. Throughout this Chapter, I will talk about green computing as long as the main component of the previous GreenErl framework.

2.1 Green computing

Green computing is a study and practice of efficient and eco-friendly computing resources [9]. It involves developing, designing, engineering, producing, using, and disposing of computing modules and devices to reduce environmental hazards and pollution [10].

Green computing has become increasingly important in recent years as the world becomes more aware of the impact of technology on the environment [4]. The use of technology has led to an increase in energy consumption and carbon emissions [4]. Green computing aims to reduce these negative impacts by developing more energy-efficient computing systems and reducing waste.

One way green computing can be achieved is through energy-efficient comput-

ing "energy efficient computing cites". This involves designing computer systems that use less energy while still providing the same level of performance. Energy-efficient computing can be achieved through various methods such as using low-power processors, optimizing software code, and using virtualization technology.

Another way green computing can be achieved is through sustainable computing practices [4]. This involves using environmentally friendly materials in the production of computer systems and reducing waste by recycling old computer systems [4]. Sustainable computing practices can also involve reducing paper usage by using digital documents instead of printed ones.

Green data centers are another aspect of green computing. Data centers consume a large amount of energy due to their high-performance computing requirements. Green data centers aim to reduce this energy consumption by using renewable energy sources such as solar or wind power. They also use energy-efficient cooling systems that reduce the amount of energy required to cool the data center. They also implement different techniques such as reducing wasted resources by tailoring the resources [11].

Environmental impact assessments are also an important aspect of green computing. These assessments evaluate the environmental impact of computer systems throughout their life cycle. They take into account factors such as energy consumption, carbon emissions, and waste production.

Green computing software in another way refers to software that is designed to reduce energy consumption and carbon emissions [12]. It involves developing software that uses less energy while still providing the same level of performance. Green computing software can be achieved through various methods such as optimizing software code, using virtualization technology, and using low-power processors.

Green computing software has become increasingly important in recent years as the world becomes more aware of the impact of technology on the environment.

The use of technology has led to an increase in energy consumption and carbon emissions. Green computing software aims to reduce these negative impacts by developing more energy-efficient software systems.

One way green computing software can be achieved is through optimizing software code. This involves designing software that uses less energy while still providing the same level of performance. Optimizing software code can be achieved through various methods such as reducing redundant code, using efficient algorithms, and reducing memory usage.

Another way green computing software can be achieved is through virtualization technology. This involves running multiple virtual machines on a single physical machine. Virtualization technology can reduce energy consumption by consolidating workloads onto fewer physical machines.

To conclude, green computing is the study and practice of efficient and eco-friendly computing resources that aim to reduce environmental hazards and pollution. It involves developing more energy-efficient computing systems and software, using environmentally friendly materials in production, reducing waste through recycling old computer systems, reducing paper usage by using digital documents instead of printed ones, using renewable energy sources such as solar or wind power in data centers, evaluating the environmental impact of computer systems throughout their life cycle, optimizing software code, and using virtualization technology.

2.2 Erlang

Erlang is a general-purpose, concurrent, functional high-level programming language that was developed by Ericsson in the late 1980s [13]. It is used to build massively scalable soft real-time systems with requirements of high availability [14]. It can be used for a wide range of applications, some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Er-

lang's runtime system has built-in support for concurrency, distribution and fault tolerance.

Erlang has several features that make it unique among programming languages and it also can serve as a runtime system. One of these features as mentioned earlier is its native support for concurrency and distribution. Erlang's concurrency model is based on lightweight processes that are isolated from each other and communicate through message passing. This makes it easy to write concurrent programs that can run on multiple processors or even multiple machines.

Another feature of Erlang is its support for fault tolerance. Erlang programs are designed to be fault-tolerant, which means that they can continue to operate even if some parts of the system fail. This makes Erlang an ideal choice for building systems that require high availability and reliability.

Erlang is a dynamically typed language, therefore errors are only raised at runtime once types of functions and variables are checked at runtime. Additionally, Erlang is strongly typed, which means there is no implicit type conversion [15].

Erlang/OTP stands for Open Telecom Platform [13, 14]. It is a set of Erlang libraries and design principles providing middle-ware to develop massively scalable soft real-time systems with requirements on high availability. It includes its own distributed database, applications to interact with other languages, debugging and release handling tools.

2.3 GreenErl framework for Linux

The previous thesis [6] work made several important contributions, including the development of GreenErl, a tool designed to measure the energy consumption of Erlang programs. GreenErl comprised an Erlang module, a Python-based graphical user interface, and the `rapl-read.c` program, making it user-friendly and effective in analyzing the energy usage of different language constructs and

elements.

The study used GreenErl to examine the energy consumption of various data structures, such as proplists, maps, and dictionaries and found significant differences in the energy consumption of various operations. The study also explored the optimal scenarios for transforming lists into maps and identified the limits and kinds of operations that justify such a transformation.

Moreover, the study investigated the effect of higher-order functions on energy consumption and suggested replacing higher-order function calls with either a list comprehension or a specialized recursive function to decrease energy consumption.

In addition, the study examined the energy cost of different parallel language constructs, particularly the energy consumption of sending different data structures between processes. The study found that sending maps instead of lists can decrease energy consumption and suggested that transforming a list into a map can be a viable option for minimizing energy consumption in parallel language constructs.

Finally, the study presented proposed refactorings based on the findings, such as replacing calls to **proplists:get_value/2** with the more efficient **lists:keyfind/3**, transforming a recursive function definition that uses a proplist as its parameter to use a map instead, and eliminating higher-order function calls by replacing calls to the **lists:map/2** and **lists:filter/2** functions with either a new recursive function or a list comprehension, depending on what the user desires.

As we said, the GreenErl framework was developed using RAPL [16] to read the energy consumption from the designated registers and since Windows does not have APIs that allow access to the needed model-specific registers [17], hence we need to find a similar tool that allows us to reproduce the same measurements and findings on Windows Operating Systems with keeping in mind different constraints. The most important constraint was that we would like to measure the same Erlang modules meaning that we don't want to redefine new function descriptions to have

the highest similar environmental comparison. We also need to consider that some functions are so fast and have low energy consumption usage so the tool we should choose can handle values in nano-seconds and micro-watts. The last constraint is that we need to have the values in a file so that we can plot it in a graph and can make the comparison.

2.4 RefactorErl

RefactorErl [18, 19] is a tool for Erlang developers that was developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. It is an open-source static source code analyzer and transformer tool that focuses on supporting the daily code comprehension tasks of Erlang developers. This tool was used to perform the previous findings and implement the refactoring [6].

Chapter 3

Tools for Windows

In this Chapter, I would like to go through all the tools that I have been testing during the research phase. Since there exists a different number of tools we need to test each one of them in detail, to see if it meets our predefined constraints.

The most important constraints to keep in mind:

- Ability to gather data for Erlang operating system level process
- Fast sampling timestamps (Nanoseconds)
- Ability to read low energy consumption values (Micro-Watts)
- Ability to store the gathered values in a file (JSON, CSV,...)
- Possibility to integrate it with GreenErl

I placed all of the screenshots and output files csv, png and other logfiles for testing the tools under the directory `~\green_erlang\tools_windows\{tool_name}` where {*tool_name*} refers to the tool currently being tested.

3.1 Trying available tools

Intel-Power Gadget

Intel Power Gadget [20] is a software-based power usage monitoring tool developed by Intel Inc. It includes an application, driver, and libraries to monitor and estimate the real-time processor package power information in watts using the energy counters in the processor. It is a software-based power estimation tool enabled for second-generation Intel Core processors or newer. It provides real-time processor package power information in watts using energy counters. Intel Power Gadget is supported on both Windows and Mac OS X. Additionally to the graphical user interface, Intel-Power Gadget has a command line tool.

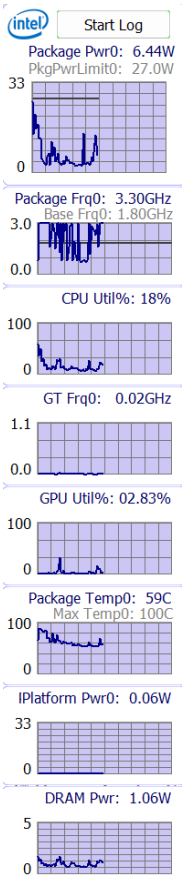


Figure 3.1: Intel-Power Gadget

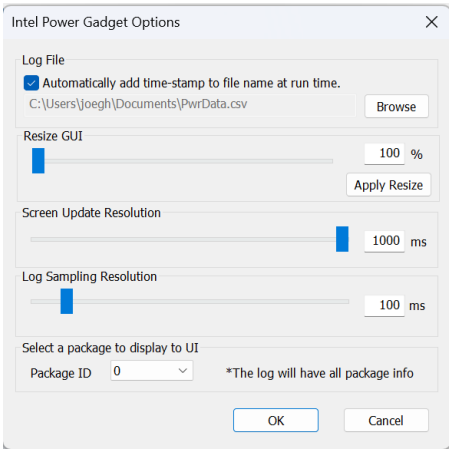


Figure 3.2: Intel-Power Gadget Settings

To be sure whether the tool fulfills those constraints, we need to test both the graphical interface (Figure 3.1) as long as the command line interface.

Starting with the graphical interface, as shown in Figure 3.2 it does not offer much flexibility. We can just set the path to the result data and the energy counter-sampling resolution that can reach a maximum of 1000 ms.

```
PowerLog3.0.exe [-resolution <msec>] –duration <sec> [-verbose] [-file
<logfile>]
```

Figure 3.3: Log power data to logfile for a period of time

```
PowerLog3.0.exe [-resolution <msec>] [-file <logfile>] [-verbose] –cmd
<command>
```

Figure 3.4: Start a command a log power data to logfile until the command finish

As figure 3.3 and figure 3.4 are showing we can get a CSV file either until a command finish executing or we can specify a duration for logging. Logfile will include the elapsed time, package power limit, processor frequency, GT frequency, processor temperature, average and cumulative power of the processor where Processor Energy is the total energy of the processor. With this result we cannot consider using Intel-Power Gadget since it doesn't provide the energy of the Erlang process but the whole process consumption.

Windows Energy Estimation Engine (E3) PowerCfg

The Energy Estimation Engine (E3) [21] is a feature in Windows that provides detailed information about battery usage. It can be used to measure our application power and carbon impact. We can use it to get detailed information about battery usage by running the **`powercfg/srumutil`** command which enumerates the entire Energy Estimation data from the System Resource Usage Monitor (SRUM) in an XML or CSV file. The E3 feature is useful for measuring

power consumption and the carbon impact of applications. The E3 CSV result as you can see in the logging folder as said in this part 3 is relatively huge sized around 10MB which includes all the applications I was running during the last 7 days by default. We can reset the local database and run the application we want to measure but the CSV file doesn't fulfill the requirement of logging a process level energy consumption. Also, it is not possible to easily integrate it with GreenErl.

CPU-Z

CPU-Z [22] is a free utility that provides detailed information about the computer's hardware. It can provide information about the CPU, motherboard, memory, and other hardware components. It can also provide real-time monitoring of our CPU and memory usage. It can be downloaded from the official website easily. We can log the results in a TXT or HTML file. However, we can't get the system-level process's energy consumption which means we cannot use it in our GreenErl framework. I won't be able to share the report since it includes detailed information about my system hardware but you can check this official CPU-Z website for more details or this TXT example [23].

Scaphandre

Scaphandre is a French word that means "diving suit" in English. It is a type of atmospheric diving suit (ADS) that allows divers to work at great depths for extended periods of time without decompression stops. Scaphandre the software [7, 24] is a tool that makes it possible to see the power being used by a single process on a computer. It is a metrology agent dedicated to electrical power consumption metrics. Scaphandre makes it easier and cheaper for tech providers and tech users to measure the power consumption of their tech services and get this data in a convenient form. It is still in the first phases of development. The latter was not easy to install at first because it consists of two modules in order to be able to use it on Windows. The first to install is Windows-RAPL-Driver [25] then we can

install it from GitHub and compile it as a rust application. We will go into more detail about Scaphandre installation and its components in the next Chapter 4.

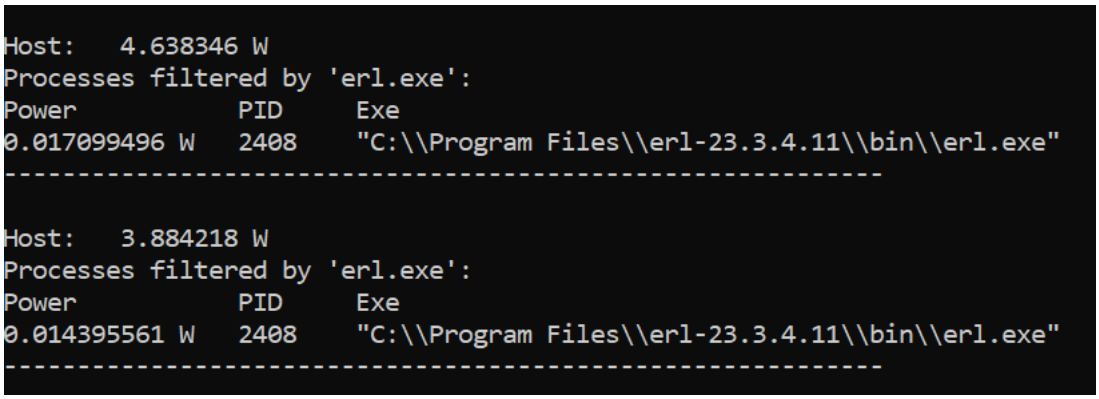


Figure 3.5: Scaphandre STD-OUT example

As Figure 3.5 shows, trying the STD-OUT feature of Scaphandre reveals the ability to get Pid level power consumption. With the JSON feature shown in Figure 3.6, we can get timestamps also. The ease of use provided by JSON structure and the level of details provided by Scaphandre, it was the right tool to use.

```
[
    {
        "host": {
            "consumption": 4260821.0,
            "timestamp": 1672858309.2251165
        },
        "consumers": [
        {
            "exe": "C:\\Program Files\\Google\\Chrome\\
    Application\\chrome.exe",
            "pid": 12836,
            "consumption": 0.0,
            "timestamp": 1672858309.2453358,
            "container": null
        },
        {
            "exe": "C:\\Program Files\\erl-23.3.4.11\\
    bin\\erl.exe",
            "pid": 6648,
            "consumption": 249948.05,
            "timestamp": 1672858311.1400018,
            "container": null
        }
    }
]

```

Figure 3.6: Scaphandre JSON example

Chapter 4

Scaphandre based tool-chain

This chapter provides an overview of the Scaphandre tool, a key component of the Windows GreenErl framework. We will first describe the architecture and design principles of Scaphandre. Then the tool customization to fit the new GreenErl framework for Windows.

4.1 A deep dive into Scaphandre

General description of Scaphandre

One of the main challenges in reducing the energy consumption of tech services is to measure it accurately and transparently. However, most of the existing tools and methods are either too costly, too complex, or too opaque to be widely adopted by providers and users. For instance, some tools require dedicated hardware or software agents that are not compatible with all platforms or architectures. Others rely on indirect estimations or averages that do not reflect the actual power consumption of a specific service or device. Furthermore, some tools do not expose the power consumption metrics to the users or clients, making it difficult to

compare or optimize them.

Scaphandre [24, 7, 8] is a new open-source tool that aims to address these challenges by providing a simple, reliable and transparent way to measure and reduce the energy consumption of tech services. Scaphandre is a metrology agent that collects power consumption metrics from various sources, such as bare metal hosts, virtual machines, containers or cloud platforms. It then exposes these metrics in a convenient form, allowing them to be sent to any monitoring or data analysis toolchain. Scaphandre also enables users and clients to access their own power consumption metrics, breaking the opacity of being on someone else's computer.

The main features of Scaphandre are:

- Measuring power consumption on bare metal hosts.
- Measuring power consumption of qemu/kvm virtual machines from the host
- Exposing power consumption metrics of a virtual machine, to allow manipulating those metrics in the VM as if it was a bare metal machine (relies on hypervisor features).
- Exposing power consumption metrics as a prometheus (HTTP) exporter.
- Sending power consumption metrics to riemann.
- Sending power consumption metrics to Warp10.
- Working on containers (Docker, Kubernetes).
- Storing power consumption metrics in a JSON file.
- Showing basic power consumption metrics in the terminal.

Architecture and Design Principles

Scaphandre is composed of three main components: sensors, exporters and consumers. Figure 4.1 shows an overview of the architecture of Scaphandre's three

main components using Prometheus as an exporter and Grafana as a consumer.

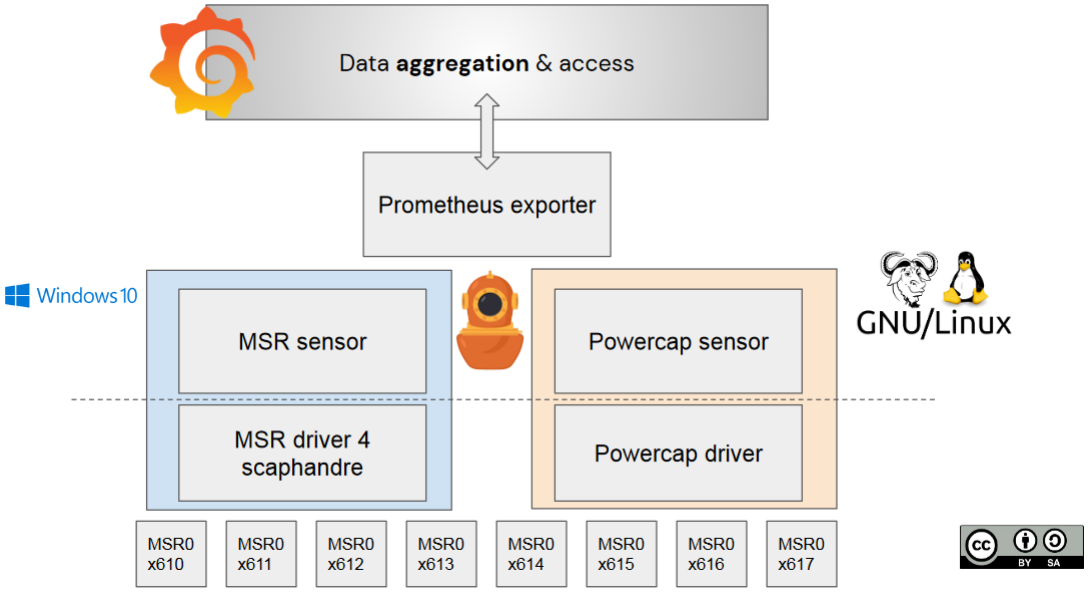


Figure 4.1: Scaphandre Prometheus exporter example [44]

Sensors

Sensors are responsible for collecting power consumption metrics from various sources. They can be either hardware-based or software-based sensors. Hardware-based sensors rely on physical devices that measure the electrical current or voltage of a host or a device. Software-based sensors use software interfaces or libraries that provide access to power consumption information from a host or a virtual machine.

Exporters

Exporters are responsible for exposing or sending power consumption metrics to different destinations. They can be either push-based or pull-based exporters. Push-based exporters send metrics periodically or on demand to a remote server or

service, such as Riemann or Warp10. Pull-based exporters expose metrics through an HTTP endpoint that can be queried by a client or service, such as Prometheus.

Consumers

Consumers are responsible for receiving or requesting power consumption metrics from exporters. They can be either human-based or machine-based consumers. Human-based consumers use graphical or textual interfaces to visualize or analyze power consumption metrics, such as dashboards or terminals. Machine-based consumers use programmatic interfaces to process or optimize power consumption metrics, such as scripts or algorithms.

windows-rapl-driver

RAPL (Running Average Power Limit) [26] is a feature of modern Intel and AMD processors that allows measuring and controlling the power consumption of various components, such as the CPU cores, the memory, and the package. RAPL provides access to power and energy counters through model-specific registers (MSRs) that can be read by software. RAPL is widely used in Linux systems for energy-aware applications, such as monitoring, profiling, and optimizing the energy efficiency of software and hardware. However, RAPL is not natively supported in Windows systems [17], which limits the applicability of RAPL-based solutions for Windows users and developers.

Windows-RAPL-Driver [25] is a project that aims to fill this gap by providing a Windows driver that enables RAPL metrics gathering from a physical computer. The driver exposes RAPL data through a device file that can be accessed by user-space applications. The driver also supports test signing and installation as a self-signed driver for testing and troubleshooting purposes.

The latter consists of two main components: a kernel-mode driver (Scaphan-

dreDrv) and a user-mode loader (DriverLoader). The driver is responsible for reading RAPL MSRs and providing RAPL data to user-space applications. The loader is responsible for installing, loading, and unloading the driver.

How Scaphandre is measuring the energy consumption

Measuring the power consumption of a single process is not a trivial task, as it involves several challenges and trade-offs. In this section, I will explain how Scaphandre computes per-process power consumption and what are the main assumptions and limitations behind its approach [27].

The basic idea behind Scaphandre's computation is to combine two sources of information: the share of resources used by a process and the total power used by the system (Figure 4.3). The share of resources used by a process is obtained by counting the number of jiffies (small intervals of time) that the process runs on the CPU (Figure 4.2). The total power used by the system is obtained by reading the power sensors available on the hardware, such as Intel's RAPL (Running Average Power Limit) interface. By multiplying the share of resources used by a process with the total power used by the system, Scaphandre can estimate the power consumption of that process.

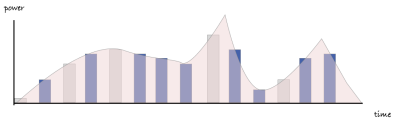


Figure 4.2: Power and share of usage [27]

Figure 4.3: Process time-sharing [27]

However, this computation is not exact, as it involves some simplifications and approximations. For example, Scaphandre assumes that the power consumption of a process is proportional to its CPU usage, which may not be true for processes that use other resources, such as memory, disk, or network. Scaphandre also assumes that the power consumption of the system is constant during each jiffy, which may not be true for systems that have dynamic power management features,

such as frequency scaling or sleep states. Scaphandre also relies on the availability and accuracy of the power sensors on the hardware, which may vary depending on the vendor and model.

Therefore, Scaphandre's computation of per-process power consumption should be considered as an estimation rather than a measurement. It can provide useful insight into the relative energy impact of different processes or applications, but it may not reflect the absolute energy consumption in terms of watt-hours or joules. Scaphandre's computation also depends on several parameters and options that can be adjusted by us, such as the sampling interval, the sensor type, or the domain to monitor. These parameters and options can affect the precision and accuracy of Scaphandre's computation and should be chosen carefully depending on the use case and context.

Installation

To be able to run Scaphandre on Windows I needed to first install Windows-RAPL-Driver [25] to allow access to metrics and then Scaphandre [8].

Throughout the installation process, I was facing many challenges since the framework was in an early phase of development. The lack of documentation for Windows users was noticeable. However, the team was helpful and easily reachable through Gitter [28]. I even opened a GitHub issue [29] and I was able to install both the driver and Scaphandre. The GitHub issue I opened is well-detailed and was referenced in the new release [30] of Scaphandre. Luckily, for those who will try the framework in the future, there is a new release V0.5.0 of Scaphandre [30] and a new documentation and explanation of the Windows-RAPL-Driver installation on the official website [31] supporting Windows.

Windows-RAPL-Driver Installation

As it was mentioned earlier, a detailed description of the steps and troubleshooting I did can be easily accessed in this GitHub issue number 247 (`https://github.com/hubblo-org/scaphandre/issues/247`).

To summarise the steps and findings needed to install Windows-RAPL-Driver [31] up to the end of December 2022:

1. Enable test mode on Windows. This is necessary because the driver is unsigned and requires custom drivers to run. To do this, I opened a command prompt as an administrator and run: **`bcdedit.exe -set TESTSIGNING ON`**. Then restarted the computer. Once done with the measurements I disabled test mode by running this command again as an administrator: **`bcdedit.exe -set TESTSIGNING OFF`**.

2. From the GitHub repository [25] I needed to compile the following files: **`DriverLoader.exe`** and **`ScaphandreDrv.sys`**. They need to be in the same folder.

3. Running **`DriverLoader.exe install`** in an administrator command prompt. This installed the driver as a service [32, 33] on my laptop.

4. Running **`DriverLoader.exe start`** to start the service. The service can be checked with the command: **`driverquery /v | findstr -i scaph`**. If running properly, it should show a line like the Figure 4.4.

```
ScaphandreDr Scaphandre Driver Serv Scaphandre Driver Serv Kernel
        System     Stopped    OK          FALSE       FALSE
 0                 4,096       0          01/18/2022 09:48:28
\??\C:\windows-rapl-driver\kernelland\Scaphandre 4,096
```

Figure 4.4: Scaphandre Driver

5. The driver is now ready to be used by Scaphandre.

Scaphandre Installation

Once the driver is ready, the very next step is to install Scaphandre itself. At this step, I spent a remarkable amount of time trying to solve the different errors that occurred during compilation. Since Scaphandre is written using Rust [34] and its package manager Cargo [35]. To summarise the important steps I followed:

1. Install Rust and Cargo using rustup [36]
2. Install Visual Studio 2019 with C++ build tools [37]
3. Install MSYS2 and add it to the environment variables PATH
4. Install pkg-config and add it to the environment variables PATH
5. Install all needed packages using MSYS2 and pkg-config
6. Clone the Scaphandre repository [8] and check out the **dev** branch
7. Compiled Scaphandre using the Rust for Windows usual toolkit cargo. Since until now, not all Scaphandre features are supported on Windows. Using the following command line to build the binary: **`cargo build --no-default-features --features`** `"prometheus json riemann"`.
8. Add Scaphandre to the environment variables PATH. Scaphandre can be found in **`Scaphandre/target/debug`** folder.
9. Test Scaphandre from a command prompt.

Once Scaphandre was installed, I was able to start experimenting with its features. Since not all features are available for Windows, only JSON, Prometheus, Riemann and STDOUT. However, despite Prometheus and Riemann being available in Scaphandre for Windows it was not possible to read the metrics since those features need a PowercapRAPL sensor which is not available on Windows (MsrRAPL only for now) [38]. In the near future, Hubblo [39] is working on different improvements and maybe all features might be available on Windows as well.

4.2 Why Scaphandre

Introduction

As said earlier, Scaphandre is a software-based power usage monitoring tool that can measure the power consumption of tech services and applications on various platforms, such as bare metal hosts, virtual machines, and Kubernetes clusters. It can also expose or send power consumption metrics to different data analysis or monitoring tools, such as Prometheus, Riemann, or Warp10. In this section, I will compare Scaphandre with other already tested watts meters, such as powercfg, CPU-Z, and Intel Power Gadget, and show why Scaphandre is a better choice for measuring power consumption in different scenarios and why I ended up choosing it for the GreenErl framework for Windows.

Powercfg VS Scaphandre

Powercfg [21] is a command-line tool that is built-in on Windows systems and can control power plans, sleep states, device power states, and analyze energy efficiency and battery life problems. However, powercfg has some limitations compared to Scaphandre. First, powercfg only works on Windows systems, while Scaphandre can work on both Windows and Linux systems. Second, powercfg can only measure the power consumption of the whole system or individual devices, while Scaphandre can measure the power consumption of specific processes or applications. Third, powercfg can only export power consumption data to XML files or display them on the terminal, while Scaphandre can send or expose power consumption data to various data analysis or monitoring tools.

CPU-Z VS Scaphandre

CPU-Z [22] is a freeware tool that can display detailed information about the CPU, memory, motherboard, and graphics card on Windows systems. It can also measure the voltage, temperature, and power consumption of the CPU. However, CPU-Z also has some drawbacks compared to Scaphandre. First, CPU-Z only works on Windows systems, while Scaphandre can work on both Windows and Linux systems. Second, CPU-Z can only measure the power consumption of the CPU, while Scaphandre can measure the power consumption of any process or application. Third, CPU-Z can display power consumption data on its graphical user interface or a text file in a poor exploitable format, while Scaphandre can send or expose power consumption data to various data analysis or monitoring tools.

Intel Power Gadget VS Scaphandre

Intel Power Gadget [20] is a software-based tool that can monitor and estimate the real-time processor package power information in watts using the energy counters in the processor. It works on both Windows and mac-OS systems and supports Intel Core processors from the second generation up to the tenth Generation. However, Intel Power Gadget also has some limitations compared to Scaphandre. First, Intel Power Gadget only works on Intel processors, while Scaphandre can work on any processor that supports RAPL (Running Average Power Limit) interface. Second, Intel Power Gadget can only measure the power consumption of the processor package, while Scaphandre can measure the power consumption of any process or application. Third, Intel Power Gadget can display or log power consumption data on its GUI, command line tool (PowerLog) or a CSV format however it dumps the data of the processor package, while Scaphandre can send or expose power consumption data to various data analysis or monitoring tools of a specific process.

Conclusion

In summary, Scaphandre has several advantages over other watts meters in terms of platform compatibility, measurement granularity (up to one nano-second sampling), and data output flexibility (JSON). Scaphandre can work on both Windows and Linux systems and support any processor that supports RAPL interface. Scaphandre can measure the power consumption of any process or application on bare metal hosts or virtual machines. Scaphandre can send or expose power consumption data to various data analysis or monitoring tools for further processing and visualization (JSON, Grafna, wrap10, etc). Therefore, Scaphandre is a better choice for measuring power consumption in different scenarios including this thesis and any possible future improvements.

4.3 GreenErl for Windows

After deciding on the tool to be used (Scaphandre), I did explore the previous Unix-like OS GreenErl framework [6], tried different implementations and decide how to edit it and integrate Scaphandre. During this section, I will talk about each part and give the final implemented result.

Previous Framework

The previous framework consisted of three components.

Rapl-read.c

It is responsible for reading the energy consumption however they modified the `rapl-read.c` program to measure the energy consumption of Erlang functions. Split each measurement function into two parts: one that reads the values before

the Erlang function is run, and one that reads the values after and calculates the energy difference.

Erlang module

The `energy_consumption.erl` module, is an Erlang module that communicates with the `rapl-read.c` program to measure the energy consumption of Erlang functions. The module has a measure function that takes various parameters, such as the path to the `rapl-read.c` program, the functions to measure, the inputs for the functions, the number of repetitions, and the log file location. The module can also generate inputs from input descriptors and measure all exported functions in a module.

Python GUI

The Python GUI, is a tool that helps to organize and visualize the measurements. The GUI uses the TkInter library to create a user interface, where you can select the Erlang file, the functions, and the inputs to measure. The GUI also allows building the `rapl-read.c` program and setting up the measurements. It allows adding multiple measurements to a queue and running them one by one. After the measurements are done, you can use the matplotlib library to plot the results on a graph or export the results to a latex graph.

Summary

To summarise, the different components of GreenErl communicate with each other. The Python script uses the subprocess library to create the Erlang shell and run the commands. The Erlang module spawns the `rapl-read.c` program and sends signals to start and stop the measurement. The `rapl-read.c` program reads the RAPL registers and sends the measured data back to the Erlang module.

Finally, the Erlang module creates the log files, which are accessed by the Python script to visualize the results.

Conclusion

As all components of the framework are well-written and easily expendable, and the only missing part for the Windows framework is the `rapl-read.c` module. Thus I tried to fill this gap between the two operating systems using Scaphandre.

GreenErl for Windows implementation

My first objective in adapting the GreenErl framework to work on Windows was to preserve the same functionality and input specifications as the original version (Linux-based GreenErl). This meant that I wanted to use the existing Erlang modules as inputs for the Erlang measure module without modifying them, if possible since the goal was to compare the energy consumption of the two operating systems. I began by removing the unused functions and parameters (to work on Windows) from the Erlang measure module to obtain a minimal command-line-based version of GreenErl (without energy consumption data). Then I integrated the Scaphandre component and modified the Python script to execute the Erlang module. Finally, I adjusted the Python script to visualize and export the results as latex files. The new implementation consisted of three main components: an Erlang module, a Python script and a Scaphandre command for collecting the measurements.

Erlang Module

The Erlang module named `energy_consumption_res.erl` exports a single function **measure/3**, This function takes three arguments: `{Module, Functions, InputDescs}, Count, ResultPath`. It measures the en-

ergy consumption of multiple functions for multiple inputs with different repetitions. It calls itself recursively to process the list of input descriptions and calls **measureFunctions/3** to measure each function with each input argument list. It also checks if the module to be measured has a function called **generate_input** that can generate input argument lists from input descriptions, and uses it if available. Otherwise, it treats the input descriptions as argument lists themselves. As an example shown in Figure 4.5, this example uses the **map** module to increment by one the `InputDescs` values in this example 10 and 20 to 11 and 21, it uses the **recursive** implementation and saves the results in `ResultPath` in this example in the `result` folder under the `C` drive. the `Count` is equal to 10 which means we repeat the measurements 10 times.

```
1 energy_consumption_res:measure({{map, [recursive], [10,20]},10,
    "C:\\result\\").
```

Figure 4.5: A code snippet from the Erlang measure module

The input description can be of the form `{Value, Count}` where Value is the input and Count is the number of repetitions for that single Value with that function. I added this feature to work optionally if existed in the input description since some functions are extremely fast and have low energy consumption (approximately 0 Watts). If the pattern {Value, Count} is not present then we use the default `Count` present in the **measure/3** parameter.

measureFunctions/3 This function takes three arguments: `{Module, Functions, Attributes, InputDesc}, Count, ResultPath`. It measures the energy consumption of each function in the `Functions` list with the `Attributes` list as input arguments. It uses the `os:cmd` module and regular expression to spawn and get the PID of Scaphandre to monitor the power consumption and writes the results to a JSON file in `ResultPath` folder. Each JSON file has the name `Module_Function_Attribute.json`, in this way, I was able to know the measurement values for each function. It also uses **measureFunction/5** to run the function and measure the time elapsed for that function call and writes

the time to a CSV file using the function **dumpTime/3**. It calls itself recursively to process the list of functions.

measureFunction/5 This function takes five arguments: `{M,F,A,O}, Count, Me, InputDesFile, Time`. It uses recursion and **timer:tc/3** to measure the time elapsed for calling the function `F` from module `M` with arguments `A`. It adds the time to the accumulator `Time` and returns it when `Count` reaches zero. It also sends a stop message to **measureFunctions/3** when done, which triggers the termination of the Scaphandre process using the `Pid` that was collected earlier.

dumpTime/3 This is a helper function that takes three arguments: `FilePath, FileName, Data`. It writes the `Data` (time) string to a file with the given `FileName` in the given `FilePath`. It uses **file:open/2**, **io:format/2**, and **file:close/1** to perform the file operations.

Python script

Similarly to the Erlang module changes, I removed the Windows non-used functionalities from the GUI part shown in the measurement Figure 4.6 and the visualization Figure 4.7. Using the TkInter library for Python, the graphical user interface (GUI) was constructed. This library facilitates the design of the user-friendly interface that enables us to choose the Erlang file with the module to be measured. We can then indicate the functions and the inputs for measurement. Moreover, we can provide the path to the Erlang measurement module and the folder name for the results that will include the JSON files. By filling out the necessary fields, we can set up a measurement and append it to a queue of measurements. I also fixed the path differences between Windows and Linux, removed the non-used packages and updated the LaTeX exporter. I created a function that calculates the consumption values from those JSON files in the directory and saves

them to the CSV files. It takes four arguments: `folder_path, count, input, pid`. It calculates the measurements based on the `Pid` provided as an argument to eliminate the wrong values and get the precise results of the exact process level consumption and not being affected by other Erlang processes running in the background if they exist. It also saves errors in the log folder in case of empty values in the JSON file or if the JSON file is not created. The visualization Figure 4.7 serves to export the Latex files and to visualize the results. We just need to provide the path to the CSV files or the whole folder. The whole interaction between the different parts is illustrated in Figure 4.8.

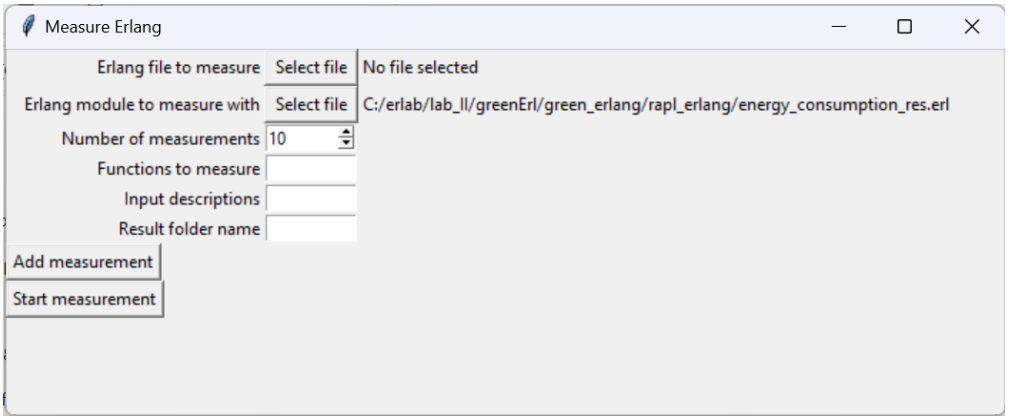


Figure 4.6: Python GUI to run the tool

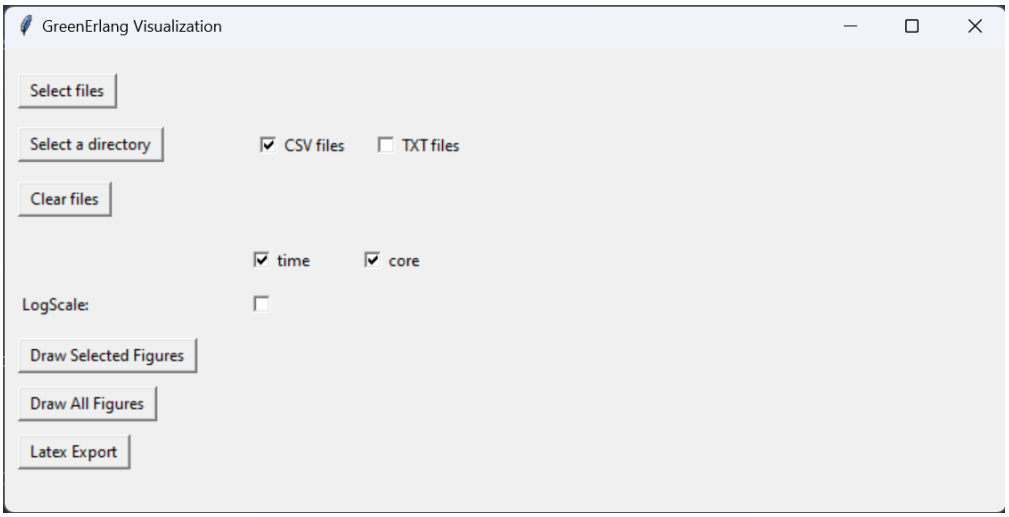


Figure 4.7: Python GUI to visualize

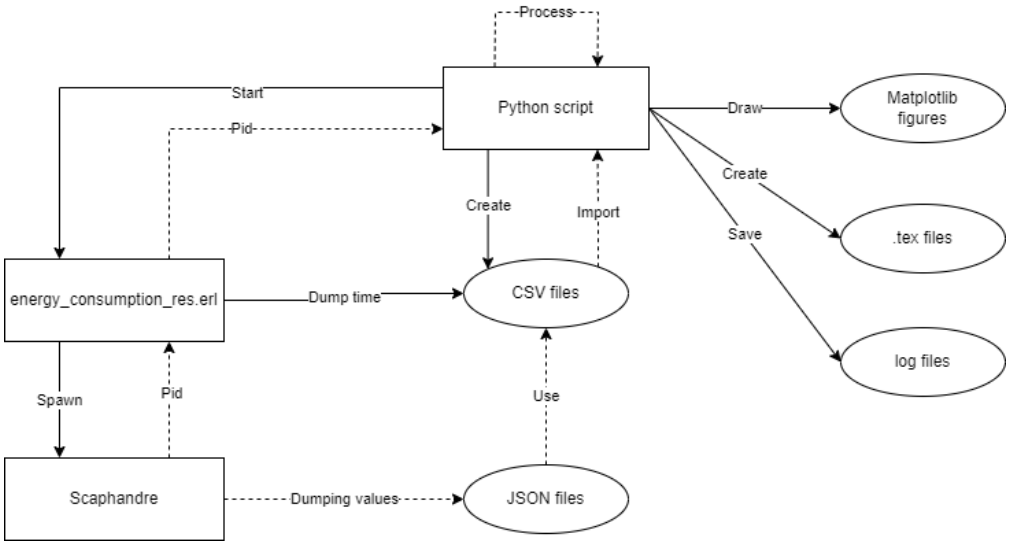


Figure 4.8: Workflow diagram of GreenErl on Windows

The use of Scaphandre

The Scaphandre tool was used in the Erlang measure module. Keeping track of the module function and input description was operated with the names as the next Figure 4.9 is showing.

```
1  InputDesFile = "\"" ++ ResultPath  ++ atom_to_list(Module) ++ "
       _" ++ atom_to_list(Function) ++
2                      "_" ++ integer_to_list(InputDesc) ++ ".json
       "++ "\"",
3       io:format("~nCurrently measuring functions with input
       desctription ~p~n",[InputDesc]),
4       % ns sampling
5       Command = "scaphandre json -s 0 -n 1 -m 100 -f " ++
       InputDesFile,
6       Output = os:cmd("wmic process call create \""++ Command ++"
       \" | find \"ProcessId\""),
7       {match, [PidString]} = re:run(Output, "ProcessId = ([0-9]+)
       ", [{capture, all_but_first, list}]),
8       Pid = list_to_integer(PidString),
```

Figure 4.9: A code snippet from the Erlang measure module

This section of the code (Figure 4.9) is responsible for preparing the input JSON file name and the Scaphandre command. Once ready to spawn Scaphandre I used `wmic process call create` which is a command from Windows Management Instrumentation (WMI) [40] to create the Scaphandre process and then I filtered the output of the command using `find "ProcessId"`. This resulted in getting the PID of the newly spawned Scaphandre. Having the PID of Scaphandre is crucial since we need to kill it by its PID once the function is finished. The communication process is illustrated in Figure 4.10.

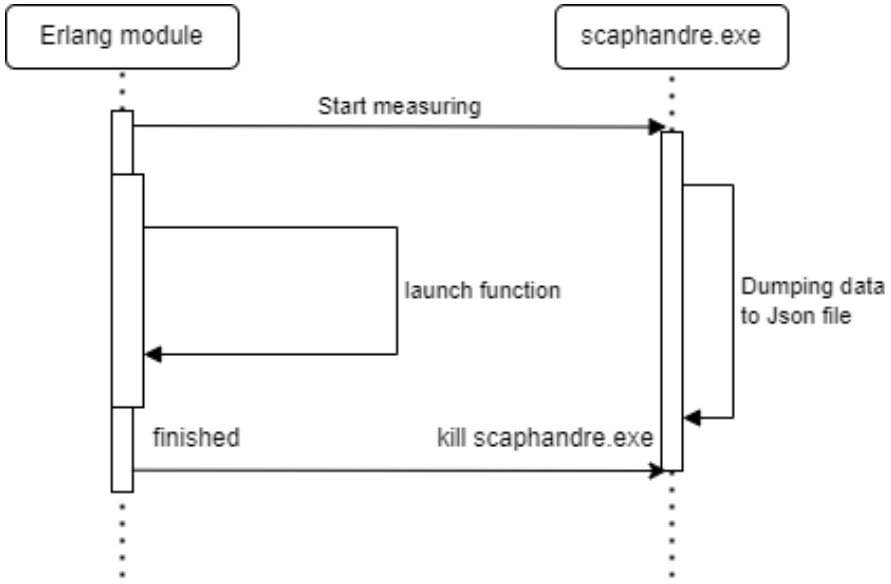


Figure 4.10: Sequence diagram of communication between the Erlang module and Scaphandre process

To use the JSON exporter feature provided by Scaphandre [41] and write the results into a JSON file I followed this approach. I provided the path to that file including the file name. As an example shown in Figure 4.11 having a module named **map** and a function as **recursive** and an input description as **100 000** the resulted JSON filename should look like this: **map_recursive_100000.json** which was used later by the Python script to calculate the function energy consumption.

```
scaphandre json -t 10 -s 0 -n 1 -m 100 -f result_path\
map_recursive_100000.json
```

Figure 4.11: JSON export example

Scaphandre also provides the possibility to gather consumption of the top 20 consumers by default, however as shown in Figure 4.11, I explicitly defined to use the top 100 consumers. After many experiments and trying different numbers of consumers, 100 was a good configuration that can cover all Erlang functions that needed to be measured.

The last feature I needed to define in order to have optimal measurement and coherent results is the sampling rate. By default in Scaphandre, this feature is set to two seconds [41], which was not sufficient in our case since some functions are fast and needed a larger sampling rate. I tried different sampling rates and ended up choosing the highest which is nano-second sampling as shown in Figure 4.11. The nano-second sampling resulted in a higher granularity of details in the graphs this can be considered a two-sided sword, on one hand, we have a detailed graph and on the other hand, it might result in more fluctuations. I ended up choosing the highest sampling rate because it gave better results on average.

Just for the purpose of the experiment while deciding on the final configurations, I repeated two different implementations to build a list 10 times and plotted the results. The first Figure 4.12 shows the energy consumption and the second Figure 4.13 shows the runtime results. As we can tell from both graphs the overall shape of both functions is consistent, we had few fluctuations but this is comprehensible since this might be linked to some process-level scheduling or CPU time-sharing, we can also notice that when we have a drop in runtime it also resulted in a drop in the consumption.

These results were good proof that the framework is solid and stable as well as ready to start using it for future measurements.

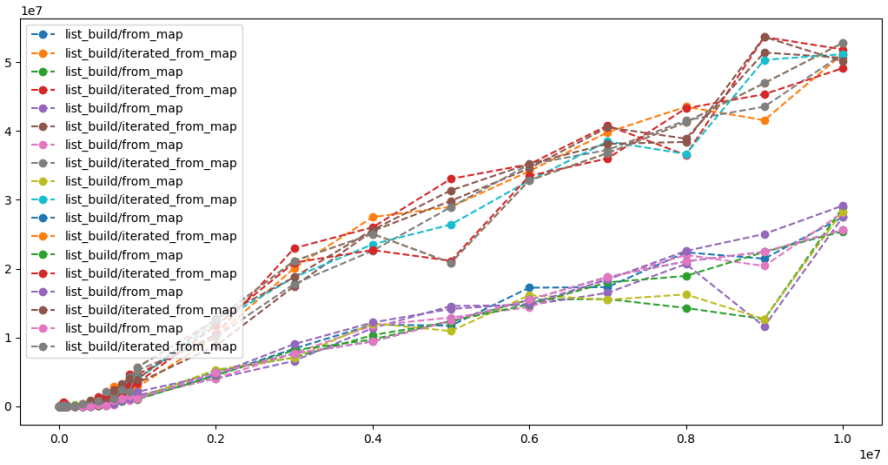


Figure 4.12: Energy consumption of building a list 10 times in two different implementations

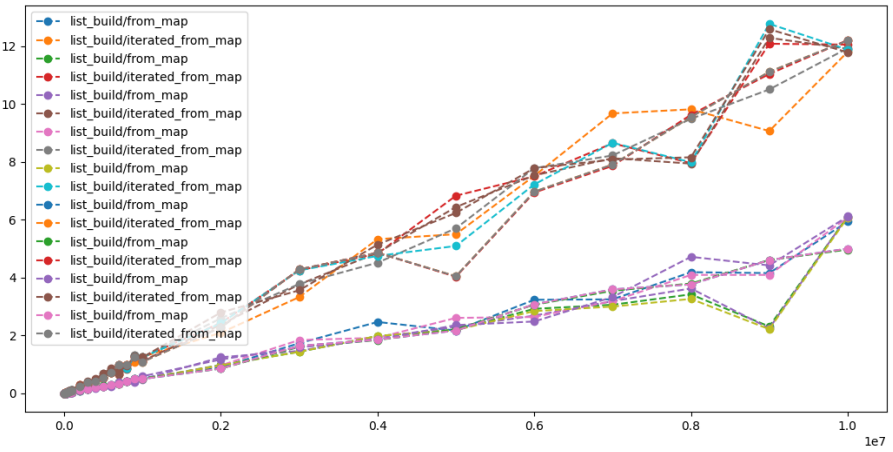


Figure 4.13: Runtime of building a list 10 times in two different implementations

Chapter 5

Measuring energy consumption

This chapter provides a comprehensive overview of the methodological approach adopted for the empirical investigation of the energy consumption of various functions, language elements, constructs and programming methods. The rationale behind the selection of these features is explained by referring to the previous findings in the literature. The chapter also reports and discusses the outcomes of the experimental measurements.

5.1 Methodology

The observed investigation of the energy consumption of various functions was carried out using the GreenErl framework for Windows, which was presented in Chapter 4. Each experiment was repeated 10 times. This number was set from the previous measurements based on the analysis of the standard deviation and the range of the measured values, which indicated a high degree of consistency. The data obtained from the experiments were plotted and subjected to descriptive statistical analysis. The runtime of the functions was also measured.

All measurements in this research were performed on a DELL XPS 13 9370 with an Intel Core™ i5-8250U CPU @ 1.60GHz 1.80 GHz, 8GB of DDR3 RAM, 64-bit operating system, x64-based processor, running Windows 11 Pro operating system Version 22H2. I used Erlang/OTP 23.3.4 for the measurements.

Before starting each measurement, I was sure that the laptop is:

- Fully charged
- Plugged into the AC
- The screen brightness is set to the minimum level
- Wi-Fi was turned off
- Laptop was in test mode (required by Scaphandre)
- Power mode was balanced
- Not using the laptop during the measurements
- Set off the screen and sleeping features both on the battery and when plugged

5.2 Previous measurements

The previous work [6] conducted a first-hand study on the energy consumption of complex algorithms in Erlang. They compared different implementations of the N-queens problem and sparse matrix multiplication [42], using various techniques such as higher-order functions, recursion, list comprehensions, lists and arrays. Other work [6] was on more basic language elements, rather than complex problems.

In this measurement, I focus on comparing different implementations of the same language element. Based on the previous findings, I selected the following areas for this analysis:

- Data structures
- Higher order functions
- Parallel language constructs
- Algorithmic skeletons

5.3 Data structures

The objective of this study was to evaluate the energy efficiency of various data structures for storing key-value pairs on Windows. I conducted a series of experiments to measure the energy consumption of different operations on these data structures and to analyze the trade-offs of changing the representation by converting the data to another format. Furthermore, I compared the energy performance of alternative methods for executing the same operation within a given data structure. The data structures that we investigated in this study were:

- List of tuples (or proplists)
- Map
- Dictionary

The operations I measured on these data structures are the following:

- Creating/Converting the data structures
- Finding the value belonging to a particular key
- Inserting a new key-value pair
- Updating the value belonging to an existing key
- Deleting an existing element

List build

In this part, I compared measuring the cost of building and converting lists in different implementations.

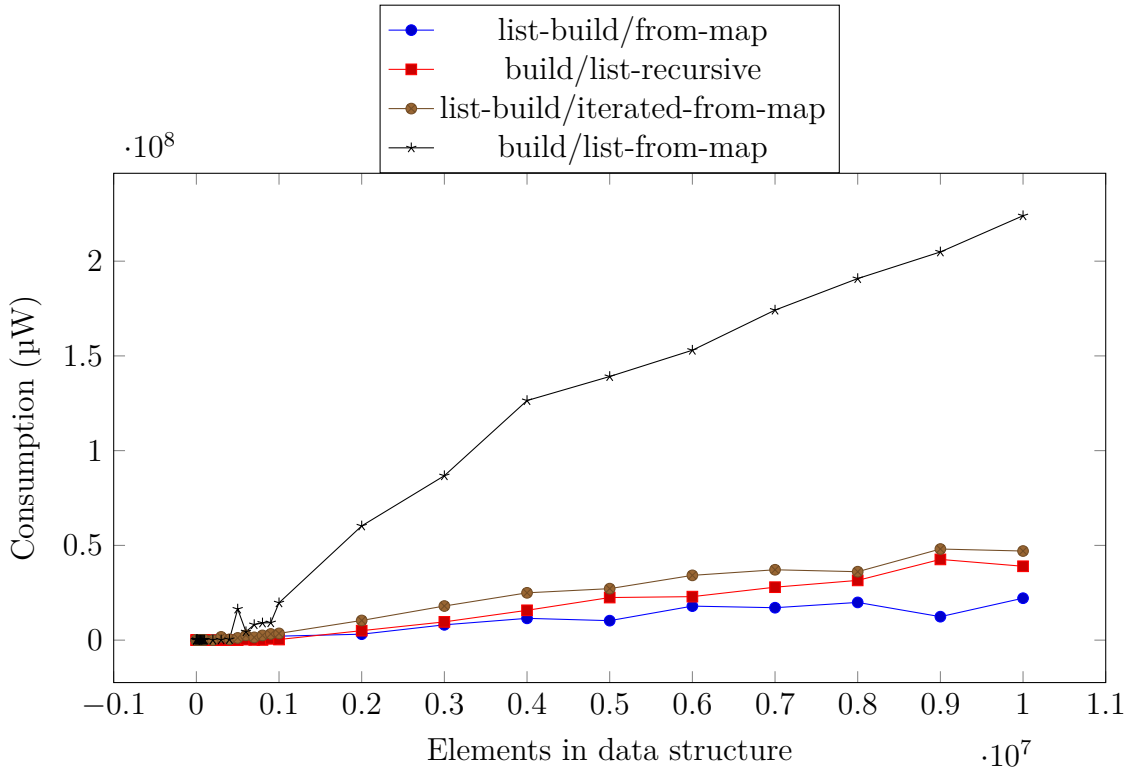


Figure 5.1: Energy consumption of building/converting a list

In summary, **from_map/1** takes an existing map and returns a list of its key-value pairs while **list_from_map/1** creates a new map with keys and values from 1 to N and returns its key-value pairs as a list of tuples. This difference in the implementation can be seen in both Figures 5.2 and 5.1 which shows that creating a new list from a new map of N values is the most consuming way of creating a new list. This is a result of a bad design in the algorithm, which developers need to avoid since it leads to a serious increase in energy consumption.

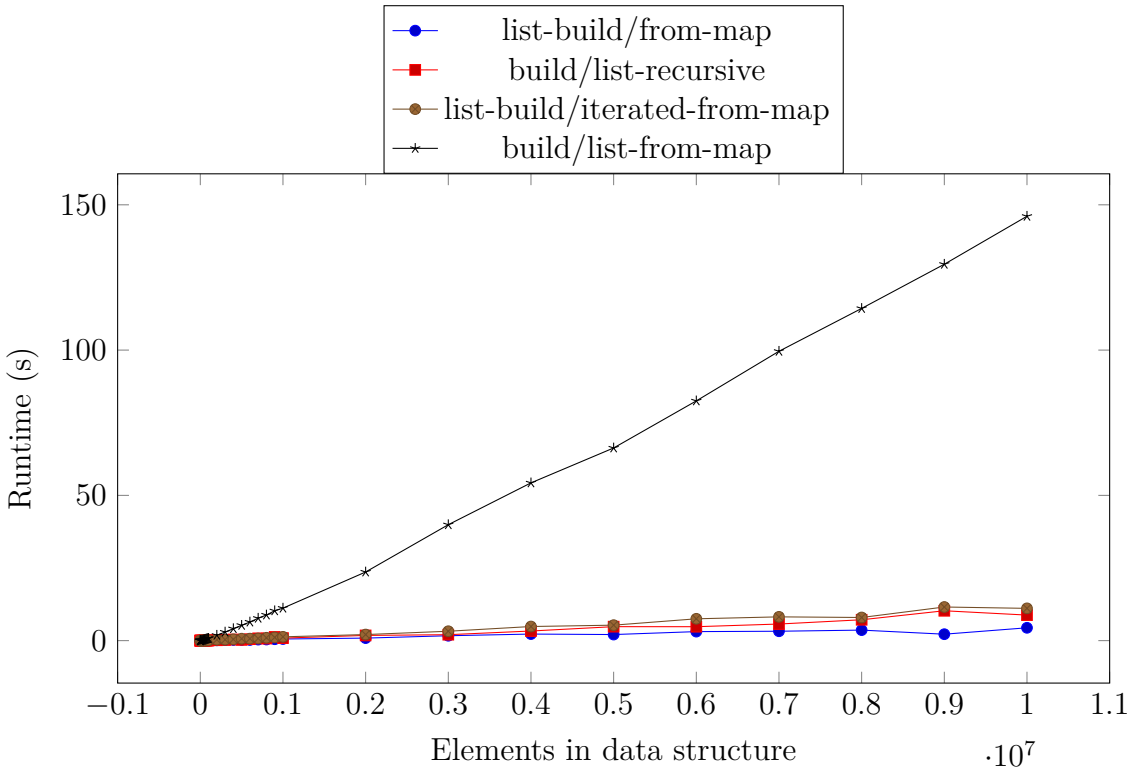


Figure 5.2: Runtime of building/converting a list

Map build

In this part, I will compare measuring the cost of building and converting maps in different implementations.

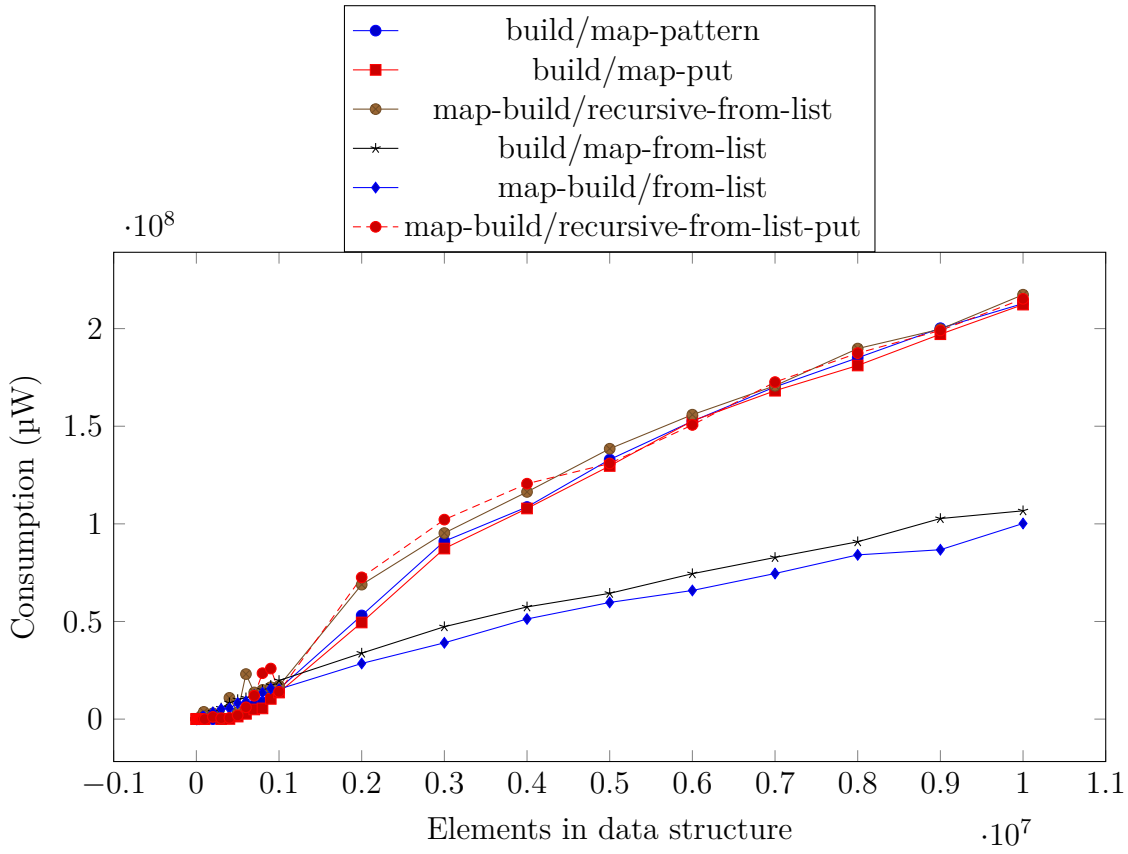


Figure 5.3: Energy consumption of building a map

As shown in the energy consumption of building a map (Figure 5.3), I can deduce that the reason behind **map_build:from_list/1** and **build:map_from_list/1** using less energy can be that they use the **maps:from_list/1** function, which is specifically designed to convert a list of key-value pairs into a map. This function is implemented in an efficient manner that takes advantage of the underlying data structures and algorithms used in the Erlang VM. On the other hand, **map_build:recursive_from_list/1** and **map_build:recursive_from_list_put/1** use recursive functions to build the map, which creates a large number of intermediate maps and consume a significant amount of memory. Similarly to the energy consumption, the runtime Figure 5.4

shows almost the same trend over time while increasing the elements in the data structure.

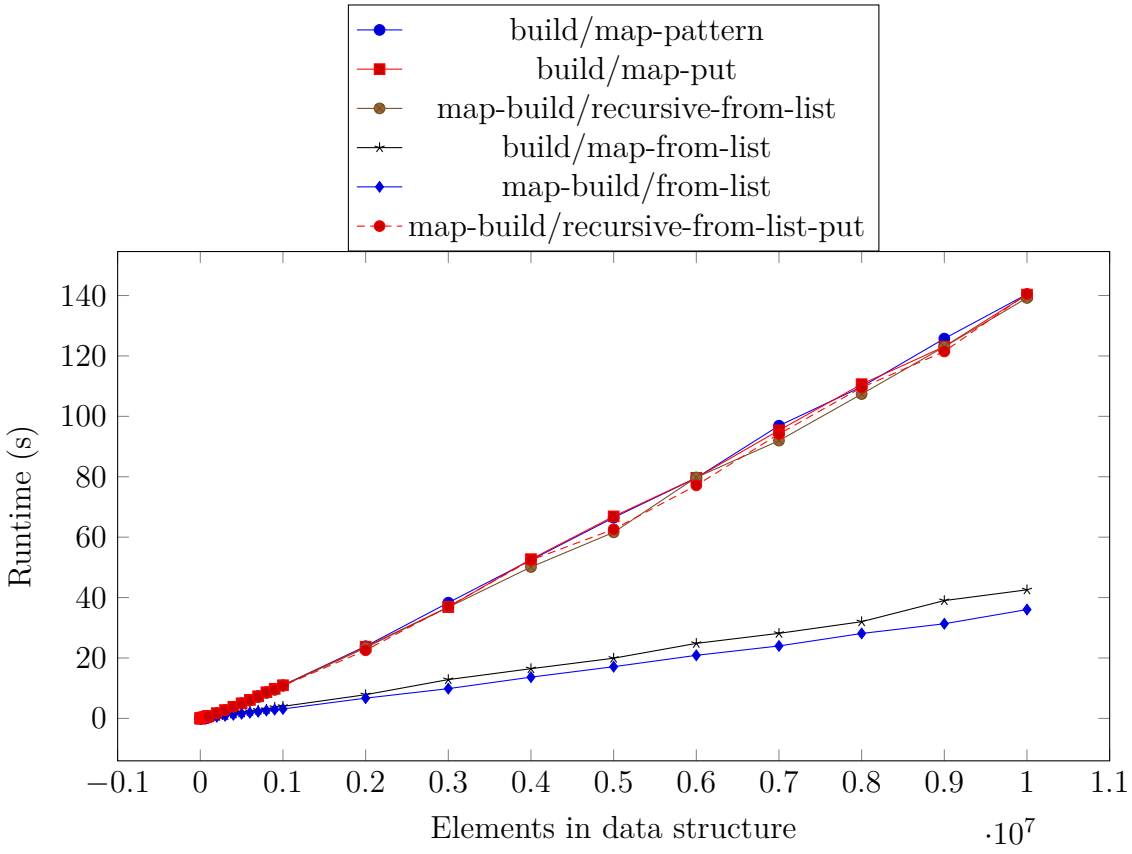


Figure 5.4: Runtime of building a map

Dictionary build

The results of the dictionary acted differently in energy consumption (Figure 5.5) compared to the previous map and list implementations. While the runtime (Figure 5.6) was slowly increasing as the elements in the dictionary increased, the **dict-build/recursive-from-list** was running slower compared to **dict-build/from-list**. The **from_list/1** relies on the OTP library's function **dict:from_list/1** for direct conversion, while **recursive_from_list/1** employs

a recursive approach to construct the dictionary manually. This might be the reason why we can see the peak rise in the first section of the graph and the relatively consistent values later.

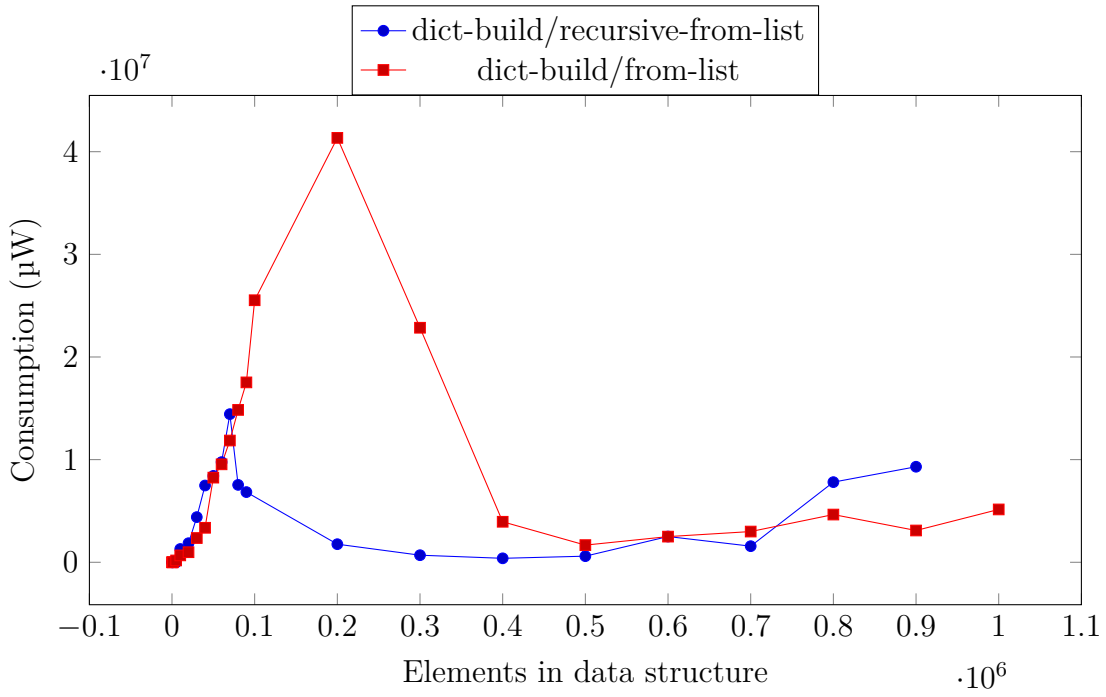


Figure 5.5: Energy consumption of building a dictionary

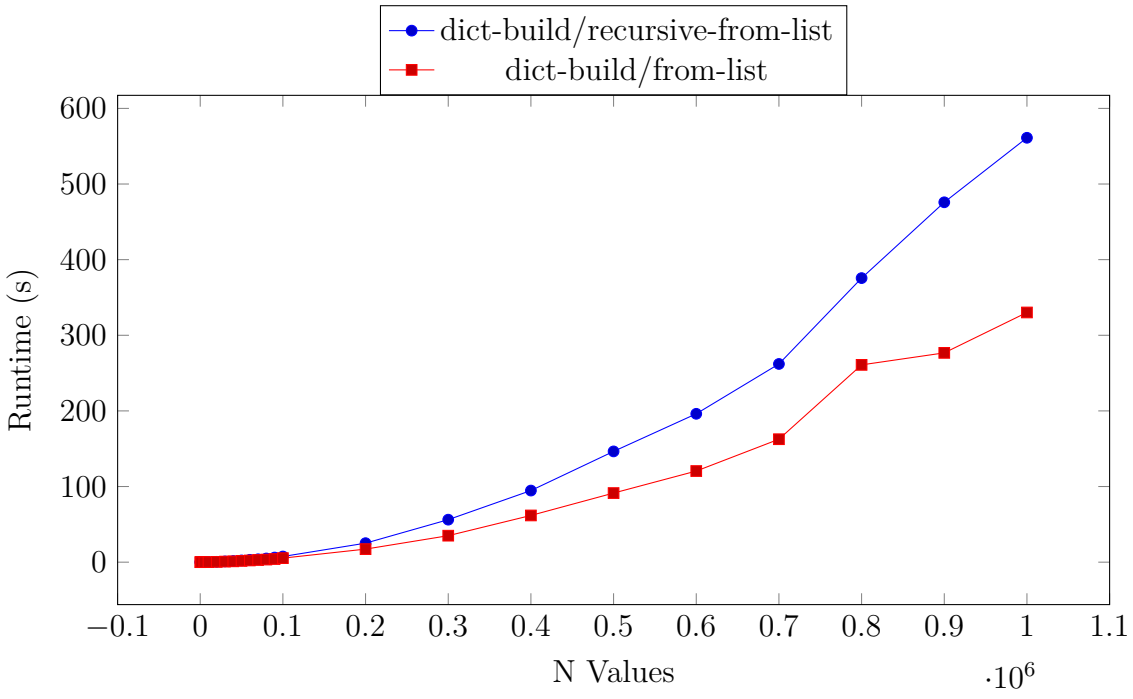


Figure 5.6: Runtime of building a dictionary

Update one element

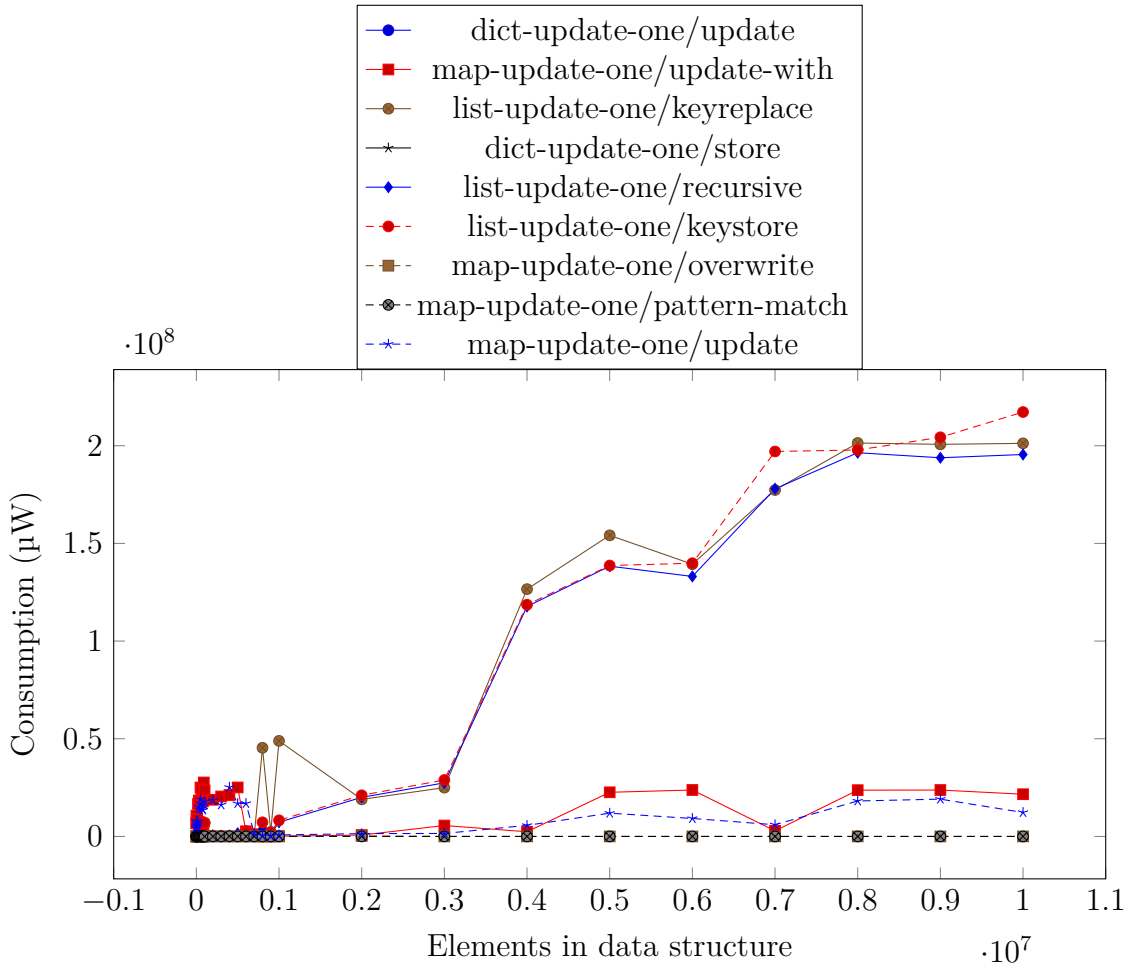


Figure 5.7: Energy consumption of updating one element in different data structures

Updating one element in a data structure's energy consumption is shown in Figure 5.8 and the runtime in Figure 5.10. Looking at both graphs we can see that the general behavior of the energy consumption is following the runtime tendency. Based on the results from Figure 5.7, the list is the most costly to update one element in it with the recursive implementation being the better choice to

use. However, the map and dictionary were the least costly. While using our own implementations such as the **map-update-one/pattern-match** which used pattern matching or recursive **map-update-one/overwrite** shows a relatively lower consumption compared to using OTP library functions such as **map-update-one/update** using **maps:update** [43] and **map-update-one/update-with** implemented using **maps:update-with** [44].

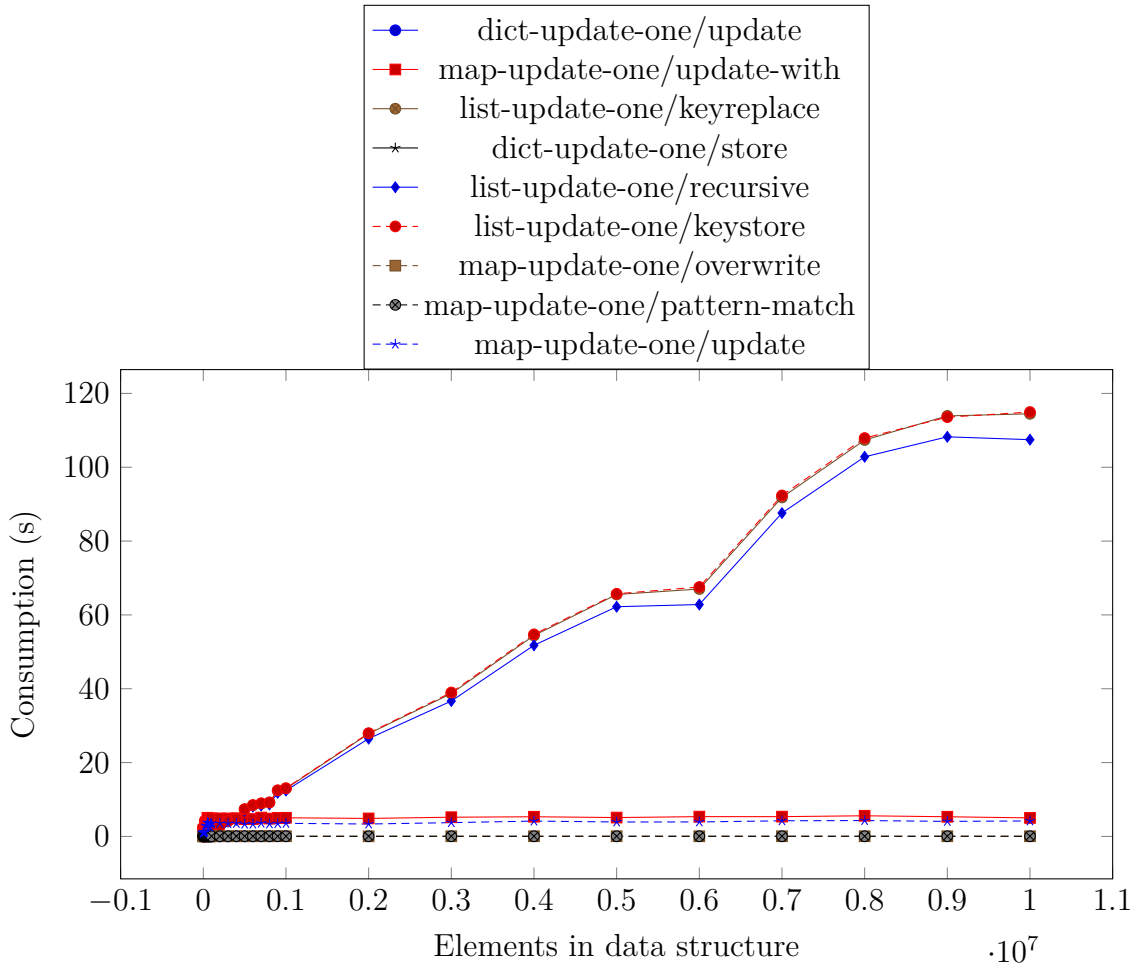


Figure 5.8: Runtime of updating one element in different data structures

Update all elements

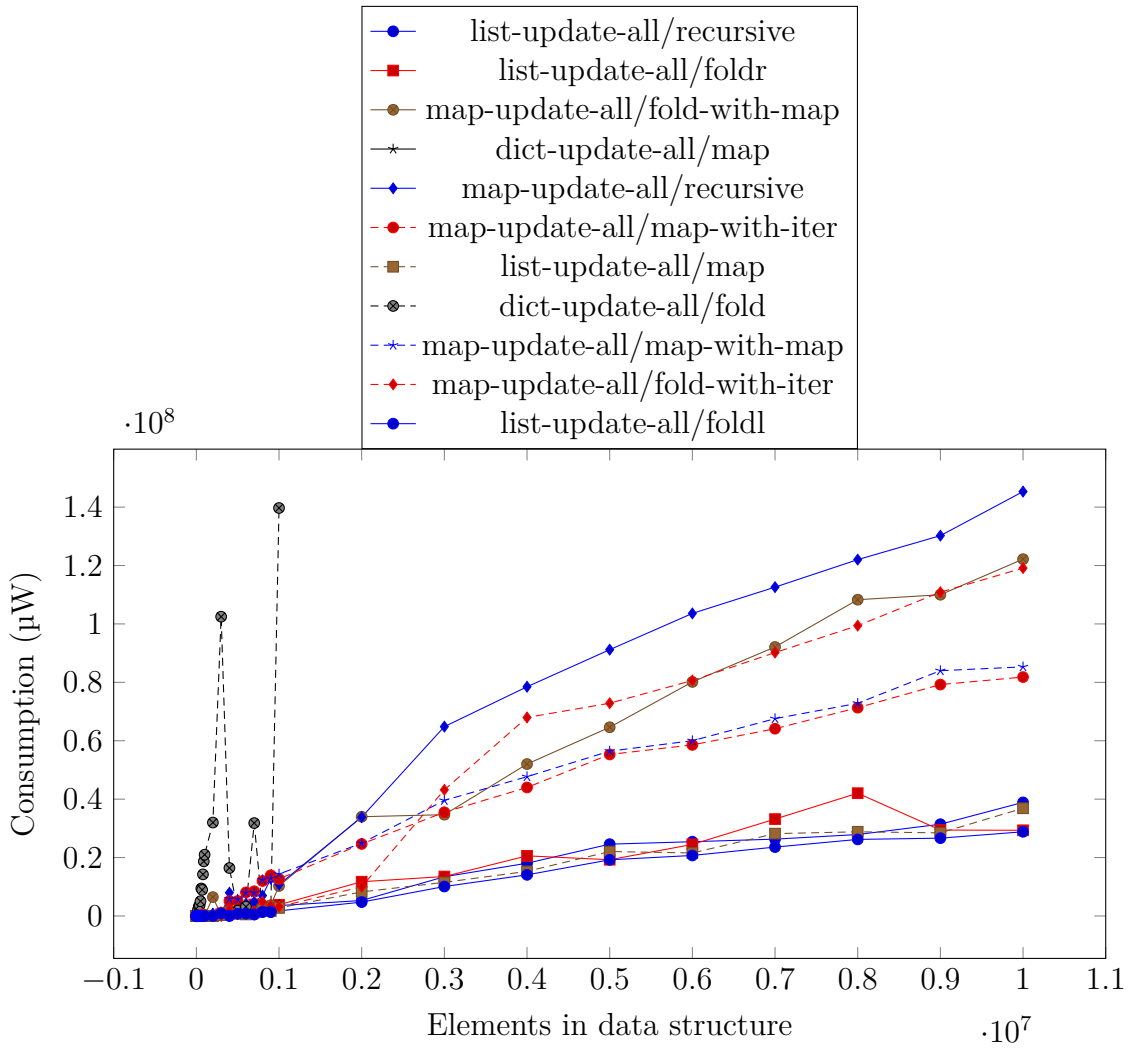


Figure 5.9: Energy consumption of updating all elements in different data structures

Opposite to updating one element, updating all the elements in the data structure shows different behavior. Contrary to using the built-in functions in update one resulting in higher consumption, updating all elements in a map using our own implementation like **map-update-all/recursive** resulted in a higher consumption

while using **maps:fold** built-in function resulted in a lower consumption in maps. However, using a built-in function for a dictionary to update all elements like **dict-update-all/fold** resulted in a high peak of consumption. The Figure 5.9 shows that the runtime of the dictionary is the highest which is why the measurement stopped at 100 000 elements.

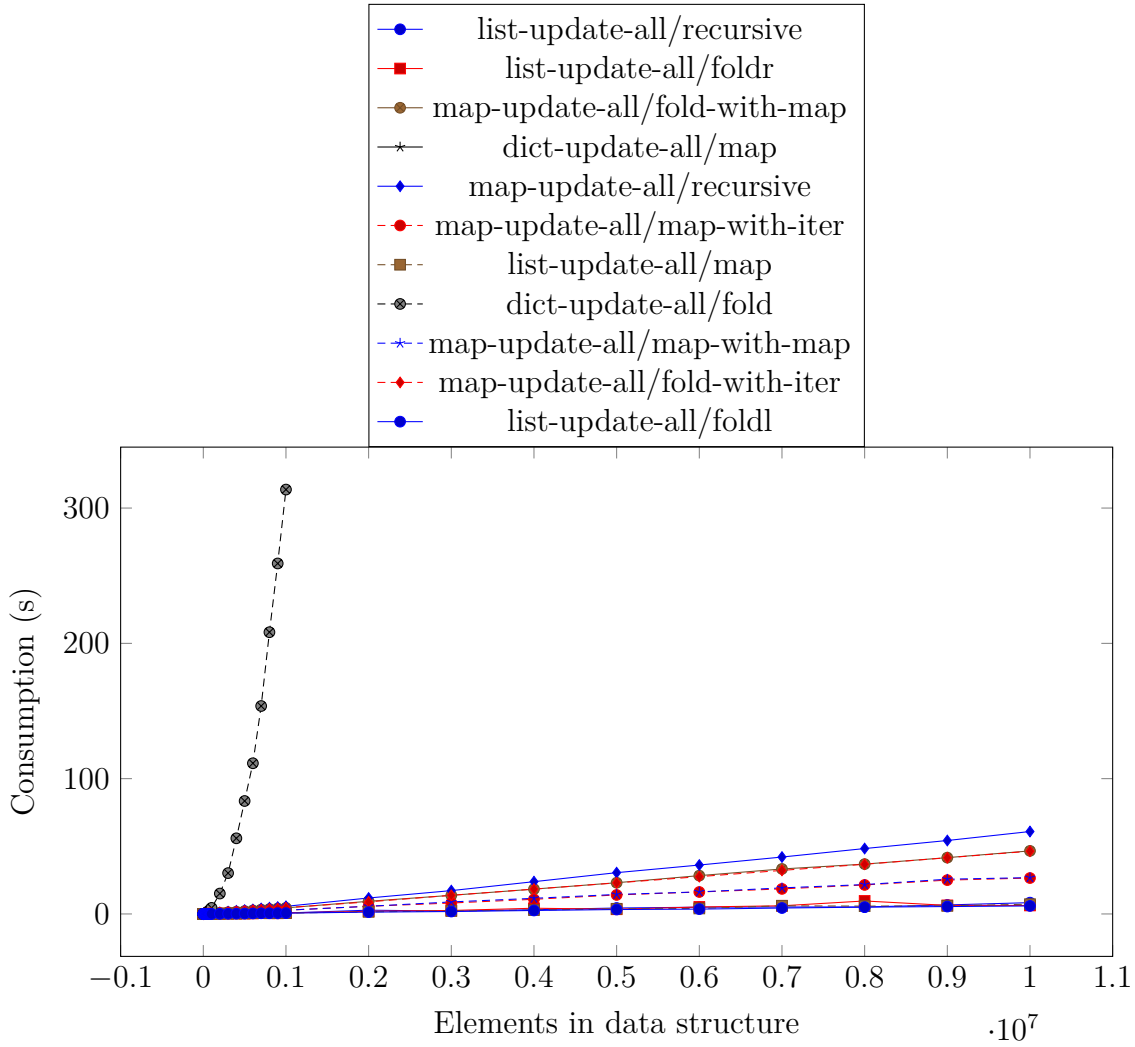


Figure 5.10: Runtime of updating all elements in different data structures

Find an element

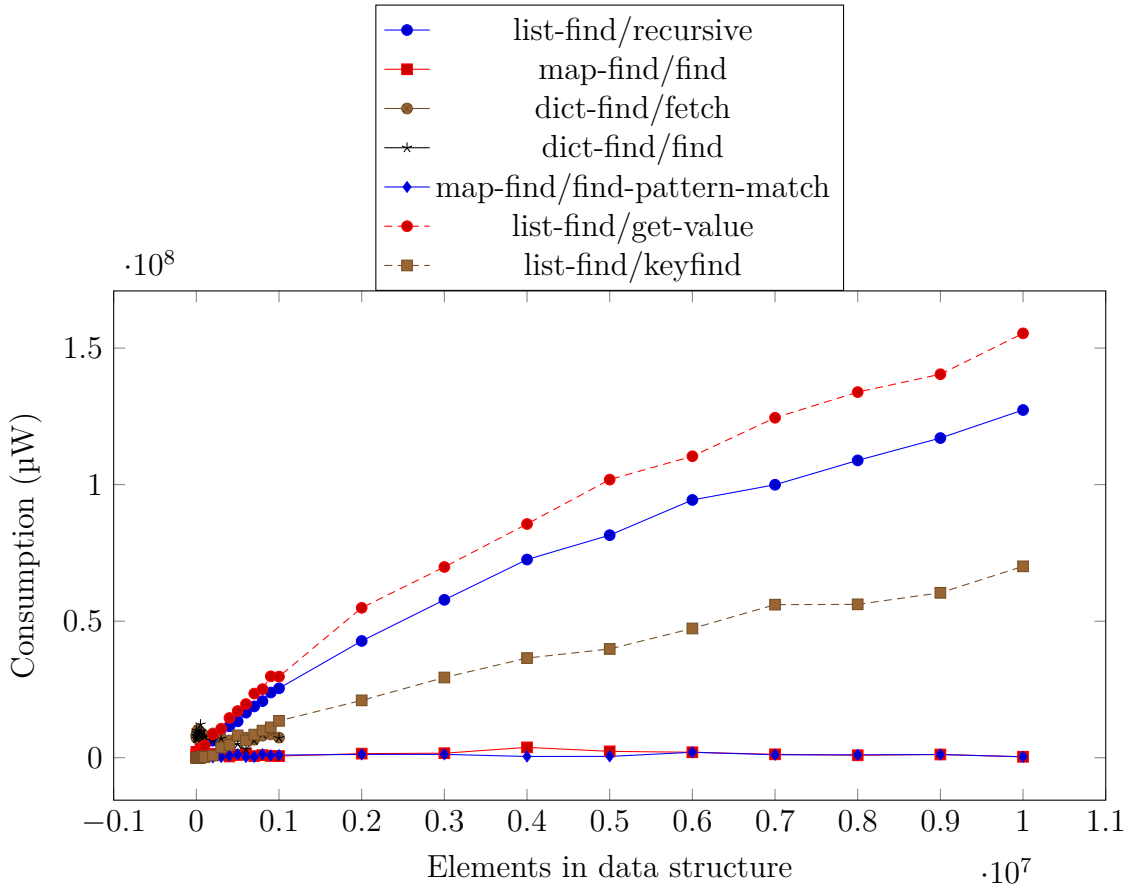


Figure 5.11: Energy consumption of finding an element in different data structures

Based on the result shown in Figure 5.11 and 5.12, finding an element in a list is always costly not only in energy consumption but also in runtime. For instance, using the library function **get_value(key,list)** [45] which looks for the value of the given key in the given list. A lite bit similar to using **get_value(key,list)**, using our implemented recursive function resulted in lower but yet relatively high results. The most optimal way of finding an element in a list is using the **list-find/keyfind** which uses the library function **keyfind/3** [46] to find a tuple within a tuple list. However, using a map is the most optimal way to find an

element with pattern matching being the efficient implementation. Similarly to the map, finding an element in a dictionary was efficient in all implementations.

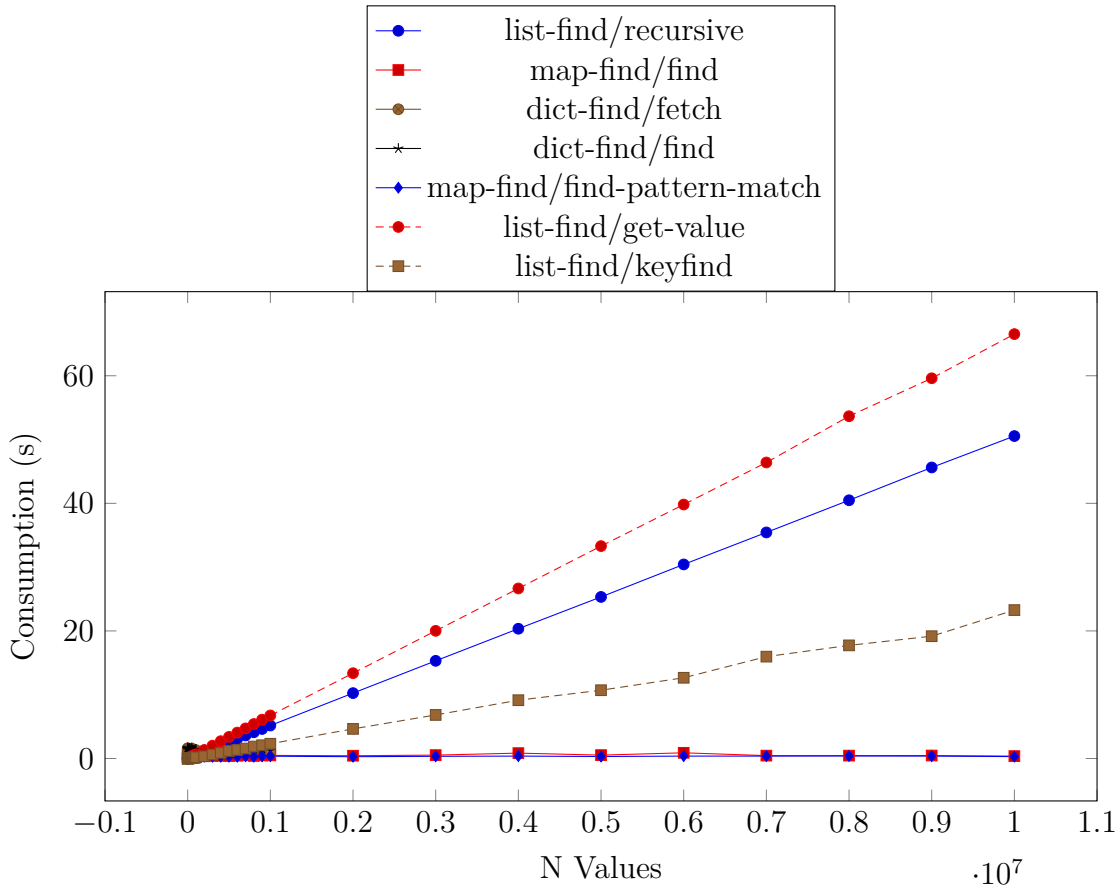


Figure 5.12: Runtime of finding an element in different data structures

Delete an element

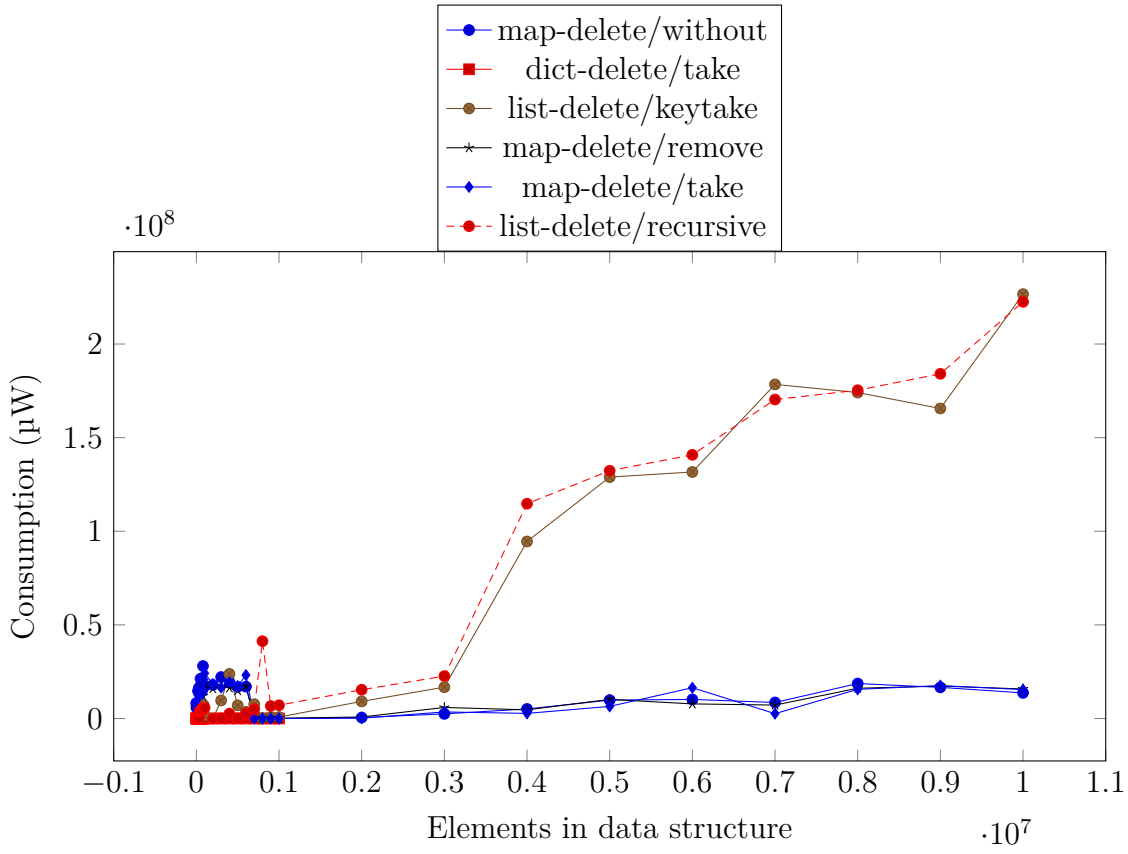


Figure 5.13: Energy consumption of deleting an element in different data structures

Similarly to finding an element, deleting an element is efficient when using a map and a dictionary. However, using the library functions or using our own implementations show no big difference in values, sometimes they are alternating in energy consumption (Figure 5.13) while it is a little bit faster to use the recursive implementation in lists (Figure 5.14).

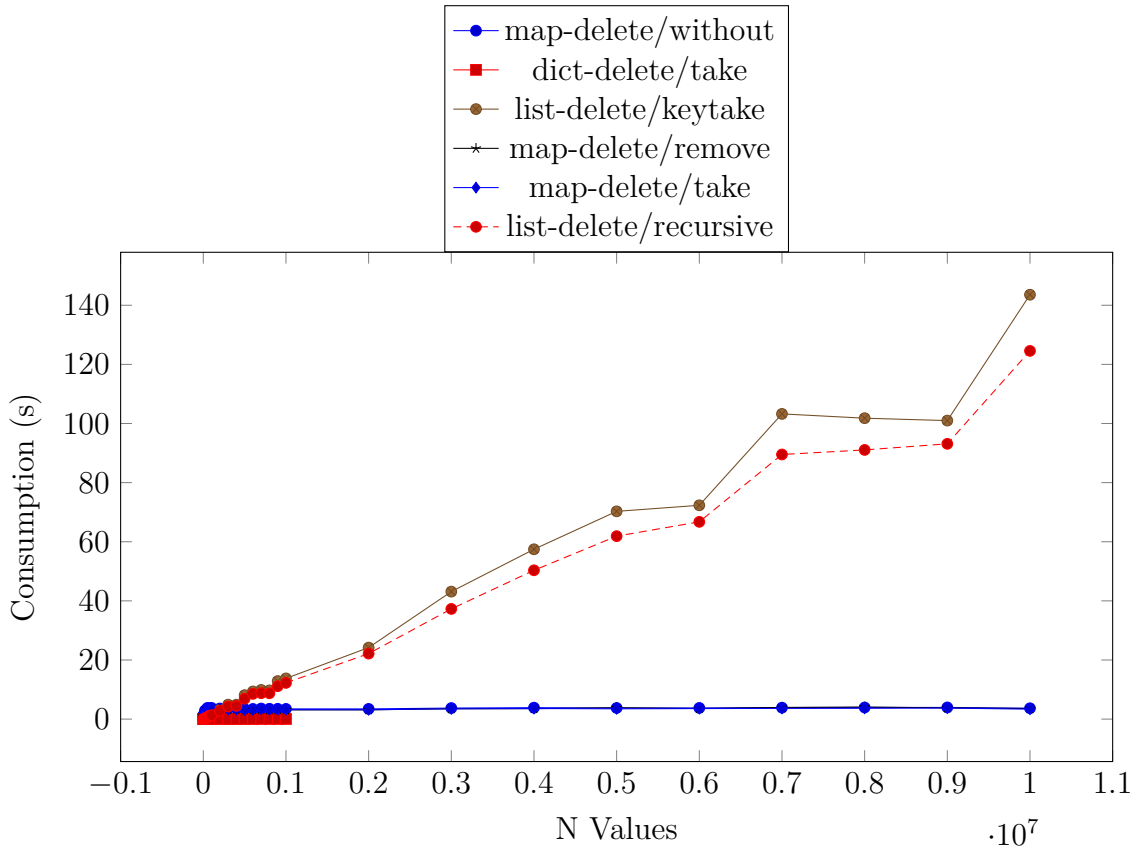


Figure 5.14: Runtime of deleting an element in different data structures

5.4 Higher-order functions

The previous work investigated the influence of higher-order functions (HOFs) on energy consumption. Building on their previous work, they demonstrated that removing HOFs from the N-queens algorithm reduced energy consumption. They further explored this effect with map, filter and their combination. They employed different methods to implement these functions: using the lists module, list comprehensions, recursive functions and their own HOFs [6]. They also examined whether the kind of function passed to the HOF (lambda or named) affected en-

ergy consumption. They experimented with lists of various sizes from 10 000 to 10 000 000 elements. In this section, I will measure all of the different implementations (map, filter, filter-map). For the maps, it increases each element by one. For the filters, it retained only the even elements.

Map

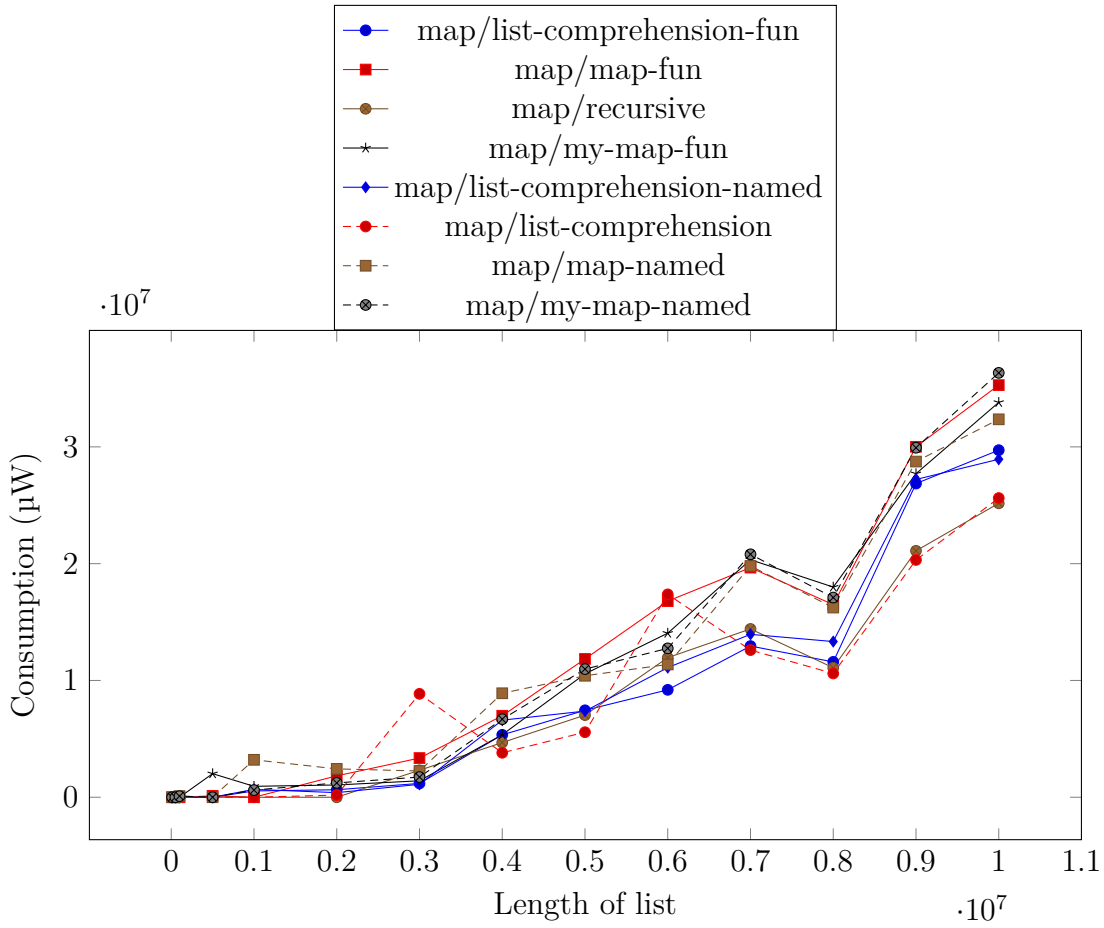


Figure 5.15: Energy cost of different map implementations and function calls

Based on the results from Figure 5.15, we can see that the energy consumption is higher whenever we use a lambda function or a named function. For example, the **map/my-map-named** that uses a named function and **map/map-fun** that uses lambda expression both resulted in alternating higher energy consumption. However, using list comprehension was relatively more efficient despite using named function or lambda expression. The most efficient implementation is using recursive and list comprehension with some fluctuations. One the runtime level, we can see almost the same behavior as shown in Figure 5.16 with a significant drop at input 9 000 000 for all functions.

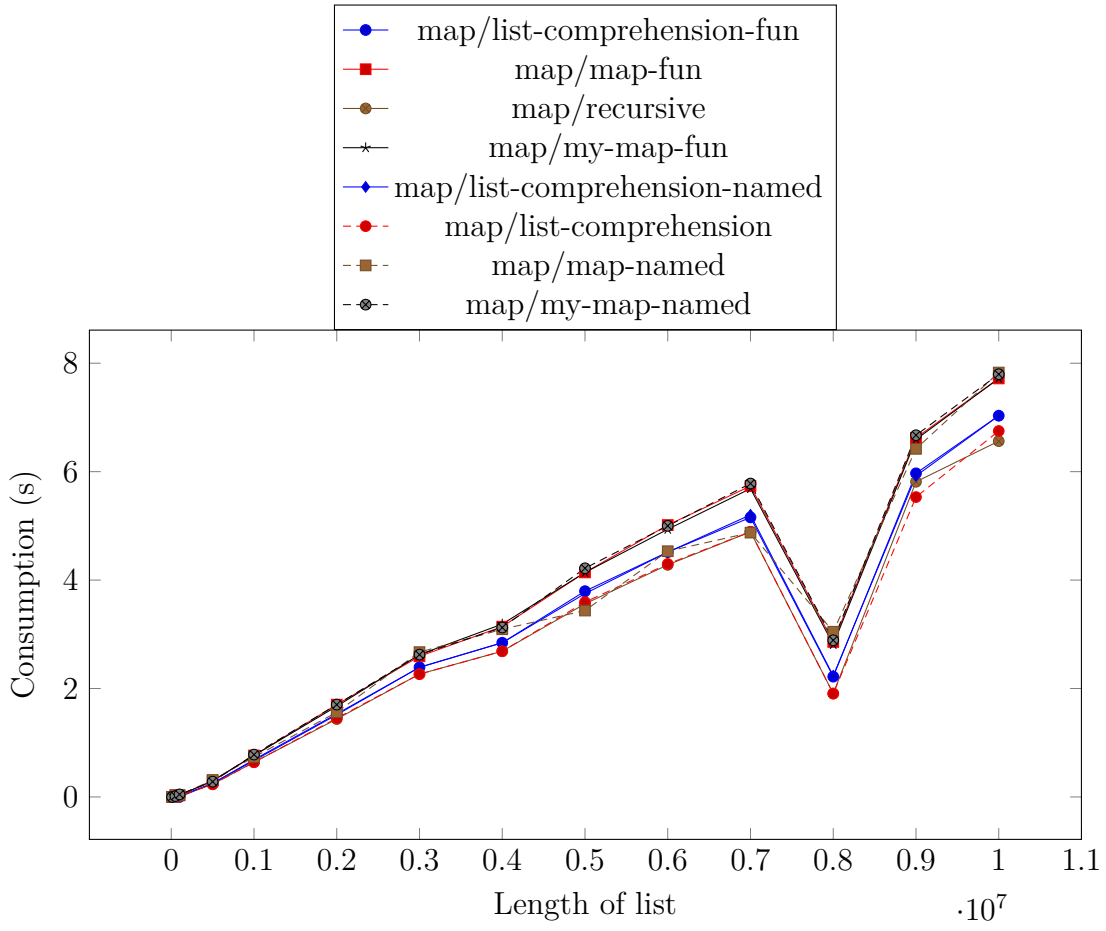


Figure 5.16: Runtime of different map implementations and function calls

Filter

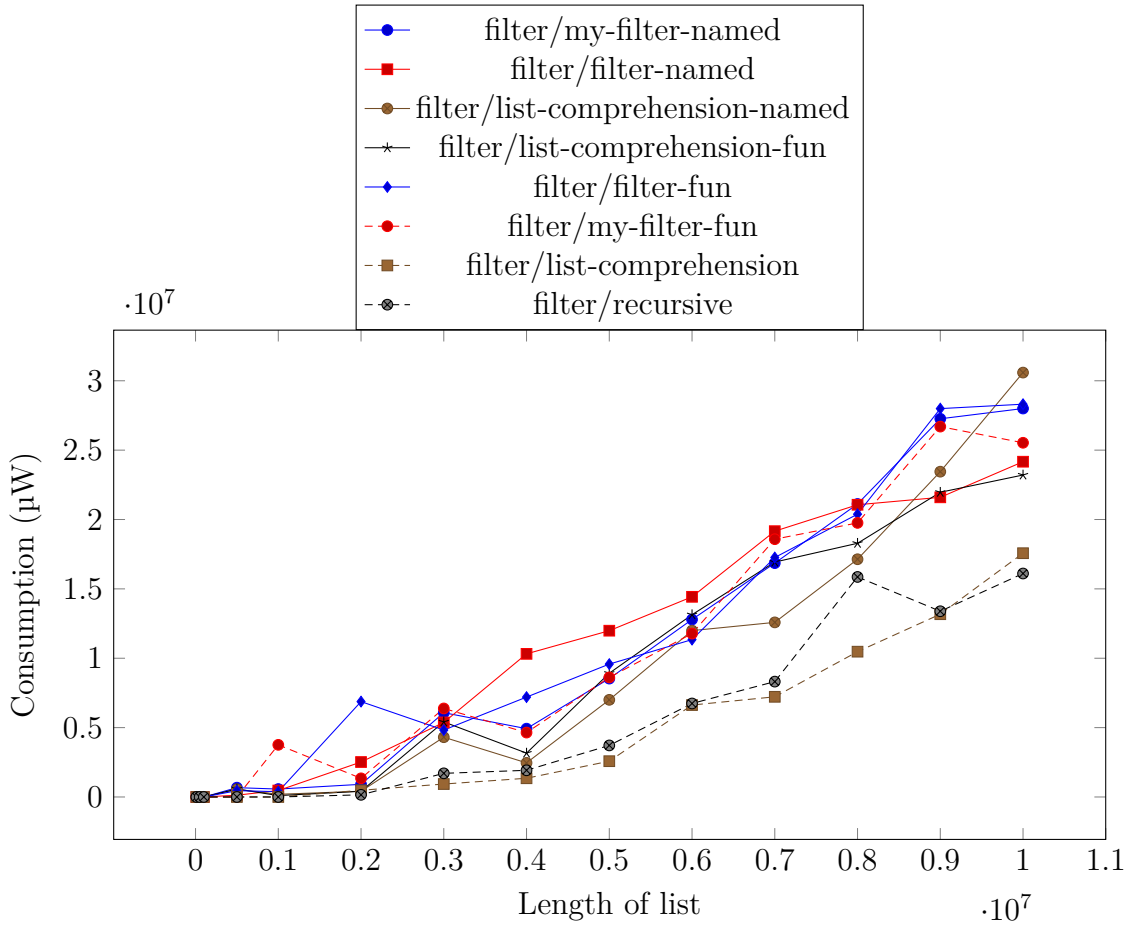


Figure 5.17: Energy consumption of different filter implementations and function calls

Similar to the map measurements, the filter was efficient when used without higher-order implementations, using recursive and list comprehension resulted in the lowest energy consumption as shown in Figure 5.17. For instance, using lambda expression **filter/filter-fun** or named function **filter/my-filter-named** has the higher consumption. For the runtime results provided in Figure 5.18, again recursive and list comprehension were the fastest, followed by the named function

and lambdas.

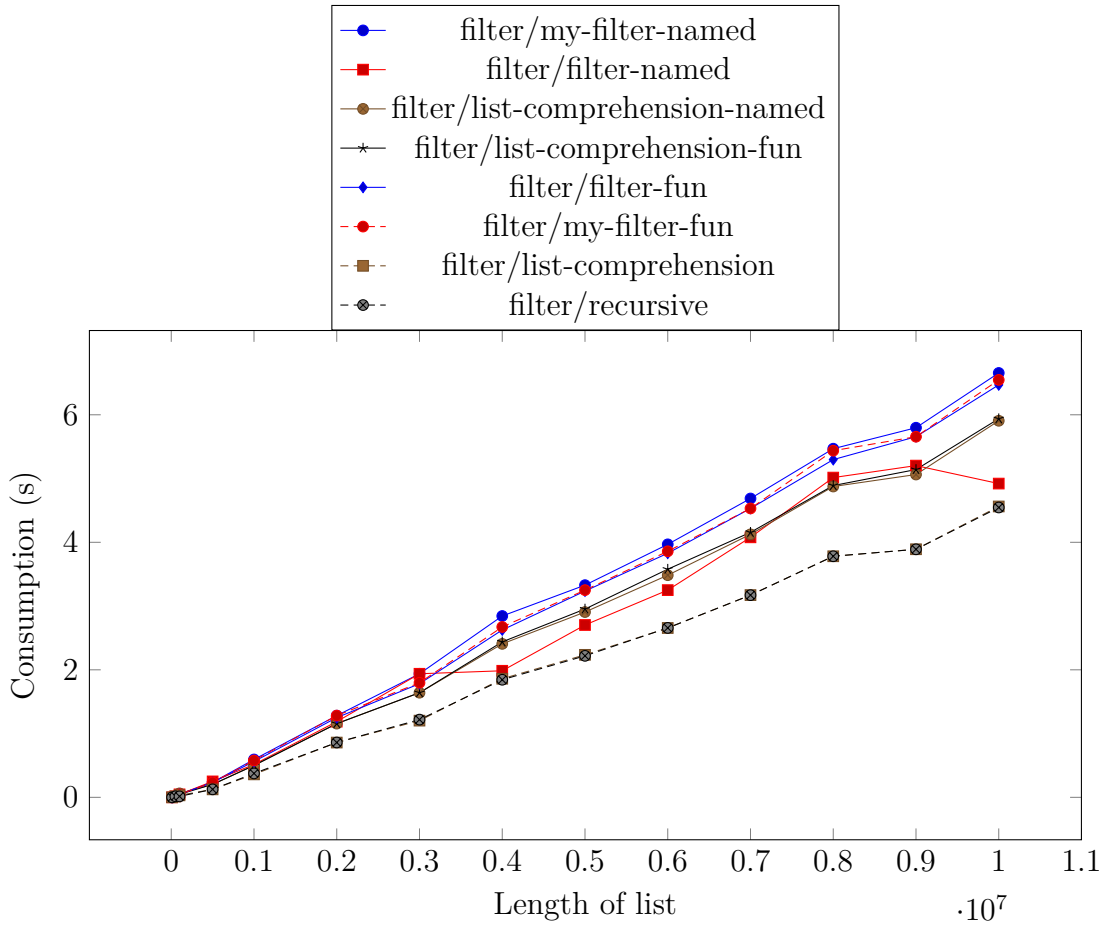


Figure 5.18: Runtime of different filter implementations and function calls

Filter map

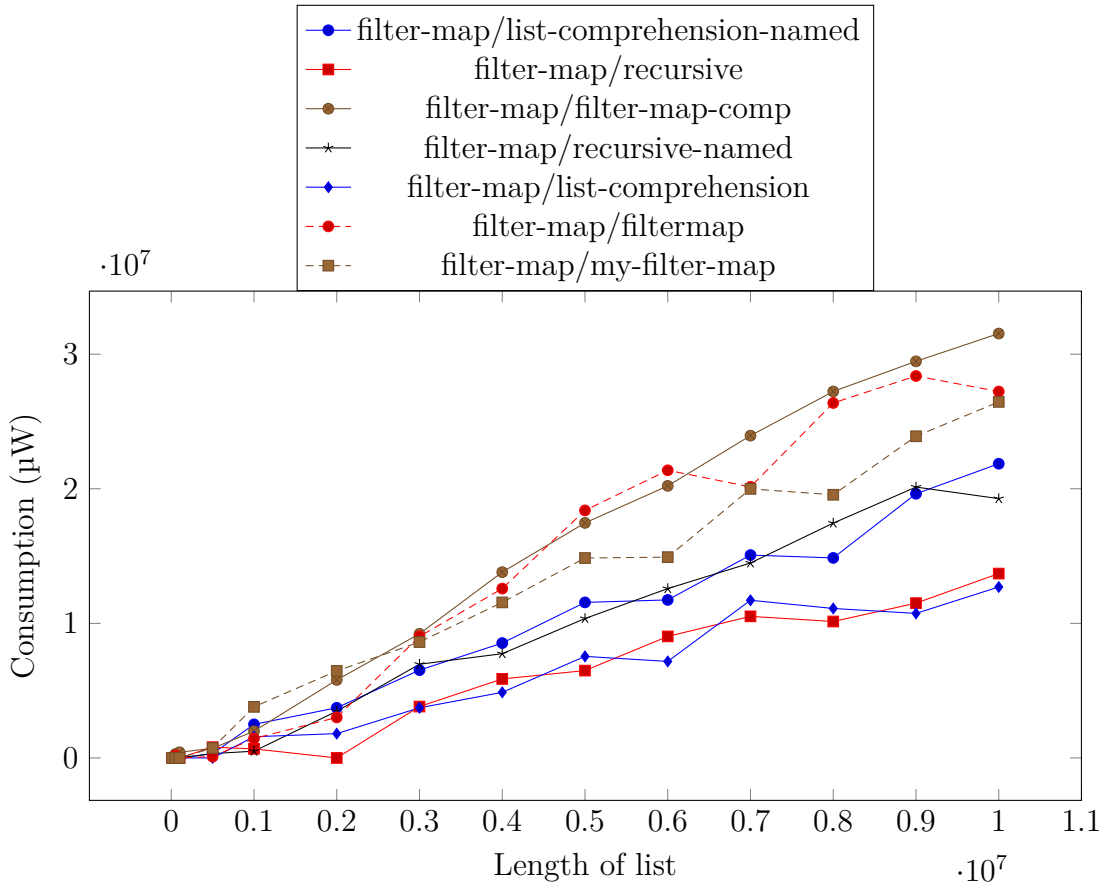


Figure 5.19: Energy consumption of different filter-map implementations and function calls

Testing higher-order function that combines map and filter. This means applying a function to the elements of a list that satisfy a condition. The results in Figure 5.19 and runtime in Figure 5.20 show that similar to filter and map, using recursion and list comprehension resulted in the most efficient implementation. Also, the composition of **lists:map/2** and **lists:filter/2** resulted in the highest consumption.

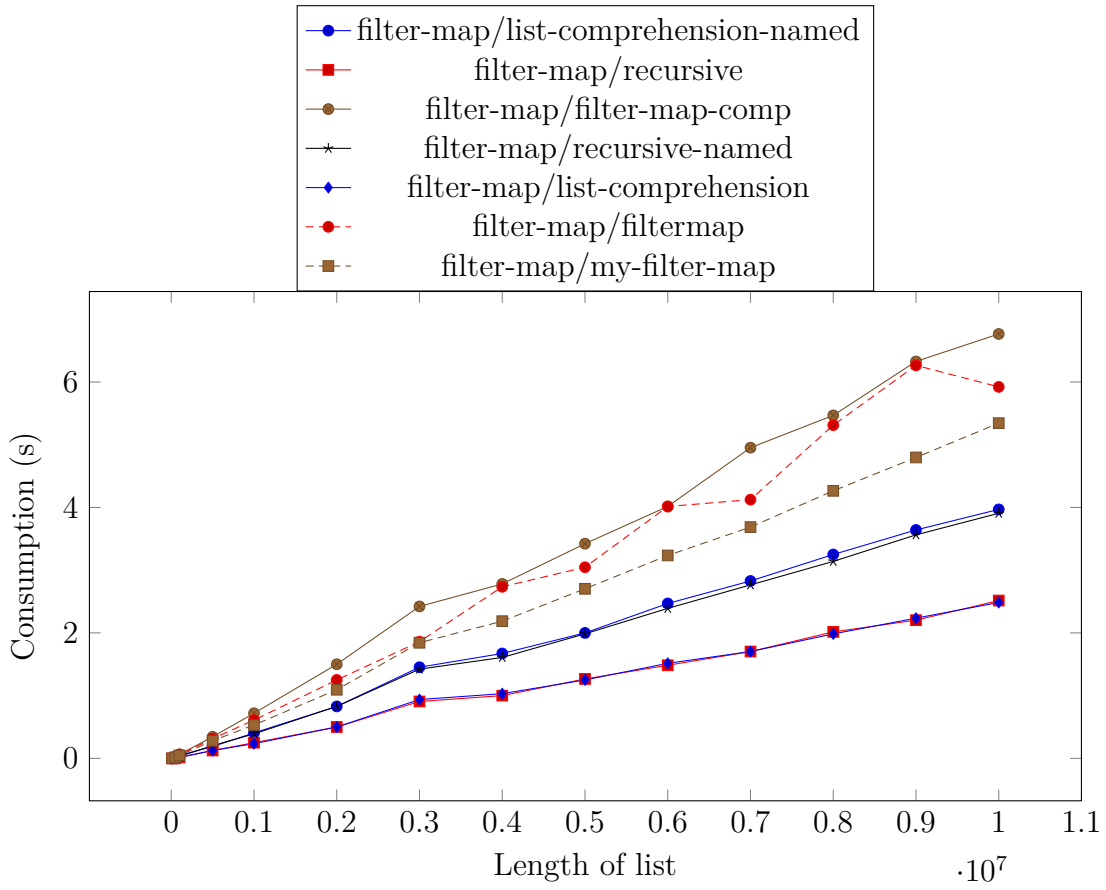


Figure 5.20: Runtime of different filter-map implementations and function calls

5.5 Parallel language constructs

Erlang supports massive concurrency with ease. It allows the creation and communication between processes with simple primitives. A process has its own isolated memory space, which holds only its own data. To share this data with another process, it must be sent as a message. The previous team [6] wanted to measure not only the energy consumption of creating, modifying and converting different data structures but also the cost of sending them between processes.

Their aim was to find out if it is worth it to transform the data to a different representation before sending it and after receiving it in order to minimize energy consumption which will be measured and compared to their results in the coming Chapter 6. However, in this section, I will measure the consumption of sending different data structures between processes in Windows.

Send

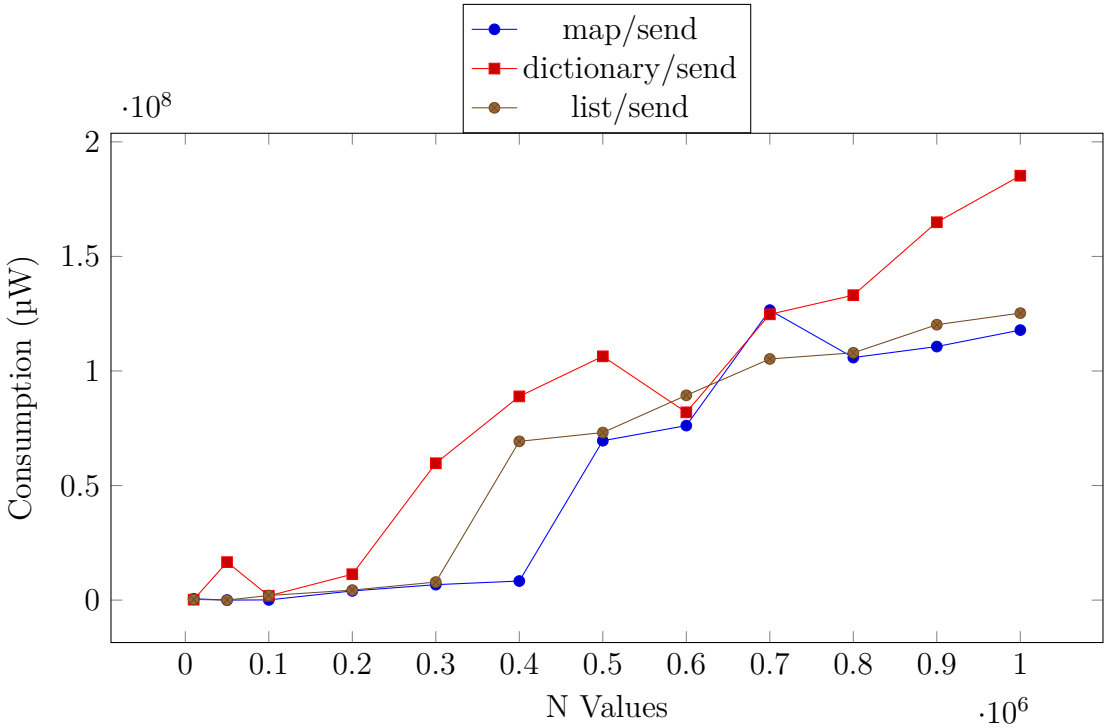


Figure 5.21: Energy consumption values for sending data structures 100 times

The **map/send** function spawns a process that communicates with the main process by exchanging a list of key-value pairs and a confirmation message. Similar to the map implementation, the **list/send** and the **dictionary/send** both communicate respectively a list and a dictionary. Based on the results shown in Figure 5.21 and runtime in Figure 5.22 we can deduce that sending a dictionary

is the worst in efficiency while sending maps is the most efficient followed by the list of tuples.

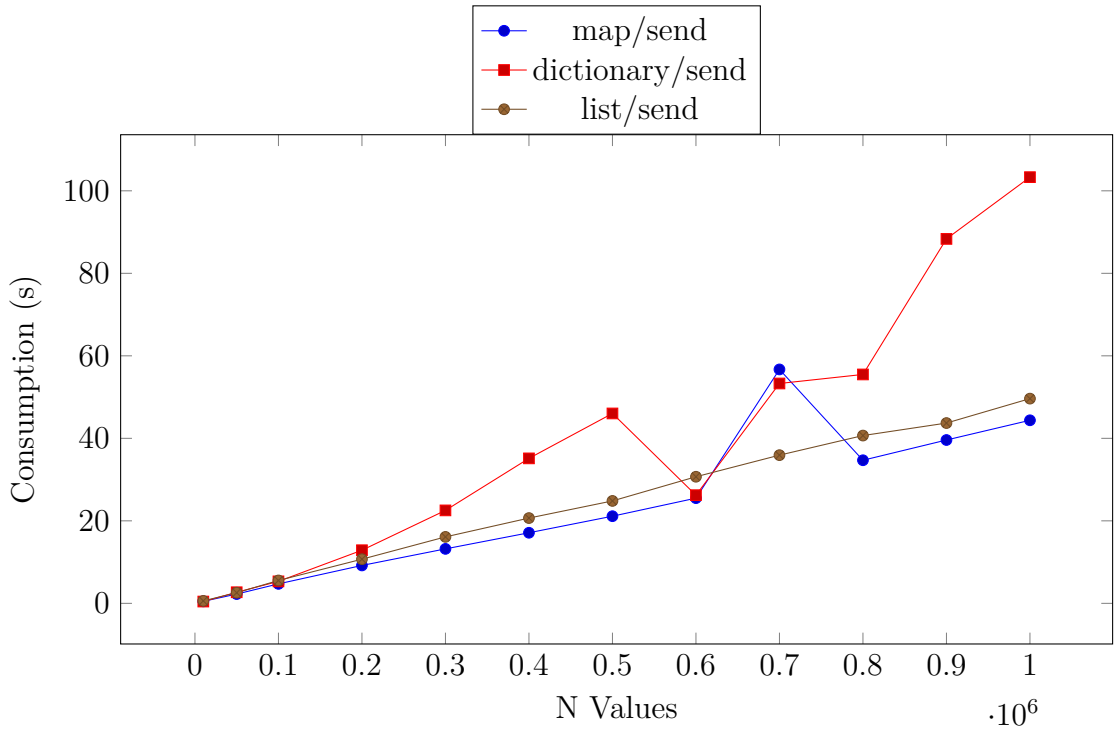


Figure 5.22: Runtime values for sending data structures 100 times

5.6 Algorithmic skeletons

This work was also driven by the research question of how different algorithmic skeletons affect energy consumption. These parallel programming patterns are widely used because they abstract away the complexity of parallel and distributed applications, so it is important to know their impact on energy consumption. In this work, I evaluated all basic skeletons such as farm and pipeline skeletons. As a further step, because these patterns can be composed, I also evaluated different combinations of skeletons.

Task farm implementations

The task farm is a popular skeleton. It parallelizes a function on an input structure. In order to have it work, we define three members with roles. The dispatcher handles the input and assigns the elements to the workers. Every worker performs the function on the input and sends the result to the collector, then asks for a new element. Finally, the collector combines the result. This skeleton does not preserve the order of the elements.

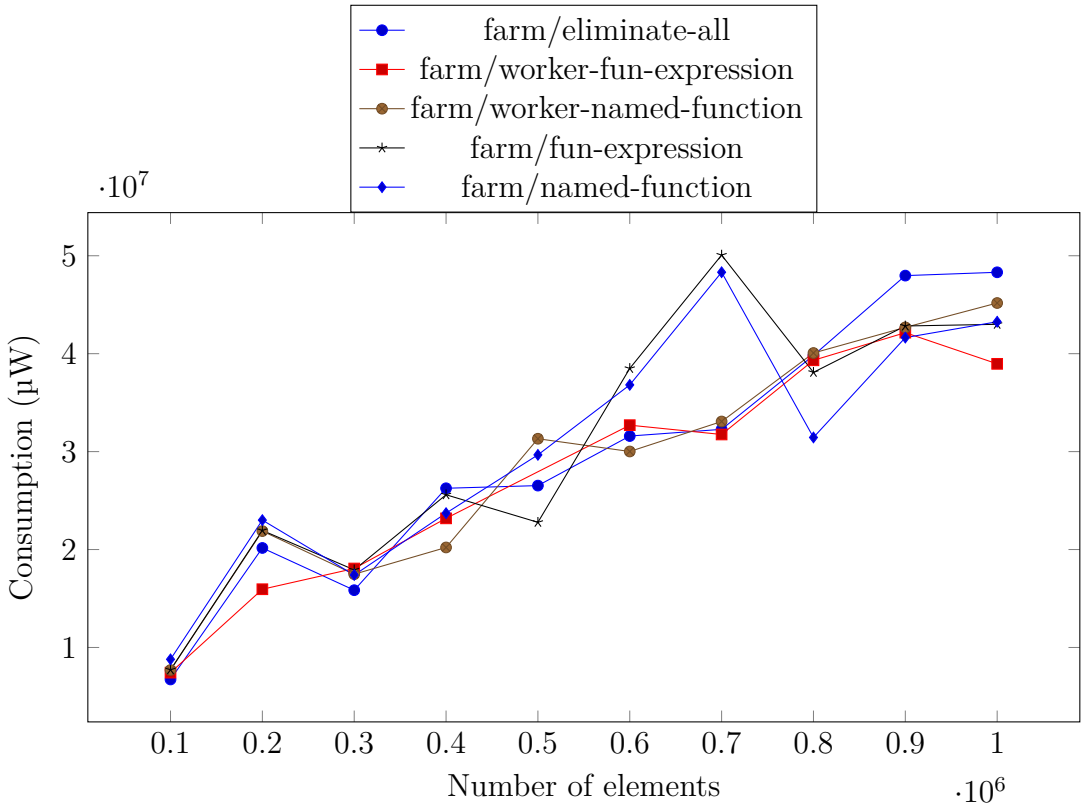


Figure 5.23: Energy consumption values for different task farm implementations

The aim is to reduce the energy consumption of basic task farm implementation, in which an implicit function is passed to the workers as an argument. Like replacing it with a fun expression then removing this argument from the workers

and using a named function and a fun expression. The results are shown in Figure 5.23 where we can see a fluctuation in the consumption but the average was approximately the same. While in Figure 5.24 shows no big difference in runtime for all implementations.

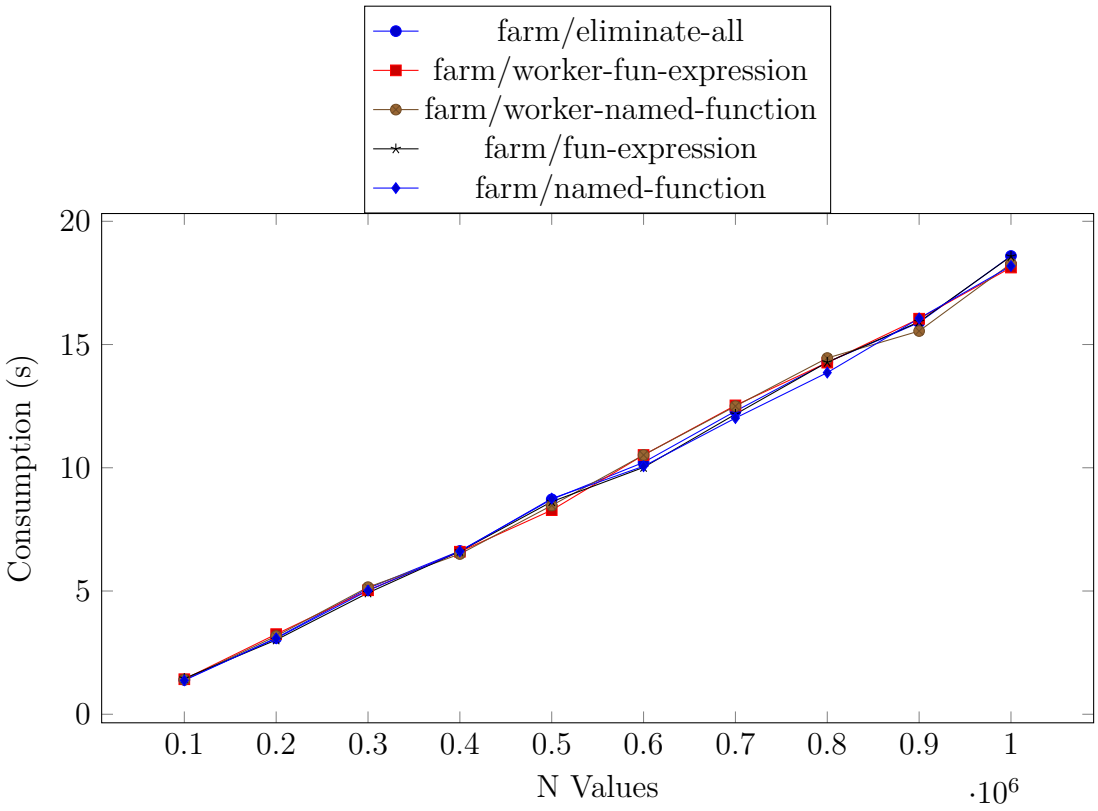


Figure 5.24: Runtime values for different task farm implementations

Identity

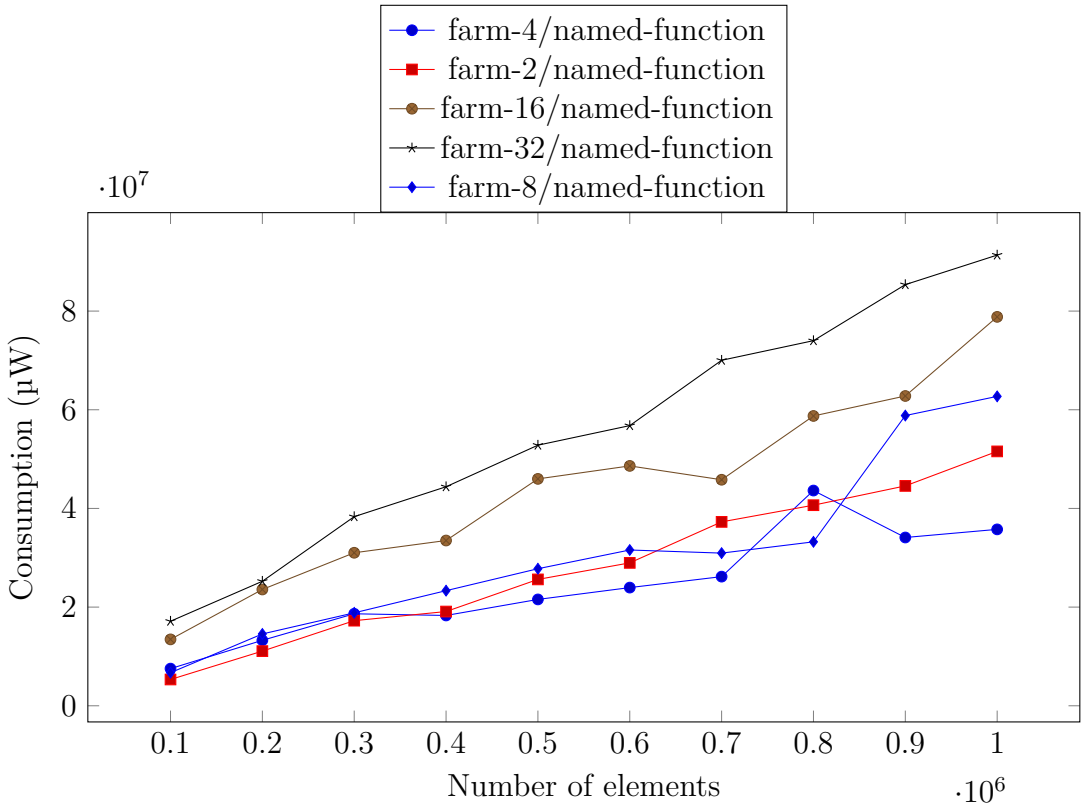


Figure 5.25: Energy consumption values for different numbers of workers

As stated earlier I have a processor with 4 physical and 8 logical cores. After testing different task farm implementations, I picked one of them and experimented with spawning 2, 4, 8, 16 and 32 workers. I used the identity function as before. The energy consumption and runtime values are displayed in Figure 5.25 and Figure 5.26 respectively. I observed that spawning 8 workers was the fastest implementation, but it used slightly more energy than spawning four workers. The computations with two and four workers were the slowest but it used the least energy. This suggests that parallel computing has a considerable effect on energy consumption, which is not always balanced by faster execution. I can see that it is

more optimal to spawn as many processes as there are physical cores rather than logical cores.

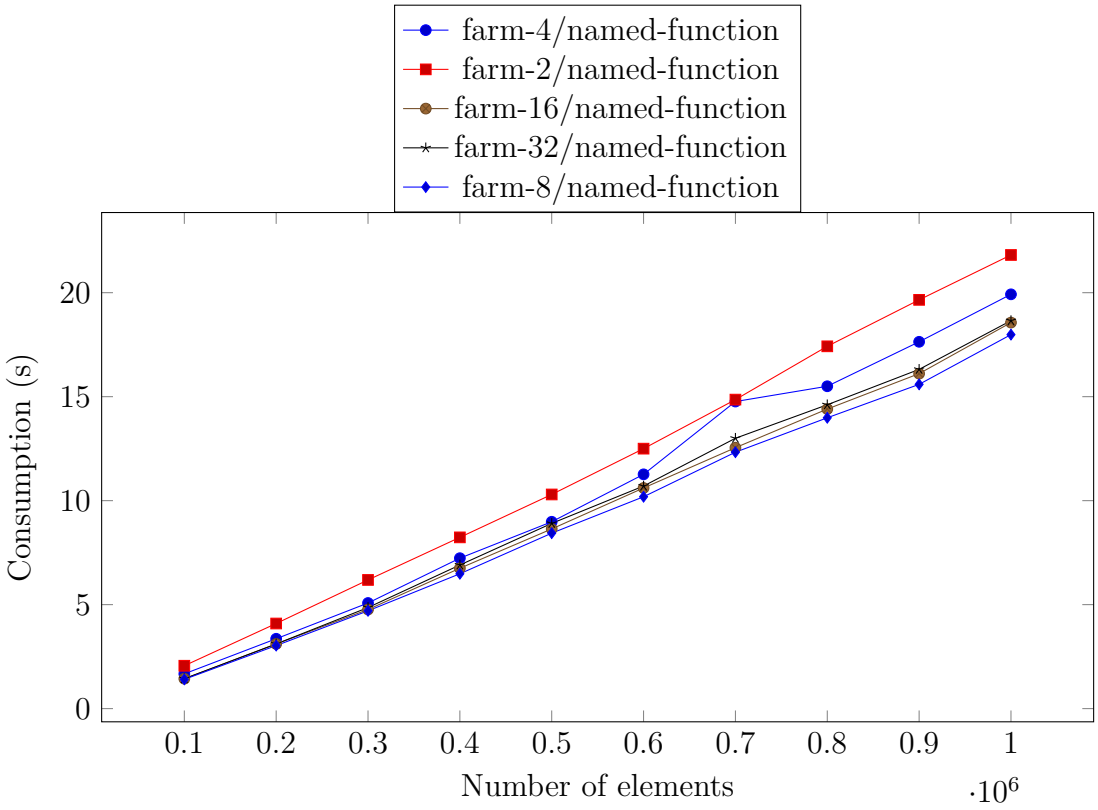


Figure 5.26: Runtime values for different numbers of workers

Fibonacci

The last measurement in this chapter is the Fibonacci implementation. The Fibonacci measurement demonstrates various approaches to parallelize the calculation of Fibonacci numbers. I measured the values for Fibonacci 1, 15 and 30 the results are shown successively. The implementation takes a list of integers as input and returns a list of their corresponding Fibonacci values as output. The functions differ in the way they create and synchronize the processes that perform the computation. Two functions, `ordparmap` and `ordparmap_hof`, preserve the order

of the input and output lists, while the other functions like `parmap`, `parmap_hof` do not. The functions also vary in the use of language features, such as recursion, higher-order functions and list comprehensions, to implement the parallelization logic. I also measured the task-farm implementation to calculate the same Fibonacci numbers and the results are plotted in the same figures. As we can see from the figures, the first insight is that the bigger the number gets the lower the differences and the gap between the implementations gets.

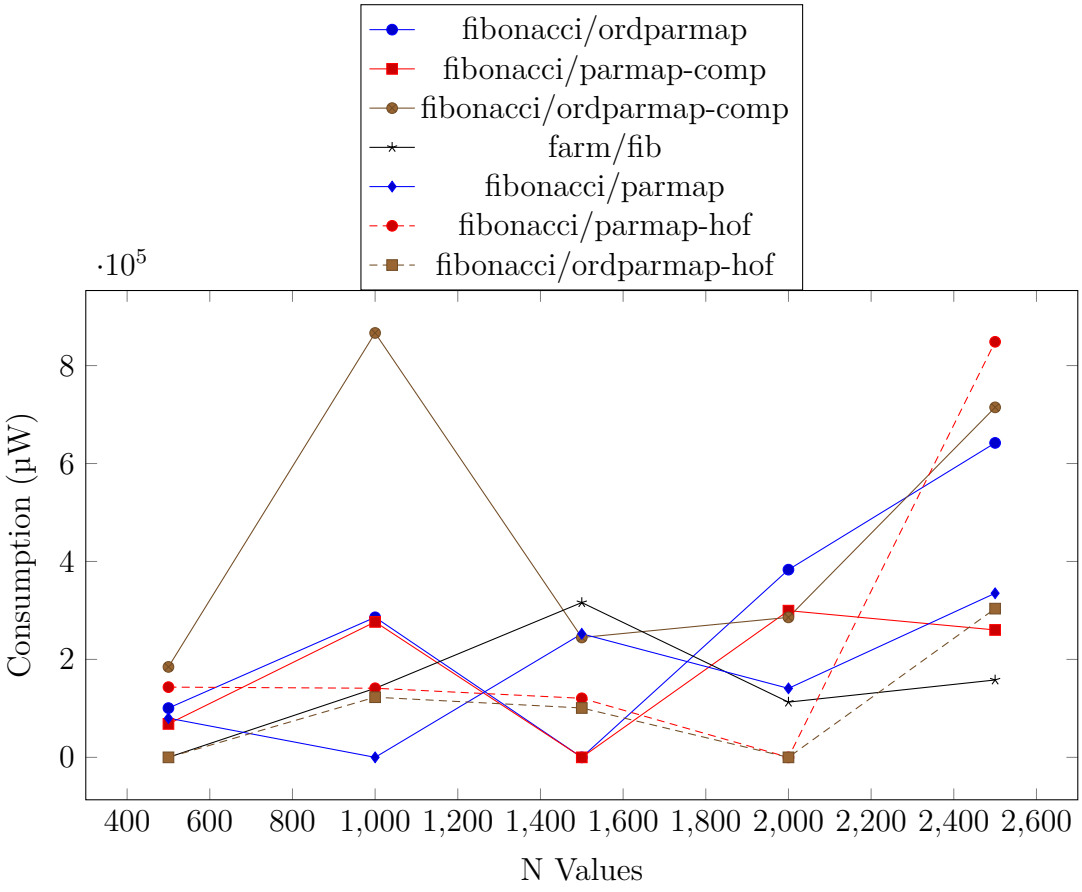


Figure 5.27: Energy consumption of calculating Fibonacci 1

For Fibonacci one, as shown in Figure 5.27 there are lots of fluctuations in runtime this can be due to parallelization and CPU scheduling. The most

consistent values are those from the task farm implementation. For the runtime, as shown in Figure 5.28 the fluctuation is still existing but it looks more linear and bounded.

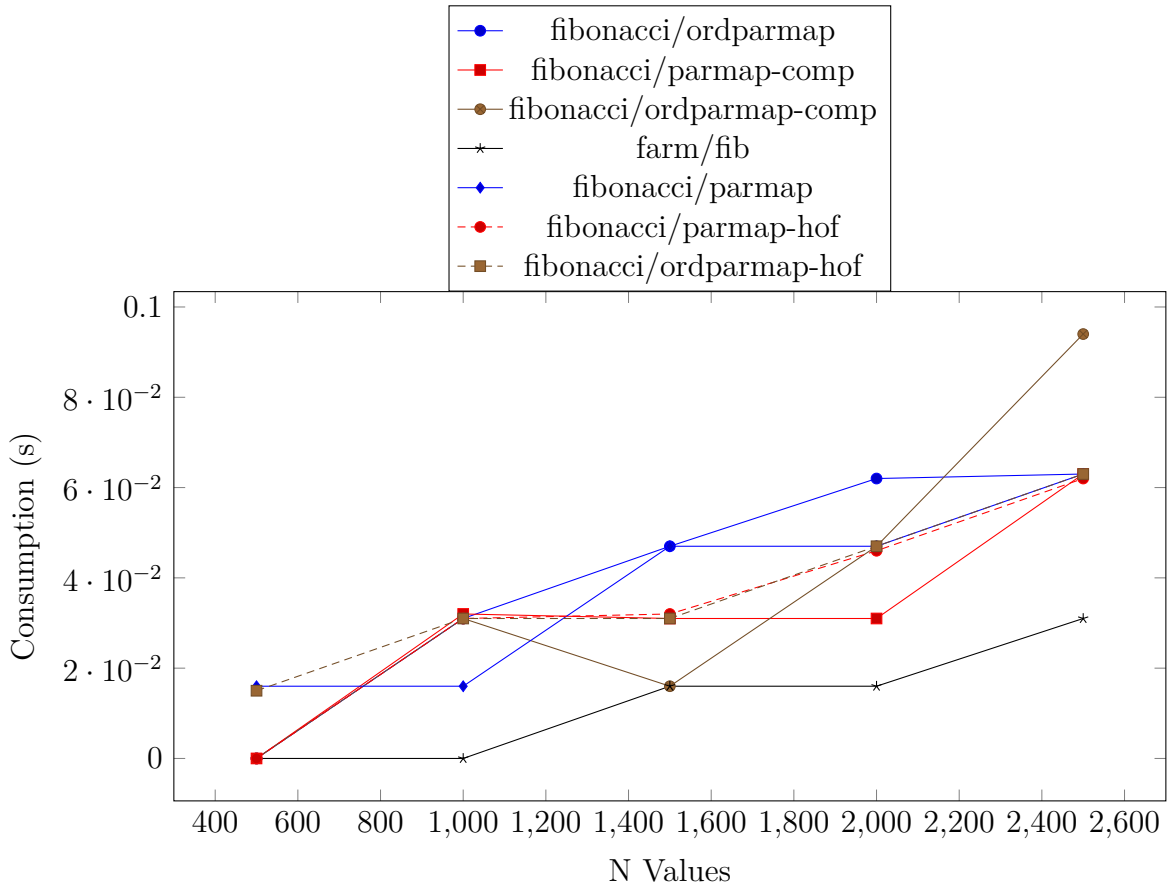


Figure 5.28: Runtime of calculating Fibonacci 1

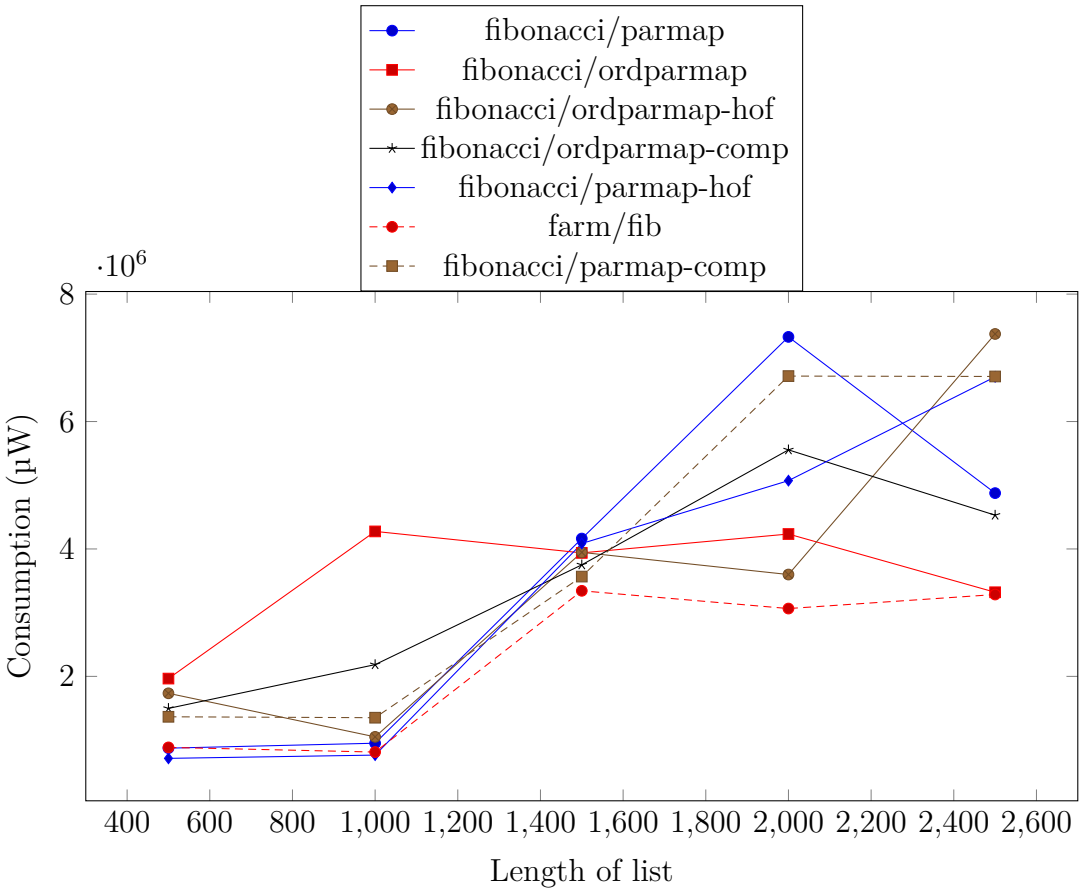


Figure 5.29: Energy consumption of calculating Fibonacci 15

As we can see in Figure 5.29, the consumption gap between the different implementations gets smaller when we increased the number to calculate from 1 to 15. However, there is still an optimal implementation for Fibonacci 15, the task farm looks the most efficient while in the runtime Figure 5.30 was almost faster than the rest of the implementations.

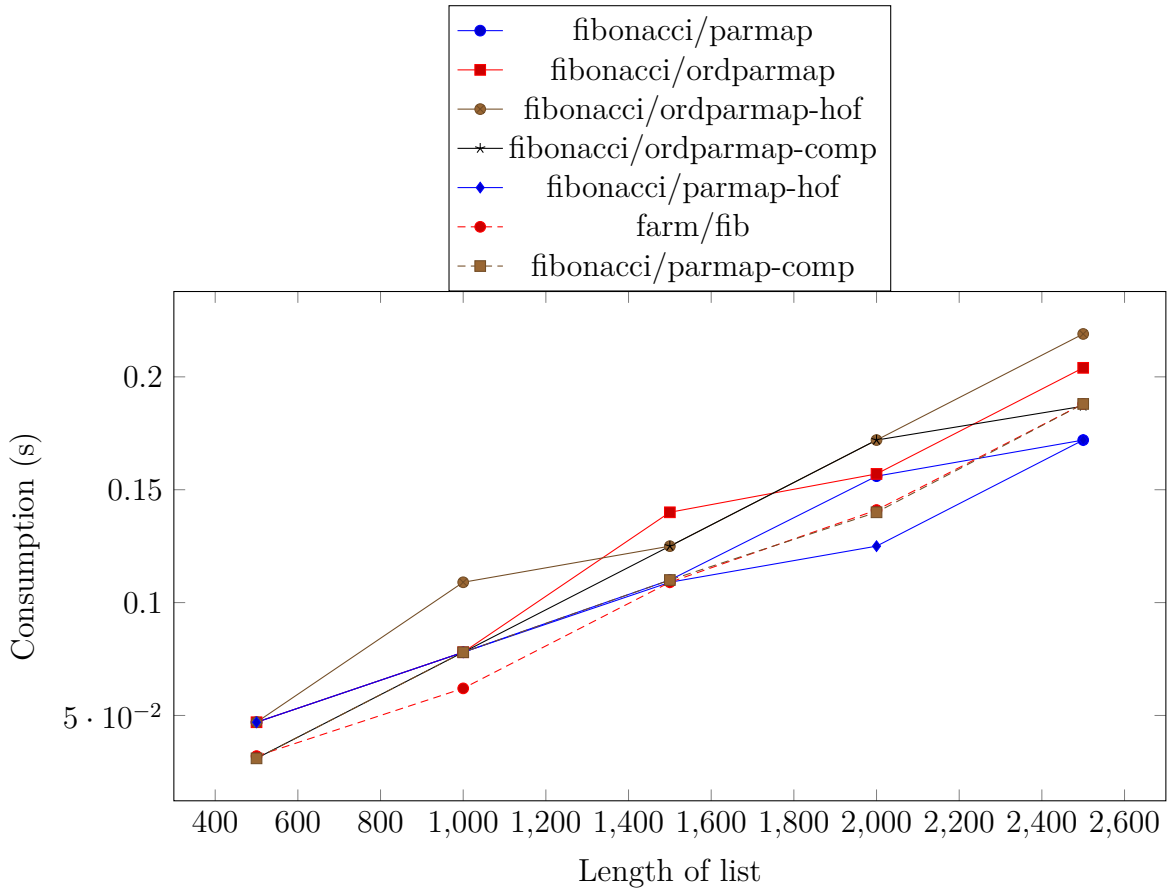


Figure 5.30: Runtime of calculating Fibonacci 15

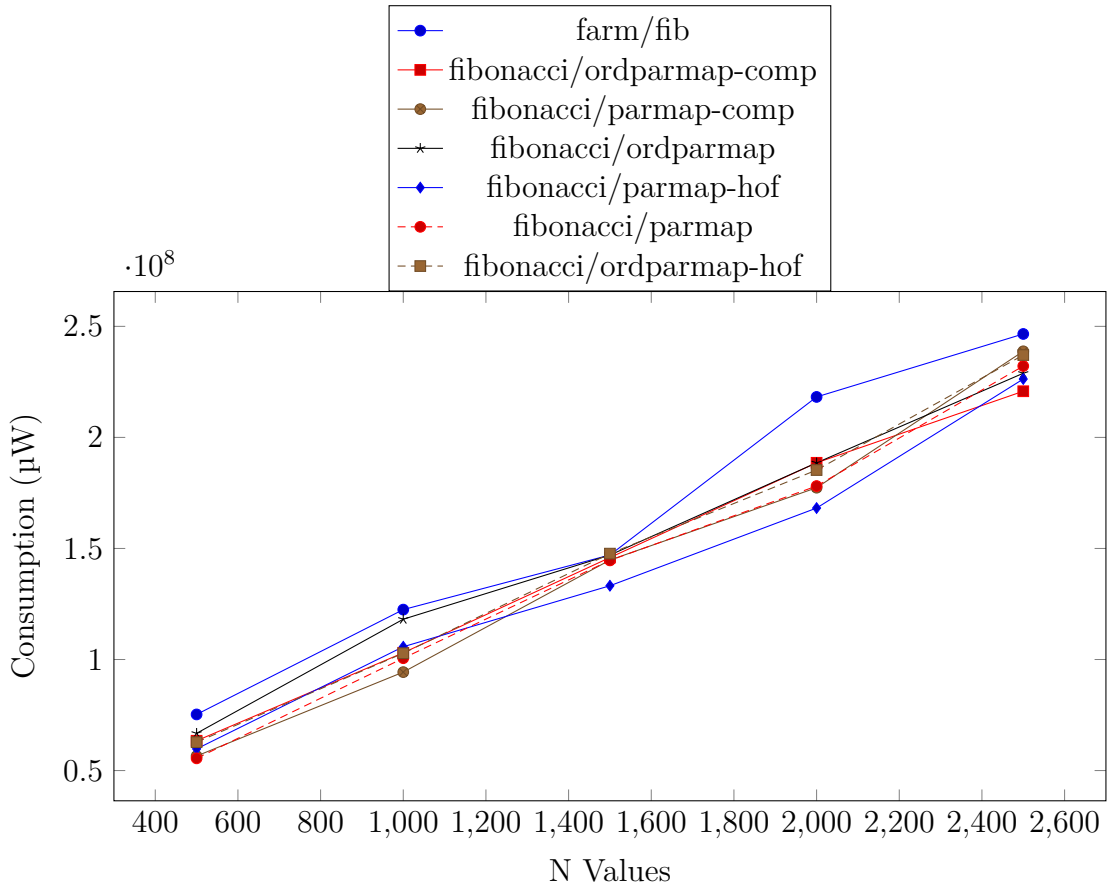


Figure 5.31: Energy consumption of calculating Fibonacci 30

In the last measurement of the Fibonacci 30, we can clearly see in Figure 5.31 how the gap is chinked to a linear result compared to the previous measurements. For instance, the runtime Figure 5.32 shows clearly that they all have the same runtime a linear constant raise in the runtime.

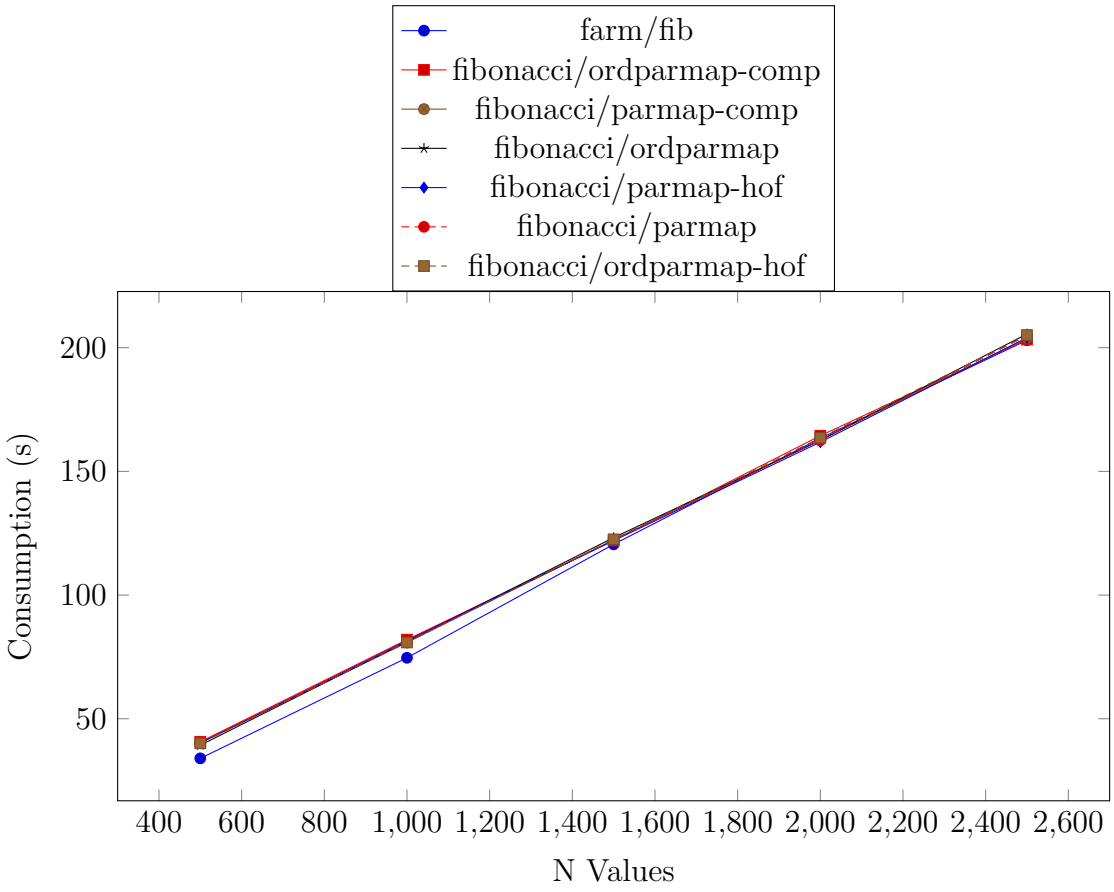


Figure 5.32: Runtime of calculating Fibonacci 30

Chapter 6

Comparing energy consumption

This research aims to compare the results and some of the behaviors of Linux and Windows operating systems. Therefore, this chapter will explore the trends between the two operating systems in more detail. The detailed methodology for the measurement and the function that I experimented on are described in Chapter 5. The results figures are presented in the appendix A.4 attached to this research, to avoid repetition and unnecessary scrolling. The appendix is structured similarly to this Chapter, with each section having subsections with Linux results followed by Windows results. The Linux energy consumption results are measured in Joules, while the Windows results are measured in Micro-watts. Due to the sampling differences between RAPL under Linux (second) [47, 16] and Scaphandre (nanosecond) sampling [41] rate that I chose in the framework presented earlier in Chapter 4, I did not convert the units to Watts, which are equivalent to Joules [48].

6.1 Data structures

Build all

As we can see from the figures, there was a noticeable difference between Linux and Windows while building a dictionary, Linux has a peak in energy consumption compared to other data structures while Windows had a much lower peak. It is interesting to see the rest of the functions had the same behavior, the **map-pattern** and **map-put** had a close consumption both on Windows as long as Linux. Similarly, **list-recursive** was the least consuming data structure to build for both operating systems. However, the runtime was approximately the same for both of the operating systems.

Finding the value for a given key

For the energy cost of finding an element in a list, we can see from both graphs that the trends are similar for Linux as long as for Windows. However, searching for an element in a map and dictionary is slightly fluctuating in the dictionary measurement for Windows but kept the overall shape that the dictionary is always higher than a map both in consumption as long as runtime for Linux and Windows.

Updating a single element

The energy cost of updating an element in a list shows that both systems handle it approximately in the same way. There was no huge difference even with the **maps-update** that is shown for reference.

Updating all elements

Interestingly, the energy cost of updating all elements in a map is always higher than updating all elements in a list for both systems with recursive being

clearly the highest in Windows while recursive and fold being alternating and highest in Linux. One important notice is that **lists-foldr** had the exact same jump in consumption at input 8 000 000 for both Windows and Linux. While the energy cost of updating all elements in a dictionary on Linux was the highest consistently, on Windows it was fluctuating too much however as an average value it is still high compared to updating all elements in a list.

Deleting

The energy cost of deleting an element from a list compared to a map looked similar in general, with a slightly higher consumption on **list-delete/keytake** on Linux. On Windows, however, it was alternating with **list-delete/recursive**.
A surprising finding is that deleting an element in a dictionary was the complete opposite result between both operating systems. Linux was higher with Windows being super efficient. This can be related to the dictionary implementation in `C`, which can behave differently on the operating systems. This finding needs more investigation and attention in future work to understand the reason behind this huge difference.
Following the dictionary behavior, deleting an element from a map was also different between both operating systems with Linux having a sudden rise in consumption from the beginning, Windows on the other hand having a drop followed by some fluctuations.

6.2 Higher-order functions

Map

The energy cost and runtime of different map implementations and function calls were slightly different, we can see in the runtime for Windows that there was a drop at input 8 000 000 but it went back to higher results after. Regarding

energy consumption, we can say that the **recursive** and **list-comprehension** are the lower for both operating systems. For energy consumption using list comprehensions with fun expressions compared to inlining the function definition the **original-comprehension** was the least on both systems but with some fluctuations on Windows. Windows also had a similar consumption for the named and lambda expression implementations while Linux has a noticeable gap between the latter functions.

Filter

The energy consumption for filter implementations and function calls on both operating systems had a similar behavior, we can see that the **recursive** and the **list-comprehension** were the most optimal implementations on Linux as well as Windows.

Filter-map

Similar to the filter results, the combination of filter and map implementation had no huge difference in the overall shape and trend of the functions. We can see that again, **recursive** and **list-comprehension** remain the most optimal implementation among the rest.

6.3 Parallel language constructs

Sending raw data structures

Sending raw data structures once looks completely different on Linux compared to Windows. On Windows, we can see that it has more stable fluctuations while Linux had a constant increase in consumption. Whereas, sending the data

structures 100 times gave more similar better-shaped figures on both operating systems. From a runtime perspective, both operating systems behaved the same.

Sending data in pieces

Sending data structures in pieces was similar in two out of four different implementations. Sending a list in binary was the least in both systems while sending a list from binary was the highest in Linux, it was remarkably high on Windows too.

6.4 Algorithmic skeletons

Measuring different task farm implementations

We can see that Linux had a more linear result on all implementations, whereas the Windows figure shows some high fluctuations but kept the same increasing rate.

Spawning different number of workers

On Linux, the energy consumption was lower spawning 8 processes, they suggested spawning the same number as the logical processes your CPU has. On Windows, we found that spawning the number of processes as the number of physical processes on our CPU which was 4 physical. Yet, it was also efficient to spawn up to 8 processes on Windows. Spawning a higher number of processes like 16 or 32 used the highest energy but run the fastest on both systems.

Measuring a more complex computation

Looking at all the results for both operating systems, the first thing to deduce is the higher the Fibonacci number gets the lower the difference between the implementations gets and the more similar both operating systems behave. For instance, Fibonacci 1 was fluctuating too much on Windows and we had no big difference in the Linux consumption. With Fibonacci 15, we can see that the task farm implementation was the most efficient compared to the parallel implementation on Linux being the efficient one. Fibonacci 30 on Linux had no difference in energy consumption, while we can tell that the parallel map implementation is the most efficient with a negligible difference between all implementations.

Chapter 7

Related work

In this chapter, we explore the existing literature and research related to green computing and energy consumption in different computing environments. Specifically, we focus on comparing the energy consumption of Windows and Linux operating systems. Additionally, we extend the comparison to programming languages such as Java, Python, C, and Haskell, examining their energy efficiency. In general, this chapter provides an overview of the previous studies conducted in these areas, establishing the foundation for our research, and presenting the findings.

7.1 Windows versus Linux operating systems

To fairly compare and study the energy consumption differences between Linux and Windows, we first need to understand the fundamental key differences between these operating systems. A detailed study was conducted by Hadeel et al [49] where they reviewed the history and development of both systems and their market share and user base. They also discussed the technical aspects of the systems, such as the kernel, the file system, the security model, the user interface

and the compatibility with hardware and software. One of the major differences is the process scheduling, where they compared the scheduling algorithms used by Linux kernel 2.6 and Windows NT-based versions, and analyzed their trade-offs between fairness and responsiveness. Another dissimilarity is in memory management, where they evaluated different paging strategies for Linux and Windows and examined the impact of swap partition and pagefile on disk fragmentation and system performance.

Another study by Beatriz Prieto et al [50] on the energy efficiency of personal computers compared to mainframe computers and supercomputers, where the authors explored the possibility of running scientific and engineering programs on personal computers and measuring these systems' power efficiency. They also showed how the power efficiency obtained for the same workloads on personal computers is similar and sometimes less than that obtained on supercomputers included in the Green500 [51] ranking. This study reveals that energy consumption not only can vary between operating systems but also when using different makers where they used five different personal computers with different processors of different generations.

A comparative analysis was performed by Sayed Najmuddin et al [52] of the power and battery consumption of Windows operating systems and modern Linux distributions, such as Ubuntu, Fedora, Debian, Red Hat, and others. They argued that one of the main factors that affect the power management of Linux is the quality of the drivers, which are often poorly written or incompatible with the hardware. The researchers also explained that this is due to the reluctance of hardware manufacturers to share the details of their products with driver developers, especially those who work for open-source operating systems. They also pointed out that some versions of the Linux kernel are not properly designed and optimized for mobile systems, as they are mainly intended for desktop platforms. They concluded by recommending that users should select the most suitable and efficient kernel for their system as a way to improve the power and battery performance of Linux.

All of the previously mentioned research proves the fact that Windows and Linux differ radically from each other.

7.2 Programming languages efficiency

As stated earlier, the second focus of this research was on how efficient programming languages are, and how they differ from each other.

To answer these questions I started by analyzing the findings of the paper written by Rui Pereira et al called "Energy Efficiency across Programming Languages" [53] where they presented a study of the energy efficiency of 27 programming languages, using 10 different algorithms. The authors measured the time, CPU usage, memory usage and energy consumption of each program execution, and used statistical methods to rank the languages according to each objective. They also analyzed the impact of the execution type (interpreted or compiled) and the programming paradigm (imperative, object-oriented or functional) on energy efficiency.

The main findings are that compiled languages are more energy efficient than interpreted ones and that functional languages are more energy efficient than imperative or object-oriented ones. The researchers also identified C, Rust, Ada and Pascal as the most energy-efficient languages, and Perl, JRuby, Python and Lua as the least energy-efficient ones. They also found interesting correlations between energy, time and memory, such as slower languages consuming less energy and memory usage influencing energy consumption such as languages with comparable energy consumption can have significantly different execution times. For instance, in the binary trees benchmark, Pascal consumed approximately 10% less energy than Chapel, but Chapel executed about 55% faster than Pascal. This finding emphasizes the results that I found in Chapter 5 where the parallel implementation of the identity function with two workers was the slowest but with the least energy consumption, which was the same finding for the previous thesis work [6].

The authors used a tool called CodeCarbonFootprint (CCF) to measure the energy consumption of each language and problem. The experiments were performed on a desktop computer with the following configuration: 16GB of RAM, an Intel® Core™ i5-4460 CPU @ 3.20GHz with a Haswell microarchitecture, and a Linux Ubuntu Server 16.10 operating system.

Another important aspect of programming languages and their energy consumption can be linked to the misuse of data structures or engineering solutions. This can be illustrated in the research done by Meszaros et al [42] where they measured the energy consumption of Erlang programs and discovered patterns and relations between language constructs and power consumption. They created a framework to make the measurement user-friendly. This framework uses RAPL to read the access the energy consumption values. All measurements were done using Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz and 8 GB of DDR3 RAM @ 1600MHz, using Ubuntu 16.04 LTS. They found that eliminating higher-order functions may result in a more efficient program. They also found that using naive parallelizing solutions can result in the worst consumption due to the fact of spawning different processes which was the case in our results mentioned in Chapter 5.

Chapter 8

Conclusion

The purpose of this Chapter is to provide an overview of the main outcomes and results of this thesis and to identify some avenues for future work.

Conclusion

Green computing is important for all kinds of systems, from handheld devices to large-scale data centers. It can help reduce greenhouse gas emissions [5, 1], energy consumption, and electronic waste. It can also save costs and improve performance for technology makers and users. This thesis was a complementary work to the previous findings [6, 54] achieved in recent years. Previously, the work was only focused on Unix-like operating systems, thus this thesis was a first step into breaking the gap between these different platforms.

As a first step, I started by looking for a tool (Chapter 3) to fill the gap between the two operating systems since RAPL [16] is not available in Windows [38]. I ended up using Scaphandre [24] since it checked all of the boxes (Chapter 4) for our constraints. I integrated the tool with the previous GreenErl framework [6] and measured different Erlang functions.

There were some nice findings about Erlang on Windows. I measured the energy consumption and runtime (Chapter 5) of some data structure implementations such as lists, maps and dictionaries. I evaluated creating, converting, updating, searching and deleting elements from those data structures where I found that in most cases list is the worst in terms of energy consumption, the dictionary was better except while updating all of its elements. I also measured some higher-order functions like applying a function on filtered elements where recursive implementations were the most efficient. since one of the many strengths of Erlang is parallelization, I also measured sending data in different forms. Lastly, I assessed some algorithmic skeletons such as task farms where I concluded it is better to spawn as many workers as the number of your physical processes.

As for the comparison section (Chapter 6), I can say the trend in both operating systems had more similarities than differences with some exceptions. For example, the dictionary had a lower consumption on Windows and a noticeably different overall behavior while being the highest consumer in Linux with the exception of updating all elements in the dictionary that resulted in it being as high in Windows as in Linux. We also noticed that Windows results had higher fluctuating values compared to Linux, this can be due to the difference in the sampling rates between RAPL [47, 16] and Scaphandre [41].

Future work

As Scaphandre [8, 24] is still in an early stage, I think it is a good approach to focus on this tool to create a cross-platform GreenErl for (Linux, Windows, VMs, and Containers). Using the same tool to get the values will give genuine comparable results. In the meantime, finding a relation between RAPL [47, 16] sampling and Scaphandre [41] in order to get a closer value in Watts or Joules so we can tell which one is more efficient.

Bibliography

[1] Andrew A Chien. Computing's grand challenge for sustainability, 10 2022. URL https://cacm.acm.org/magazines/2022/10/264841-computings-grand-challenge-for-sustainability/fulltext.

[2] Simon Unge. How cisco is using erlang for intent-based networking, 06 2020. URL https://codesync.global/media/how-cisco-is-using-erlang-for-intent-based-networking-cbf20/.

[3] Around the. Datareportal – global digital insights, 2013. URL https://datareportal.com/global-digital-overview#:~:text=Internet%20use%20around%20the%20world,million%20new%20users%20every%20day.

[4] IBM . What is green computing?, 04 2022. URL https://www.ibm.com/cloud/blog/green-computing.

[5] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. Deep learning's diminishing returns, 09 2021. URL https://spectrum.ieee.org/deep-learning-computational-cost.

[6] Gergely Nagy Áron Attila Mészáros. Greenerl measuring the energy consumption of erlang programs and energy conscious refactorings. Master's thesis, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, 2019.

[7] Hubblo. Scaphandre documentation, 2023. URL https://hubblo-org.github.io/scaphandre-documentation/index.html.

[8] hubblo org. hubblo-org/scaphandre: Energy consumption metrology agent. let "scaph" dive and bring back the metrics that will help you make your systems and applications more sustainable !, 03 2023. URL https://github.com/hubblo-org/scaphandre.

[9] Ishita Ray. Green computing, 02 2012. URL https://www.researchgate.net/publication/270570843_Green_Computing.

[10] B Saha. Green computing, 2014. URL https://www.semanticscholar.org/paper/Green-Computing-Saha/78ab8107b069c3f5503cacf616d8575aed628297.

[11] Jordi Torres, David Carrera, Kevin Hogan, Ricard Gavalda, Vicenc Beltran, and Nicolas Poggi. Reducing wasted resources to help achieve green data centers. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 04 2008. doi: 10.1109/ipdps.2008.4536219. URL https://ieeexplore.ieee.org/abstract/document/4536219.

[12] Sanjay Podder, Adam Burden, Shalabh Kumar Singh, and Regina Maruca. How green is your software?, 09 2020. URL https://hbr.org/2020/09/how-green-is-your-software.

[13] About - erlang/otp, 2023. URL https://www.erlang.org/about.

[14] Erlang. Erlang, 2023. URL https://www.erlang.org/.

[15] Fred Hebert. Introduction | learn you some erlang for great good!, 2013. URL https://learnyousomeerlang.com/introduction.

[16] Rapl - powerapi, 2023. URL https://powerapi.org/reference/formulas/rapl/.

[17] tools/power/rapl — firefox source docs documentation, 2023. URL https://firefox-source-docs.mozilla.org/performance/tools_power_rapl.html#:~:text=Unfortunately%2C%20rapl%20does%20not%20work,the%20relevant%20model%2Dspecific%20registers.

[18] Eötvös Loránd University. Refactorerl - refactoring erlang programs, 2013. URL https://plc.inf.elte.hu/erlang/refactorerl-features.html.

[19] Eötvös Loránd University. Refactorerl - refactoring erlang programs, 2015. URL https://plc.inf.elte.hu/erlang/.

[20] Intel® power gadget, 2014. URL https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html.

[21] windows-driver content. Powercfg command-line options, 12 2021. URL https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/powercfg-command-line-options.

[22] Cpu-z | softwares | cpuid, 04 2023. URL https://www.cpuid.com/softwares/cpu-z.html.

[23] Cpu-z txt, 2023. URL https://www.scribd.com/document/414756413/CPU-Z-TXT#.

[24] scaphandre - rust, 2023. URL https://docs.rs/scaphandre/latest/scaphandre/.

[25] hubblo org. hubblo-org/windows-rapl-driver: Windows driver to get rapl (cpu+ram energy consumption) metrics from a bare metal computer., 04 2023. URL https://github.com/hubblo-org/windows-rapl-driver.

[26] Reading rapl energy measurements from linux, 2023. URL https://web.eece.maine.edu/~vweaver/projects/rapl/.

[27] How scaphandre computes per process power consumption - scaphandre documentation, 2023. URL https://hubblo-org.github.io/scaphandre-documentation/explanations/how-scaph-computes-per-process-power-consumption.html.

[28] Element, 2023. URL https://app.gitter.im/#/room/#hubblo-org_scaphandre:gitter.im.

[29] hubblo org. Scaphandre windows installation · issue 247 · hubblo-org/scaphandre, 12 2022. URL https://github.com/hubblo-org/scaphandre/issues/247.

[30] hubblo org. Release scaphandre v0.5.0 · hubblo-org/scaphandre, 01 2023. URL https://github.com/hubblo-org/scaphandre/releases/tag/v0.5.0.

[31] Compilation for windows - scaphandre documentation, 2023. URL https://hubblo-org.github.io/scaphandre-documentation/tutorials/compilation-windows.html.

[32] gewarren. Tutorial: Create a windows service app - .net framework, 09 2022. URL https://learn.microsoft.com/en-us/dotnet/framework/windows-services/walkthrough-creating-a-windows-service-application-in-the-component-designer.

[33] gewarren. Introduction to windows service applications - .net framework, 09 2021. URL https://learn.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications.

[34] Learn rust, 2023. URL https://www.rust-lang.org/learn.

[35] The rust community's crate registry, 2023. URL https://crates.io/.

[36] Install rust, 2013. URL https://www.rust-lang.org/tools/install.

[37] TylerMSFT. Install c and c++ support in visual studio, 12 2021. URL https://learn.microsoft.com/en-us/cpp/build/vscpp-step-0-installation?view=msvc-170.

[38] Compatibility - scaphandre documentation, 2020. URL https://hubblo-org.github.io/scaphandre-documentation/compatibility.html.

[39] Hubblo, 04 2023. URL https://github.com/hubblo-org.

[40] stevewhims. Wmi command-line (wmic) utility - win32 apps, 03 2023. URL https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmic.

[41] Json exporter - scaphandre documentation, 2023. URL https://hubblo-org.github.io/scaphandre-documentation/references/exporter-json.html.

[42] A.A. Mezsaros, G. Nagy, I. Bozo, and M. Toth. Towards green computing in erlang. *Studia Universitatis Babeș-Bolyai Informatica*, 63:64–79, 06 2018. doi: 10.24193/subbi.2018.1.05. URL https://www.readcube.com/articles/10.24193%2Fsubbi.2018.1.05.

[43] Erlang – maps, 2023. URL https://www.erlang.org/doc/man/maps.html#update-3.

[44] Erlang – maps, 2023. URL https://www.erlang.org/doc/man/maps.html#update_with-3.

[45] Erlang – proplists, 2023. URL https://www.erlang.org/doc/man/proplists.html#get_value-2.

[46] Erlang – lists, 2023. URL https://www.erlang.org/doc/man/lists.html#keyfind-3.

[47] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements, 01 2018. URL https://www.researchgate.net/publication/322308215_RAPL_in_Action_Experiences_in_Using_RAPL_for_Power_Measurements.

[48] Electricity and energy terms in lighting (j, kw, kwh, lm/w), 2015. URL https://www.stouchlighting.com/blog/electricity-and-energy-terms-in-led-lighting-j-kw-kwh-lm/w#:~:text=Watts%20are%20defined%20as%201,per%20second%20that%20it's%20running.

[49] Hadeel Tariq Al-Rayes. Studying main differences between linux, 09 2012. URL https://www.researchgate.net/publication/328125275_Studying_Main_Differences_Between_Linux.

[50] Beatriz Prieto, Juan José Escobar, Juan Carlos Gómez-López, Antonio F. Díaz, and Thomas Lampert. Energy efficiency of personal computers: A comparative analysis. *Sustainability*, 14:12829, 01 2022. doi: 10.3390/su141912829. URL https://www.mdpi.com/2071-1050/14/19/12829.

[51] Home - | top500, 2013. URL https://www.top500.org/.

[52] Sayed Najmuddin, Zabihullah Atal, and Riaz Ahmad Ziar. Comparative analysis of power consumption of the linux and its distribution operating systems vs windows..., 12 2021. URL https://www.researchgate.net/publication/361225454_Comparative_Analysis_of_Power_Consumption_of_the_Linux_and_its_Distribution_Operating_Systems_vs_Windows_and_Mac_Operating_Systems.

[53] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE 2017*, 2017. doi: 10.1145/3136014.3136031. URL https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf.

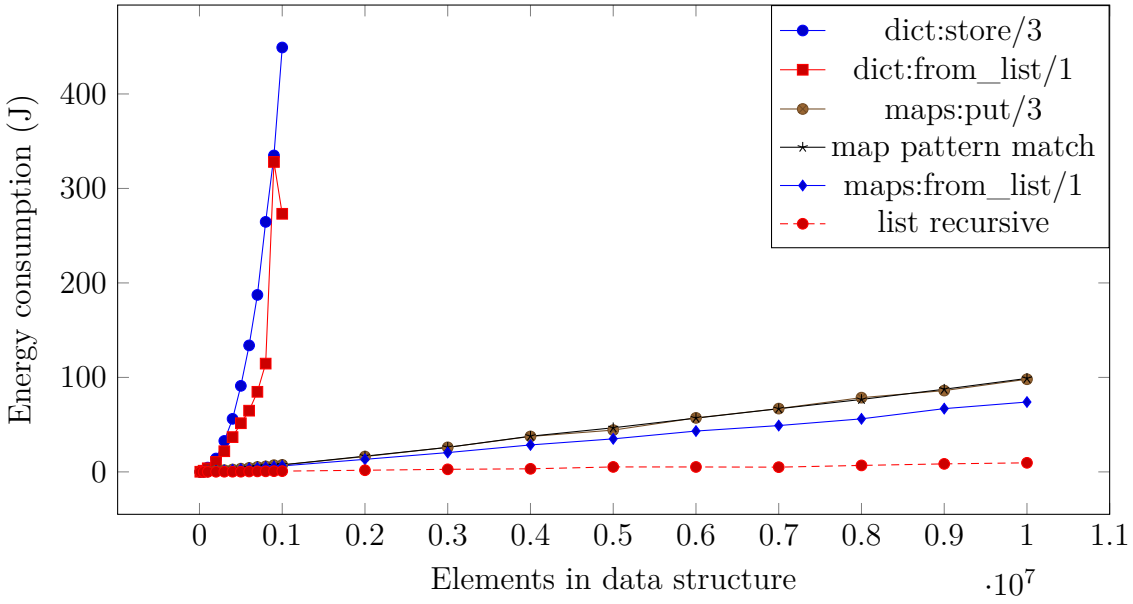
[54] Jessica Tatiana Carrasco Ortiz. Green computing in erlang. Master's thesis, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, 2017.

Appendix A

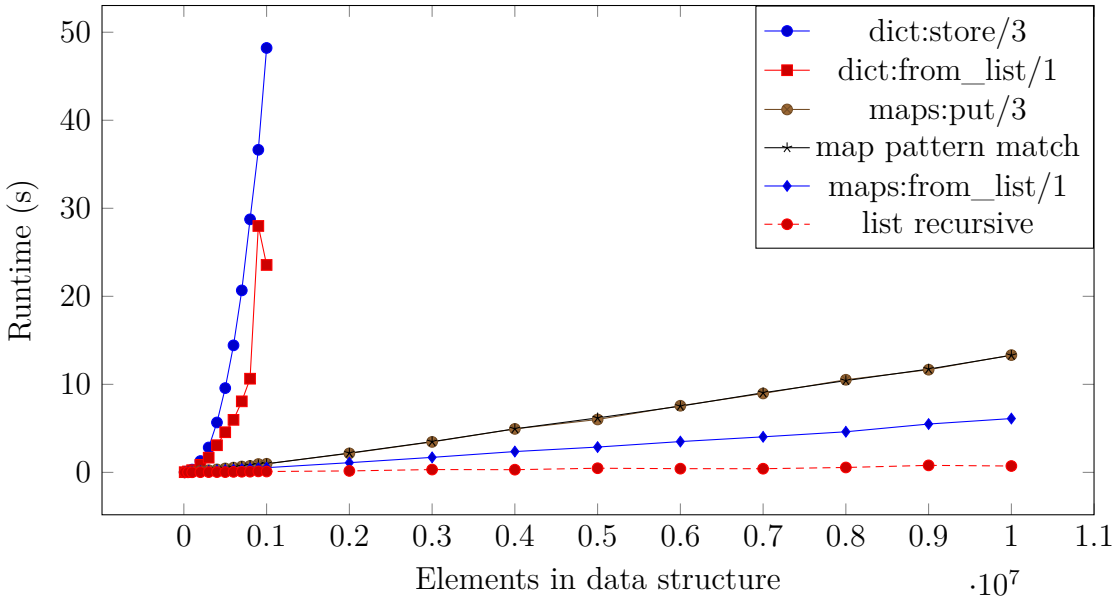
Comparison

A.1 Data structures

Build all

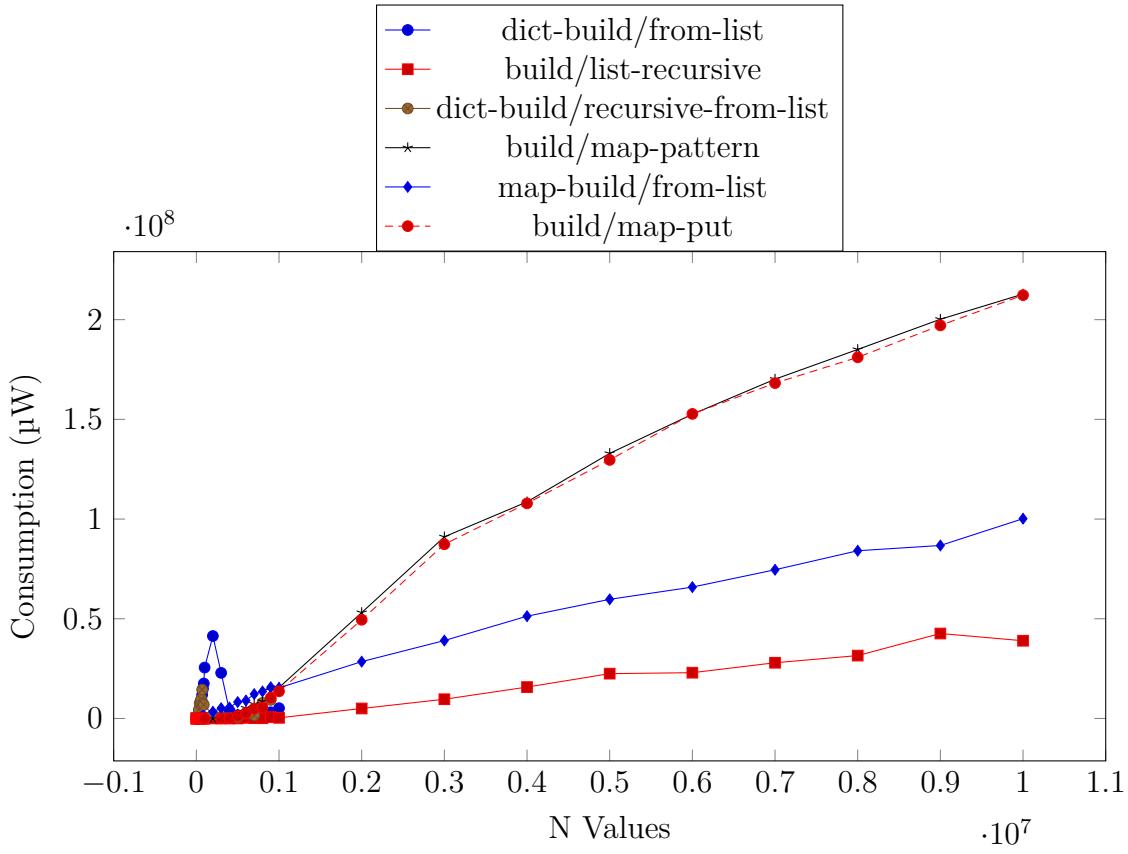


(a) Linux: Energy consumption values

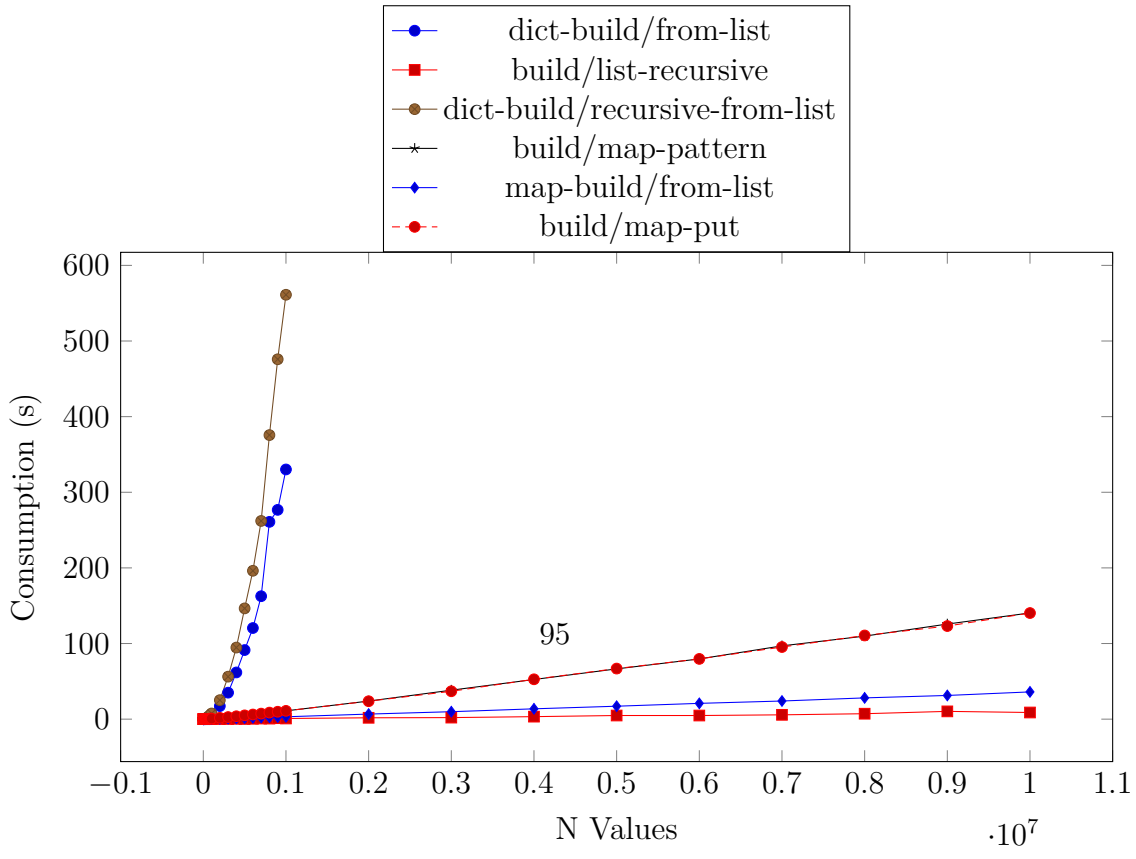


(b) Linux: Runtime values

Figure A.1: Linux: Energy cost and runtime of creating different data structures in various modes, note that map and dict from list creation includes the cost of recursively creating a list as well.



(a) Windows: Energy consumption values



(b) Windows: Runtime values

Finding the value for a given key

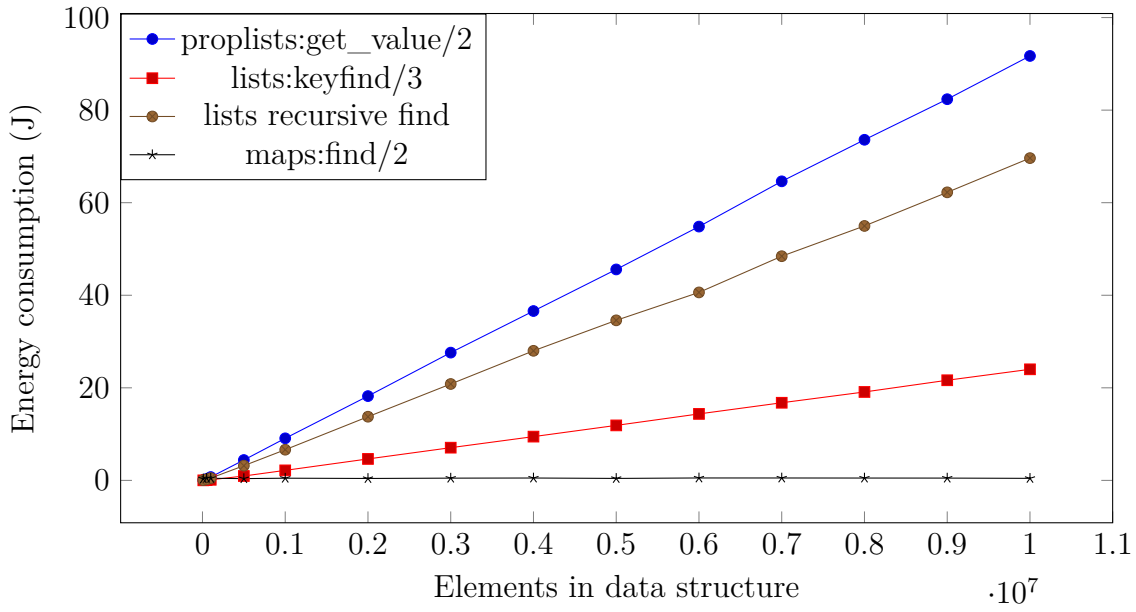


Figure A.3: Linux: Energy cost of finding an element in a list (`maps:find/2` is shown for reference). All measurements on this figure consist of 100 repeated finds.

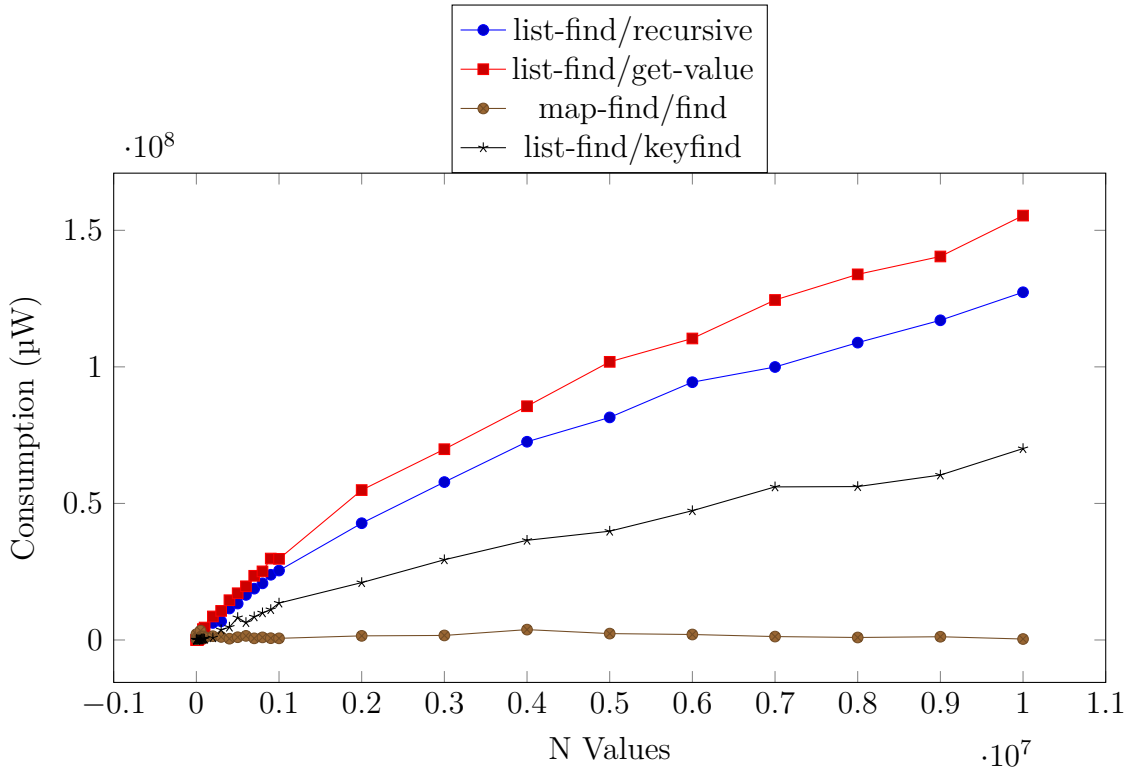
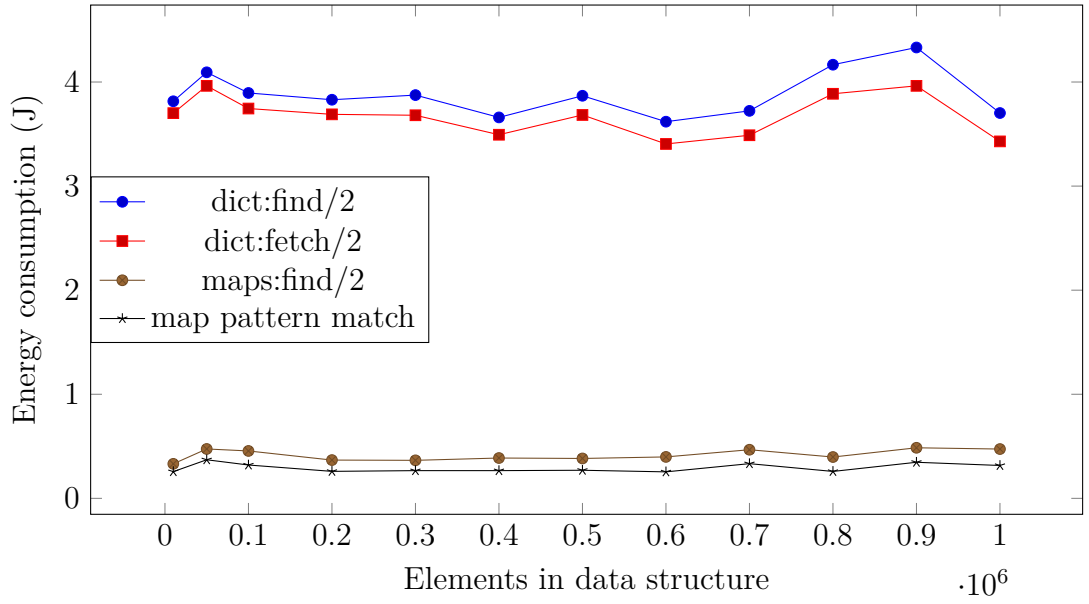
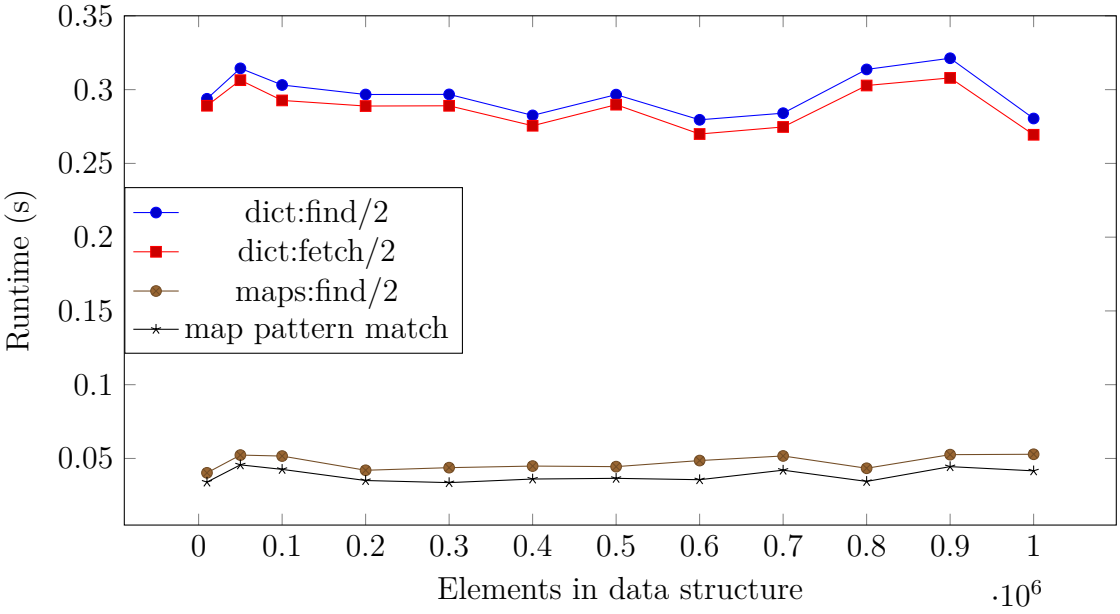


Figure A.4: Windows: Energy cost of finding an element in a list (`maps:find/2` is shown for reference). All measurements on this figure consist of 100 repeated finds.

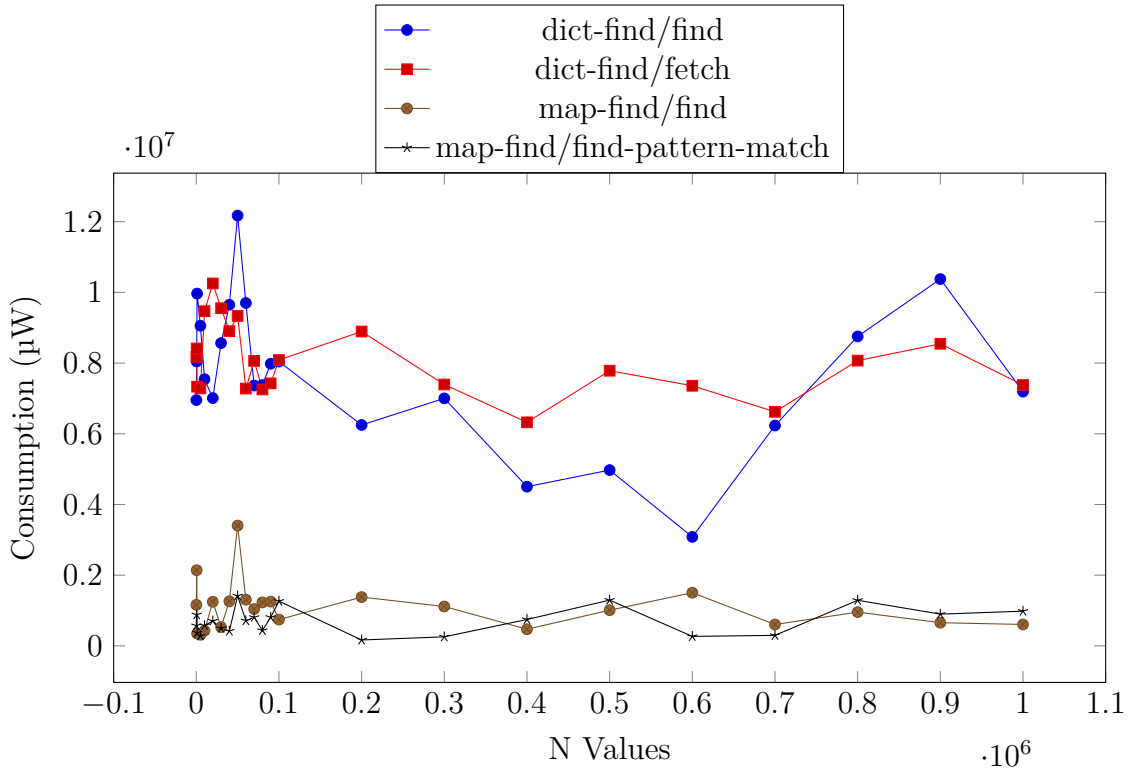


(a) Linux: Energy consumption values

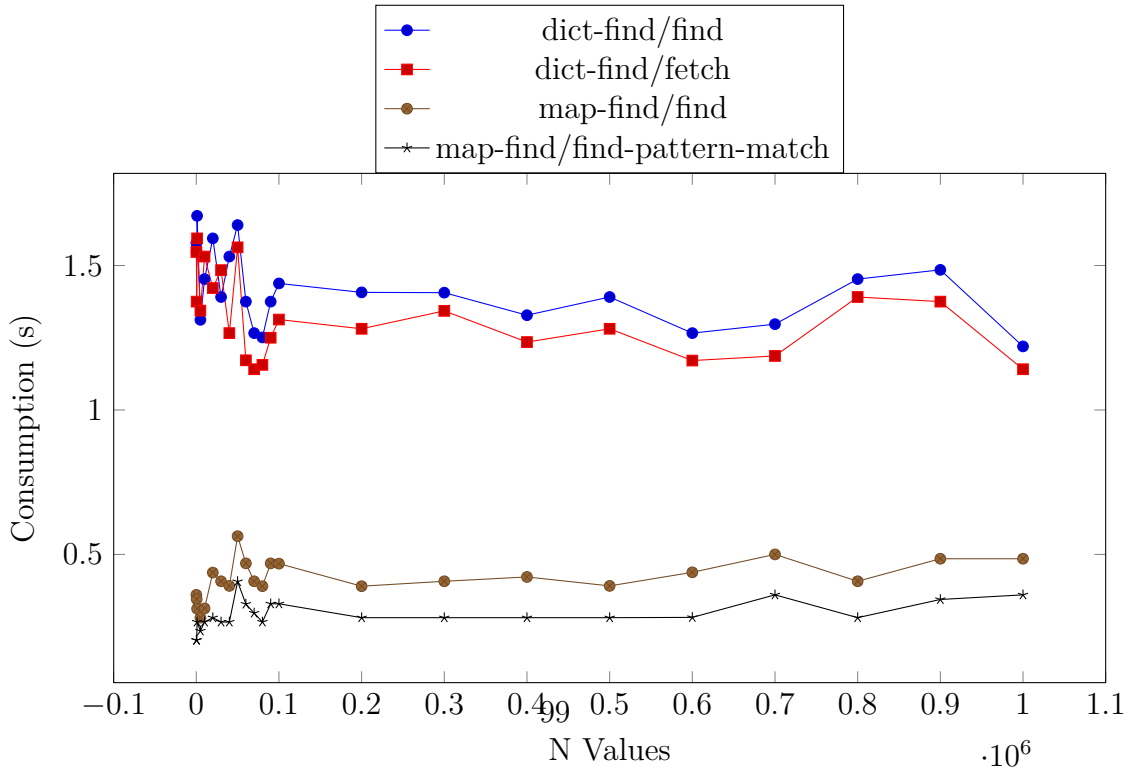


(b) Linux: Runtime values

Figure A.5: Linux: Energy cost and runtime of searching an element in a map or a dict. All measurements on this figure consist of 1 000 000 repeated finds.



(a) Windows: Energy consumption values



(b) Windows: Runtime values

Figure A.6: Windows: Energy cost and runtime of searching an element in a map or a dict. All measurements on this figure consist of 1 000 000 repeated finds.

Updating a single element

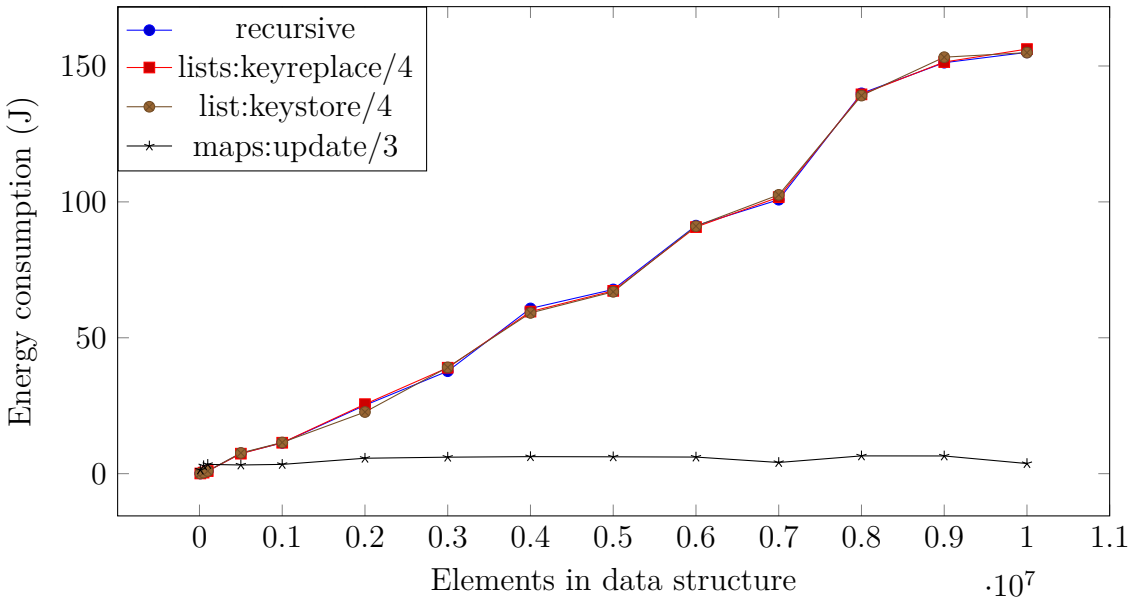


Figure A.7: Linux: Energy cost of updating an element in a list (`maps:update/3` is shown for reference). All measurements on this figure consist of 100 repeated updates.

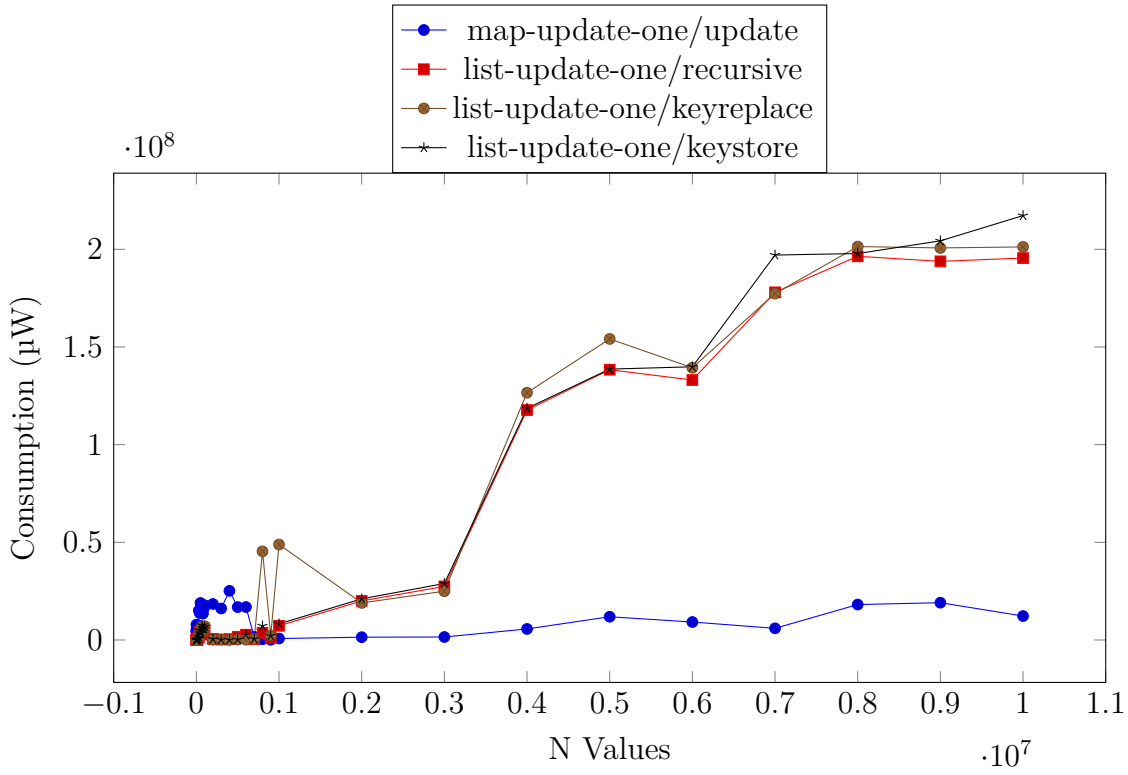


Figure A.8: Windows: Energy cost of updating an element in a list (`maps:update/3` is shown for reference). All measurements on this figure consist of 100 repeated updates.

Updating all elements

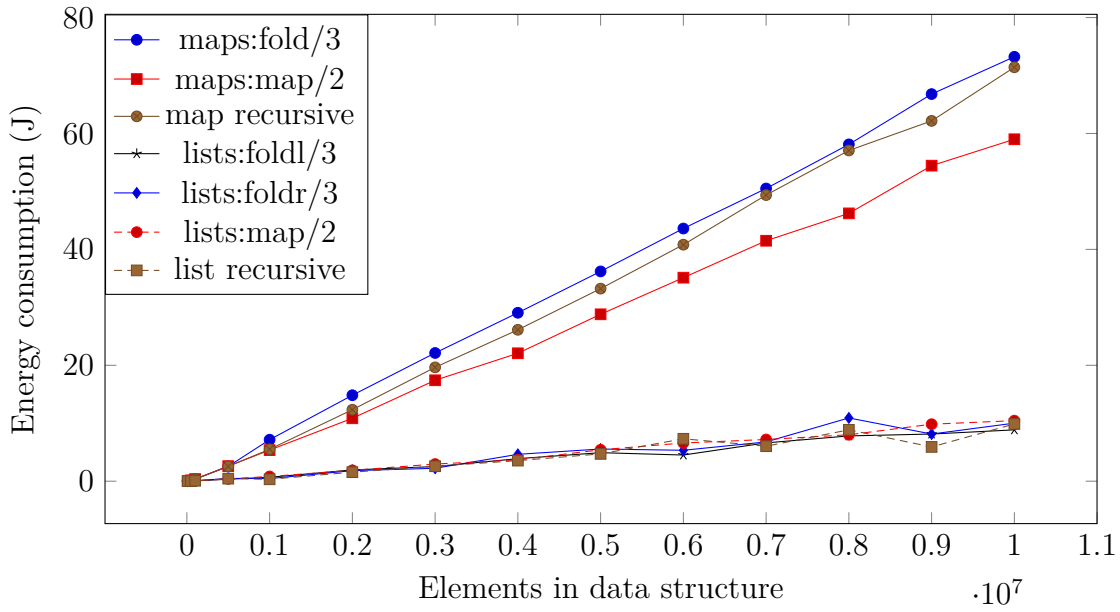


Figure A.9: Linux: Energy cost of updating all elements in a map or in a list.

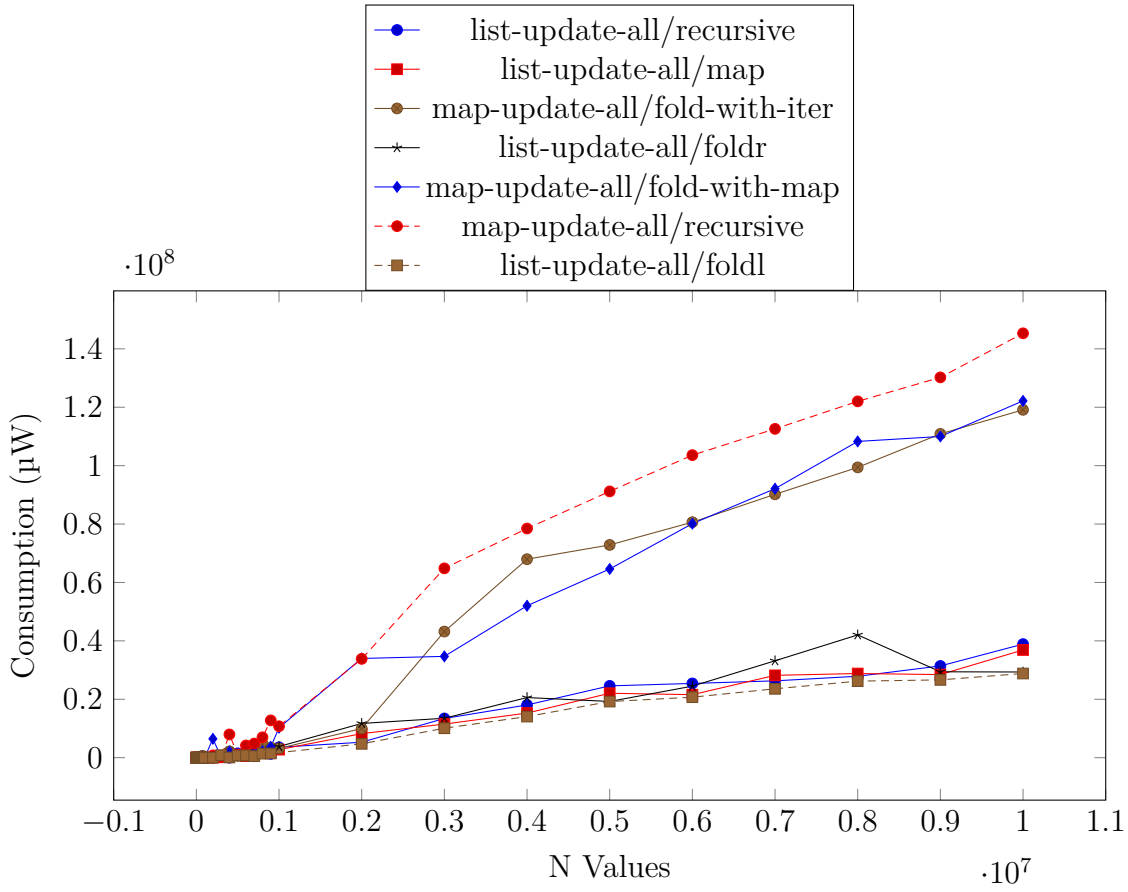


Figure A.10: Windows: Energy cost of updating all elements in a map or in a list.

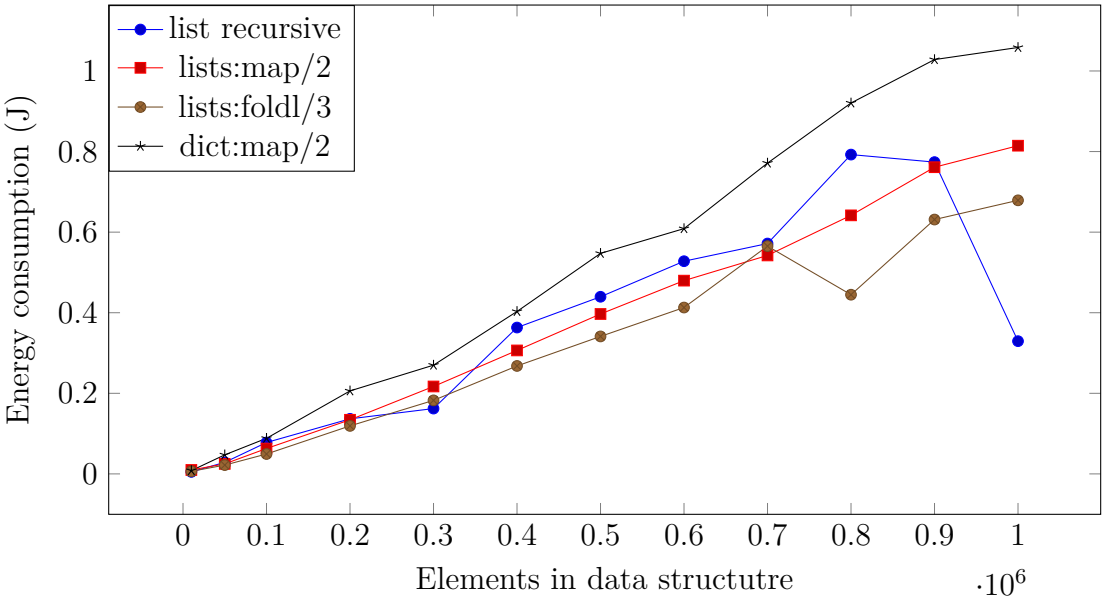


Figure A.11: Linux: Energy cost of updating all elements in a dictionary or in a list.

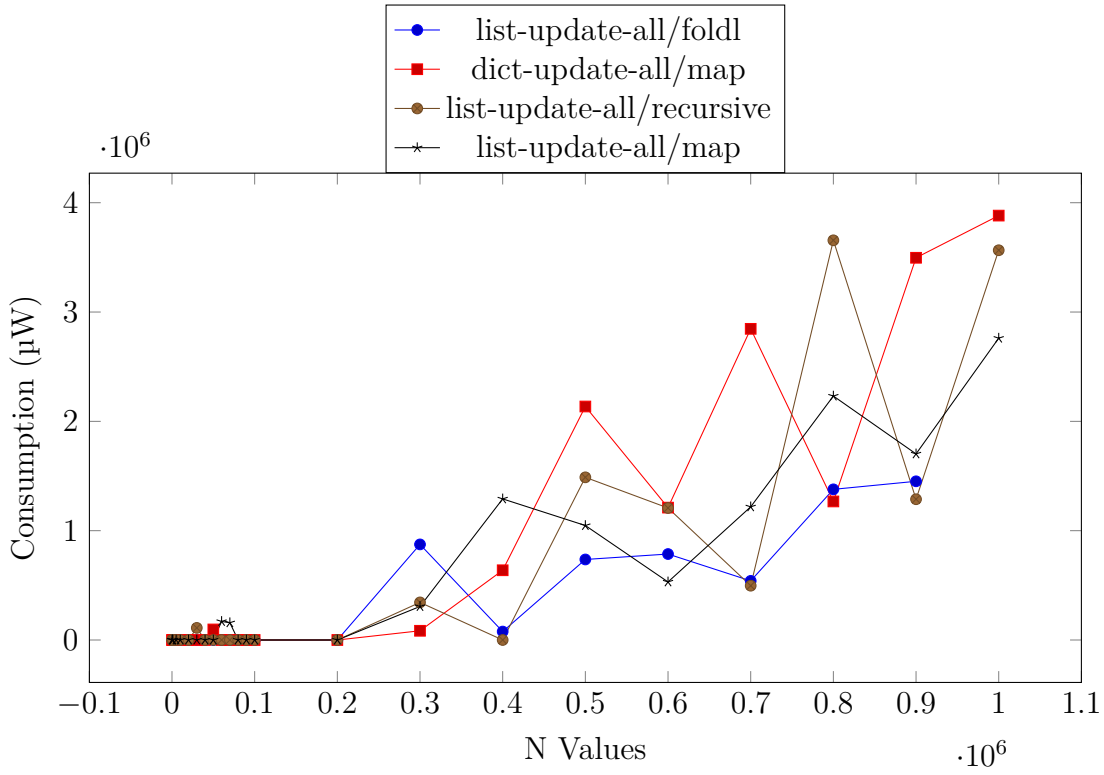


Figure A.12: Windows: Energy cost of updating all elements in a dictionary or in a list.

Deleting

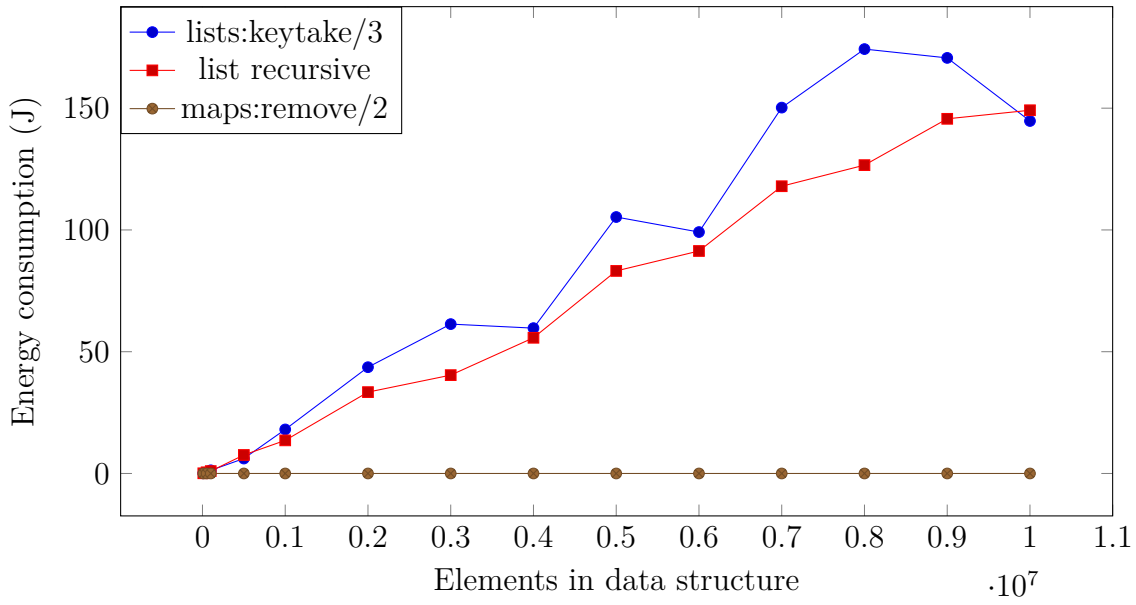


Figure A.13: Linux: Energy cost of deleting an element from a list (`maps:remove/2` is shown for reference). All measurements on this figure consist of 100 repeated deletes.

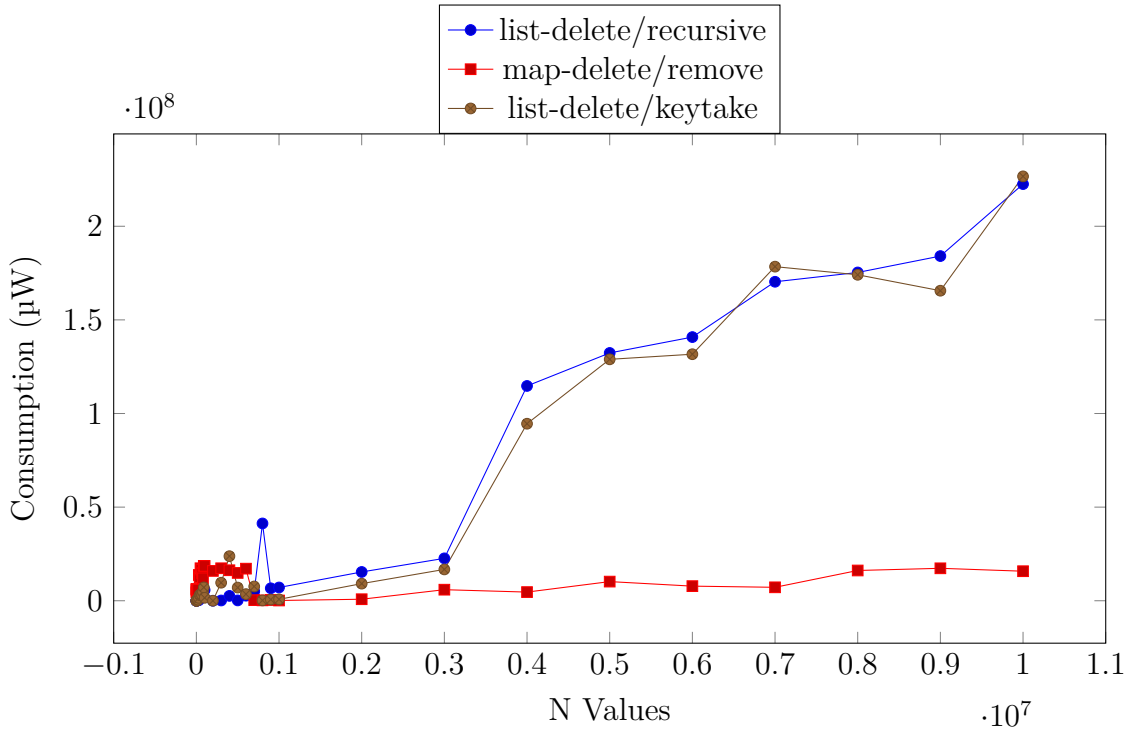


Figure A.14: Windows: Energy cost of deleting an element from a list (`maps:remove/2` is shown for reference). All measurements on this figure consist of 100 repeated deletes.

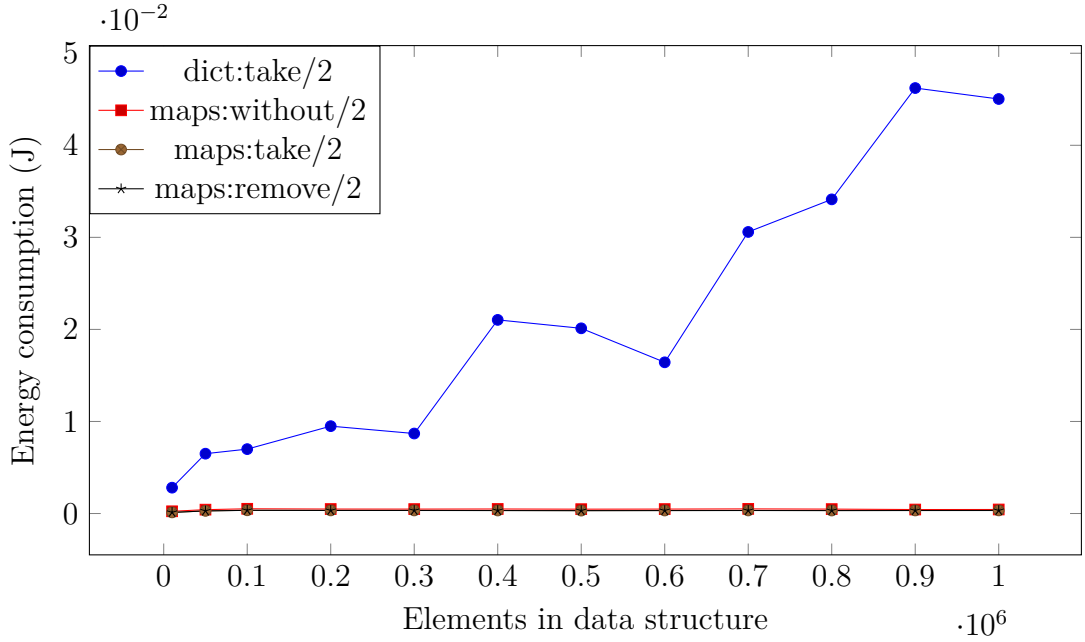


Figure A.15: Linux: Energy cost of deleting an element from a dictionary or from a map. All measurements on this figure consist of 100 repeated deletes.

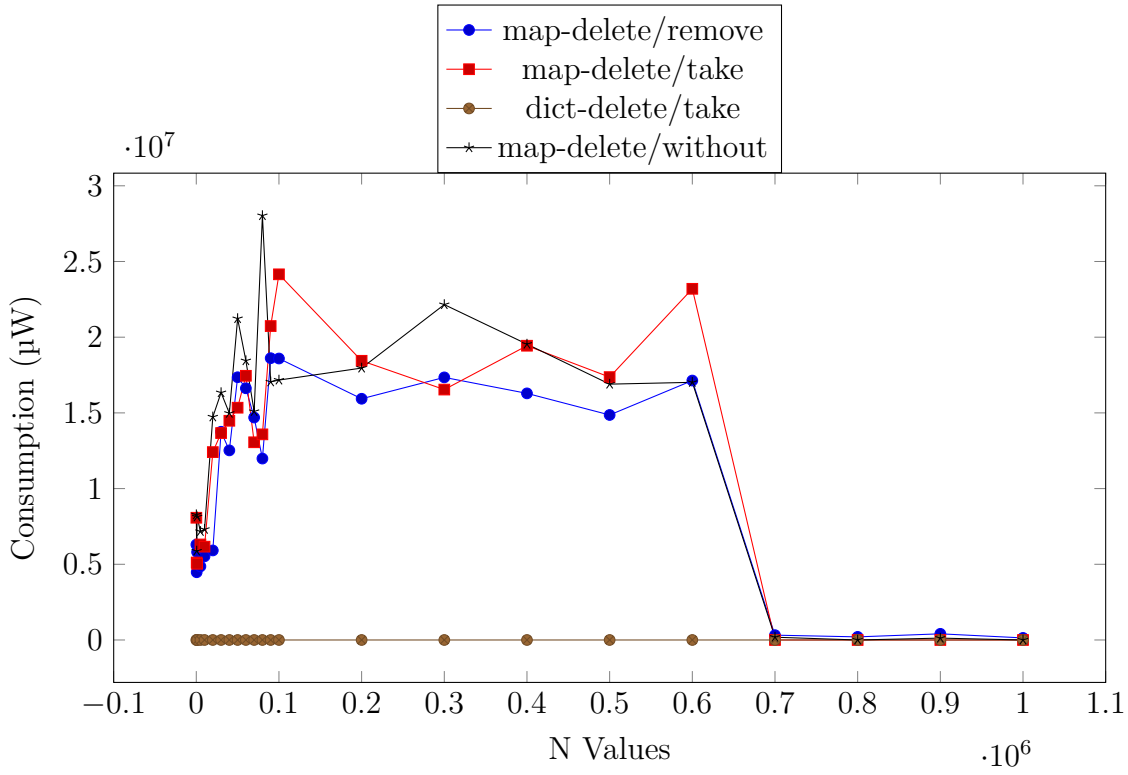


Figure A.16: Windows: Energy cost of deleting an element from a dictionary or from a map. All measurements on this figure consist of 100 repeated deletes.

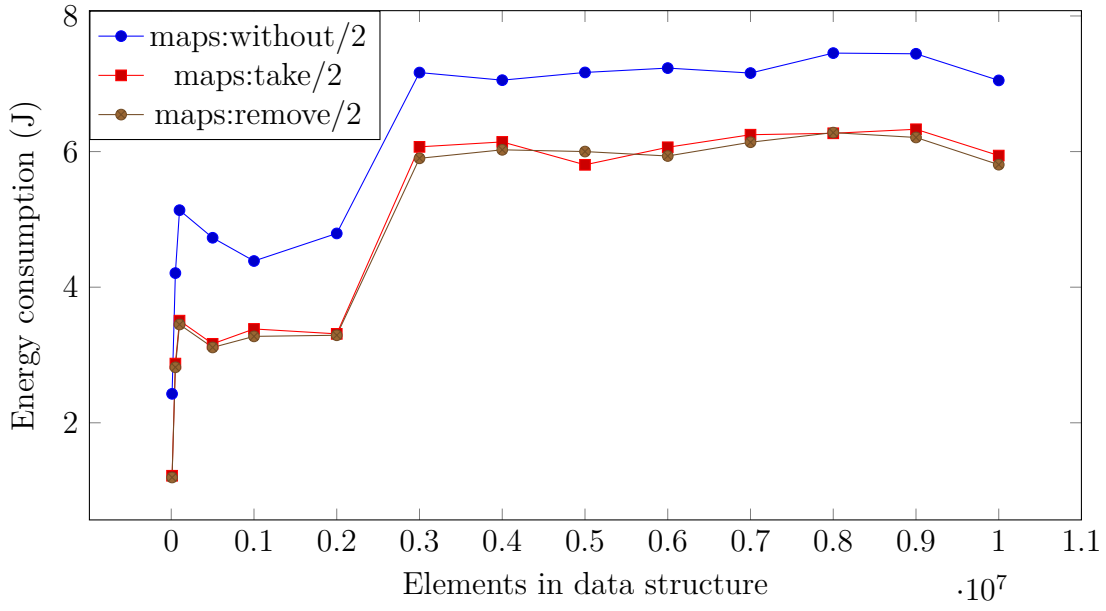


Figure A.17: Linux: Energy cost of deleting an element from a map. All measurements on this figure consist of 1 000 000 repeated deletes.

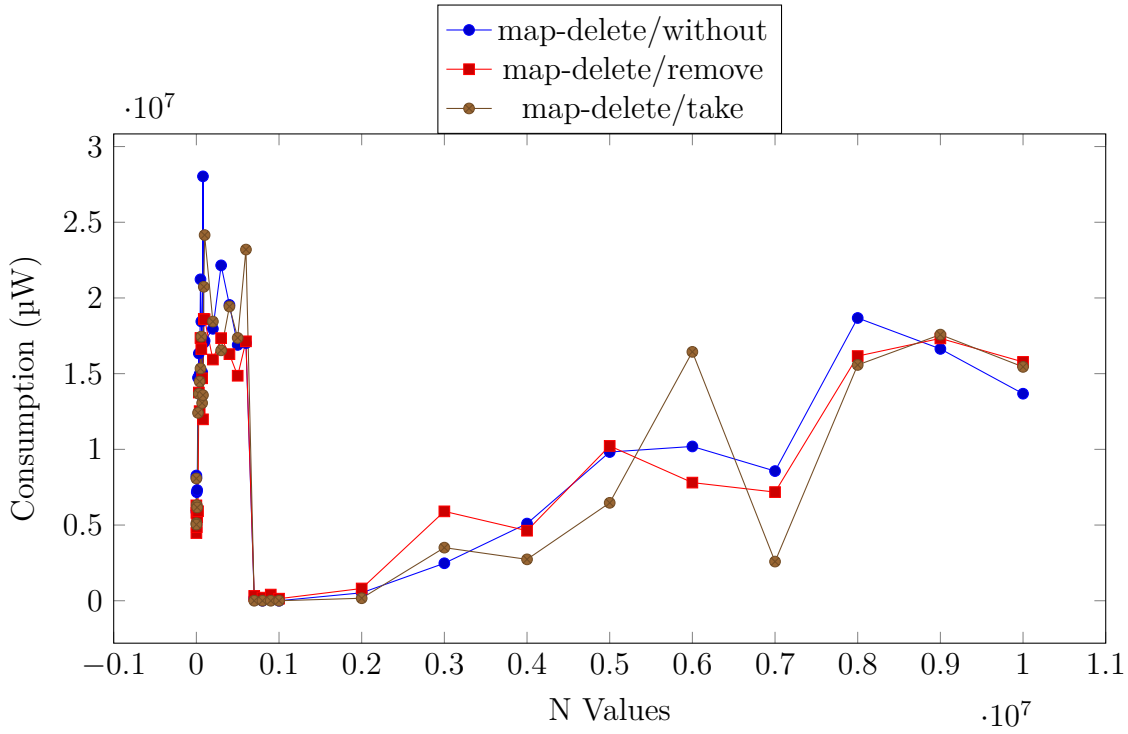
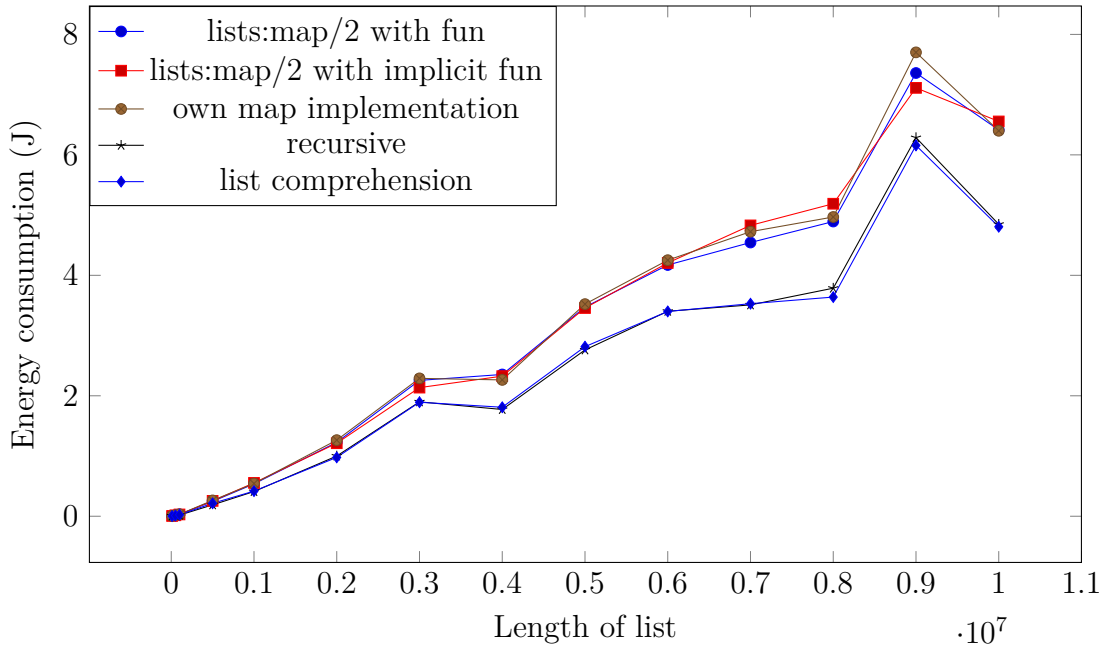


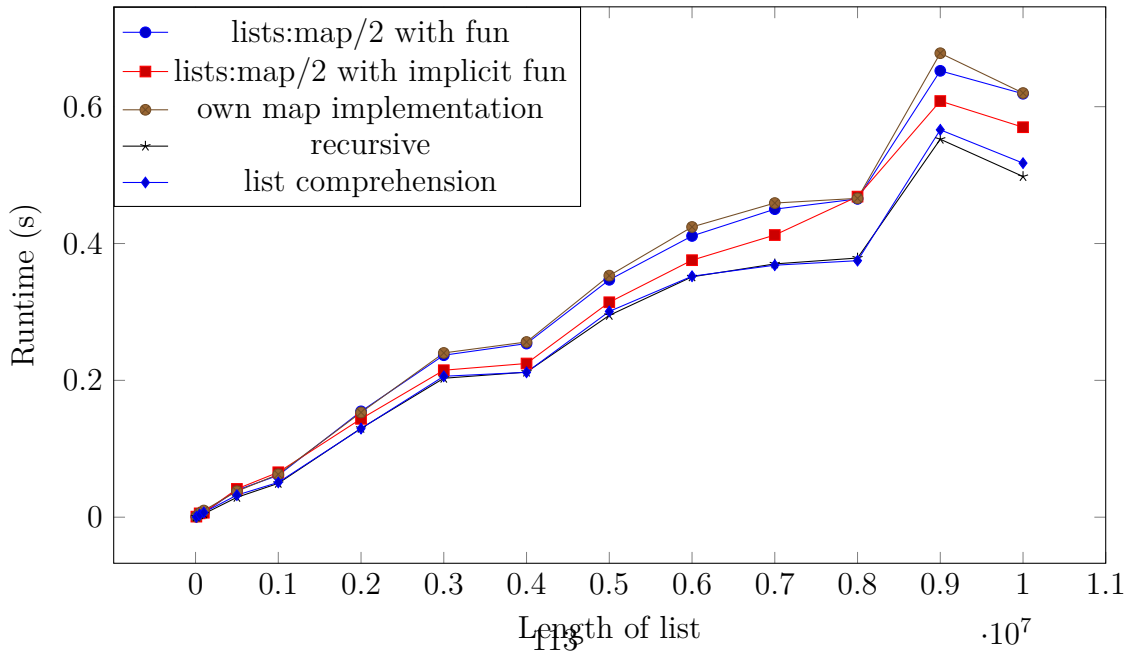
Figure A.18: Windows: Energy cost of deleting an element from a map. All measurements on this figure consist of 1 000 000 repeated deletes.

A.2 Higher-order functions

Map

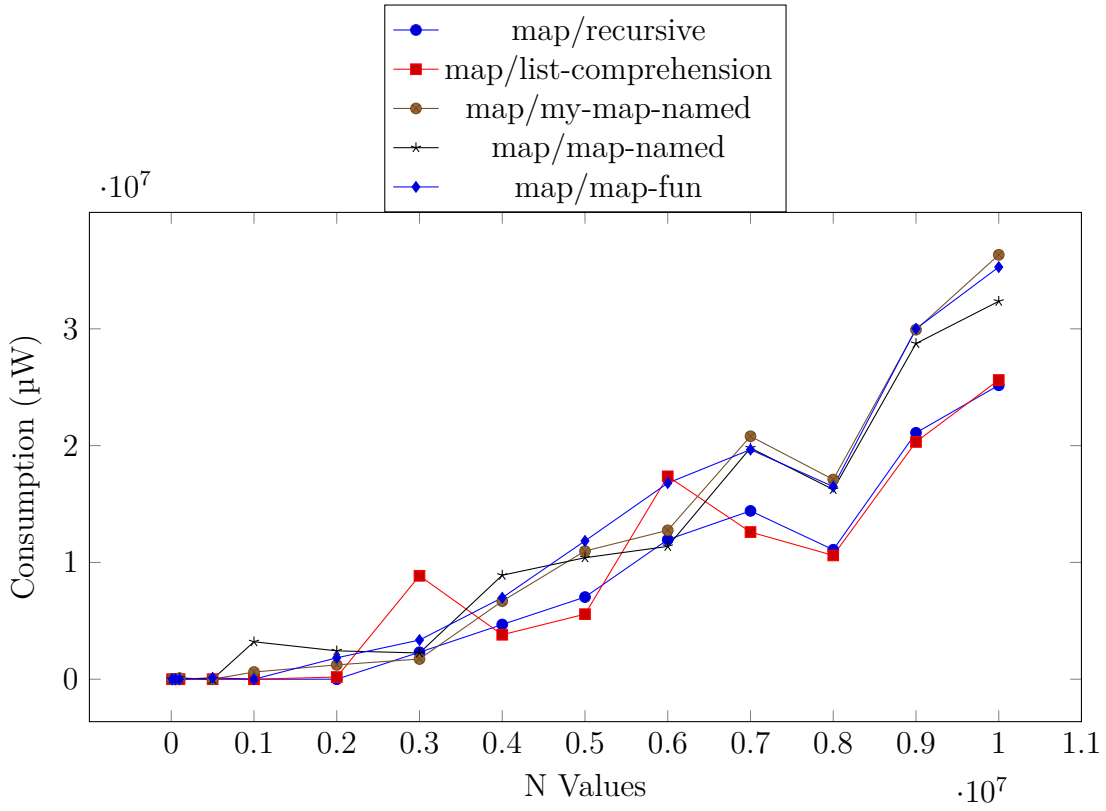


(a) Linux: Energy consumption values

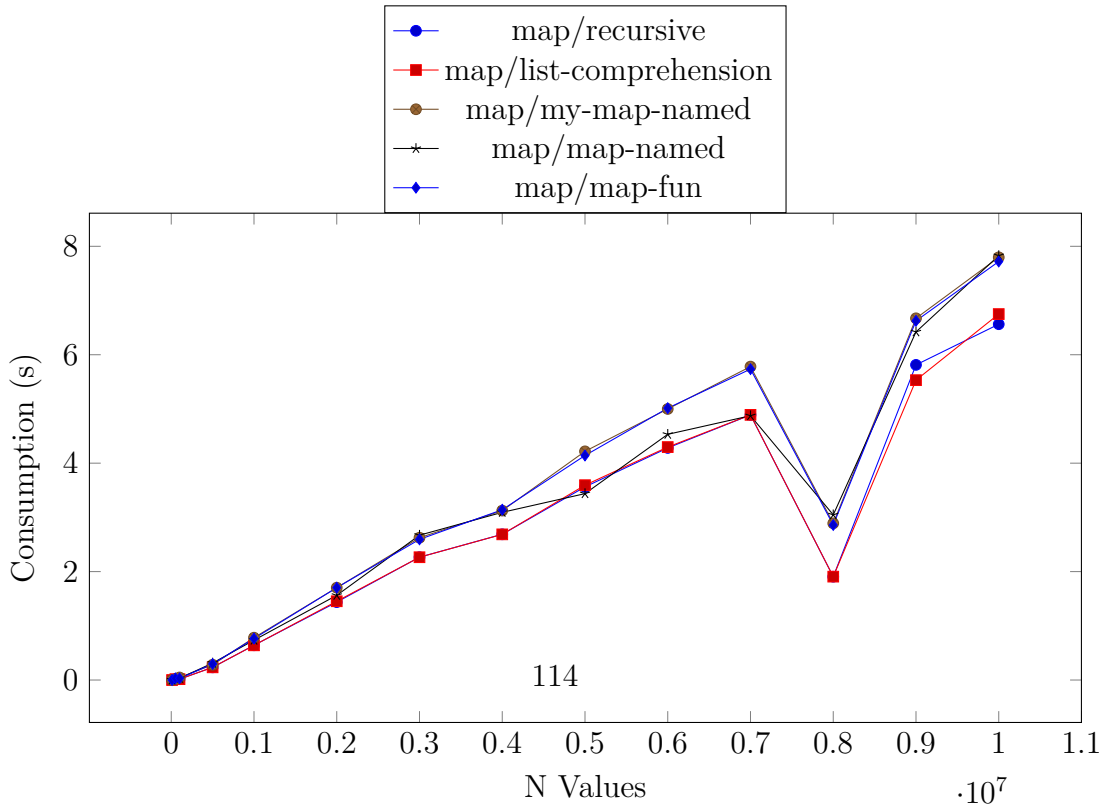


(b) Linux: Runtime values

Figure A.19: Linux: Energy cost and runtime of different map implementations and function calls.



(a) Windows: Energy consumption values



(b) Windows: Runtime values

Figure A.20: Windows: Energy cost and runtime of different map implementations and function calls.

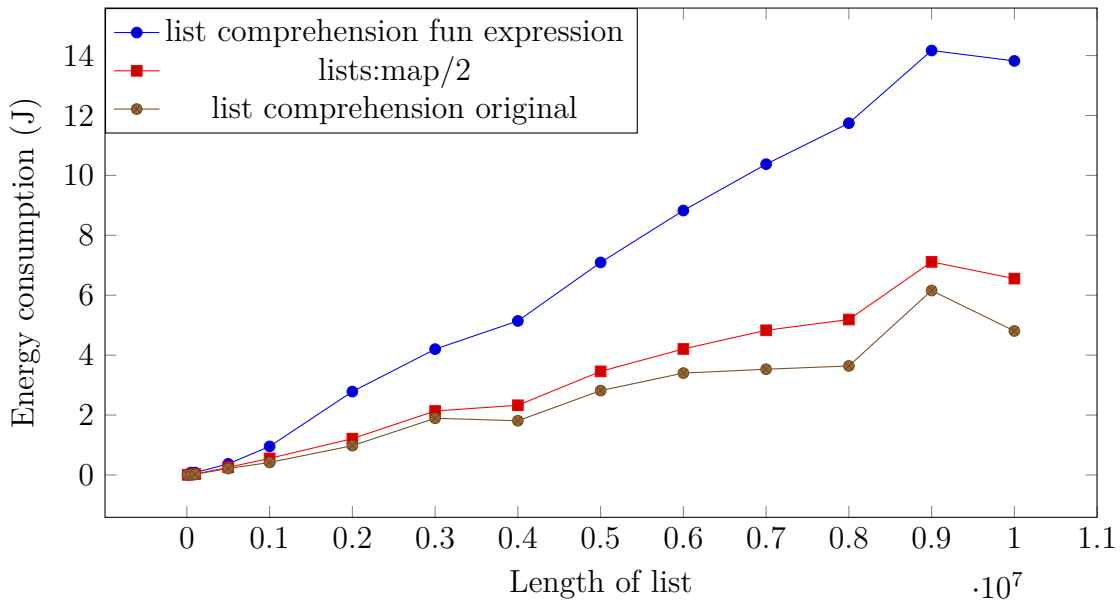


Figure A.21: Linux: Energy consumption values using list comprehensions with fun expression compared to inlining the function definition and higher-order function calls.

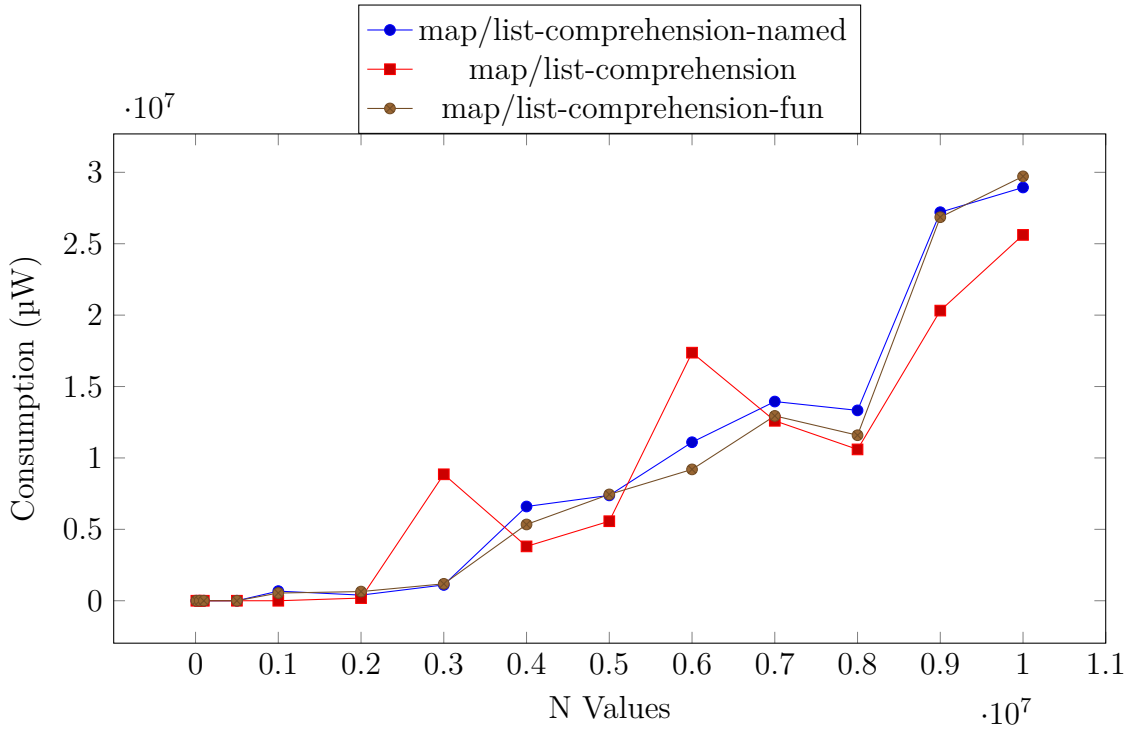


Figure A.22: Windows: Energy consumption values using list comprehensions with fun expression compared to inlining the function definition and higher-order function calls.

Filter

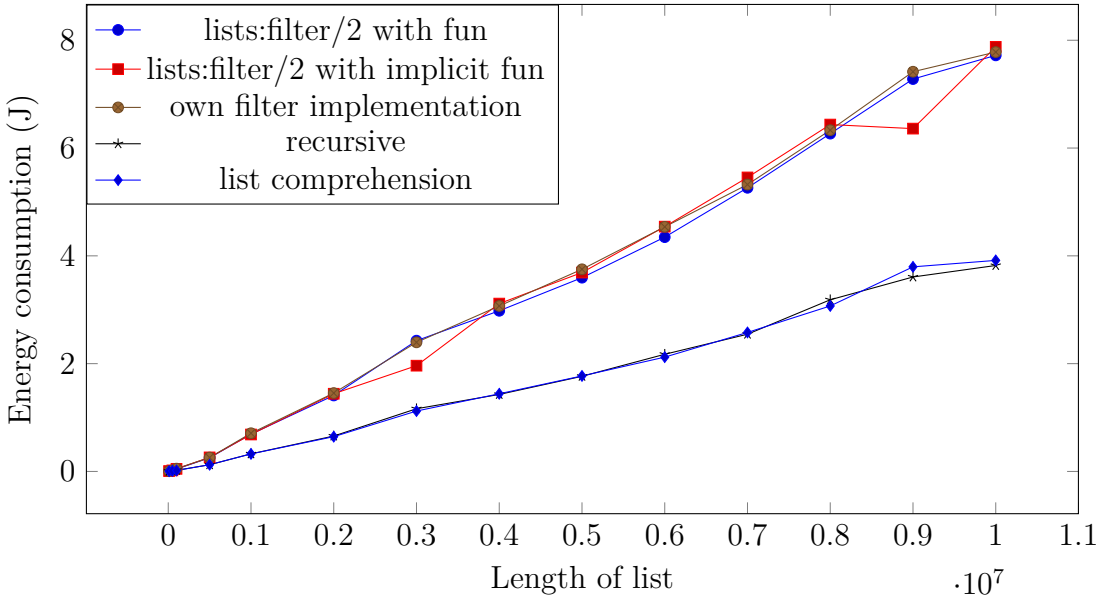


Figure A.23: Linux: Energy consumption values for different filter implementations and function calls.

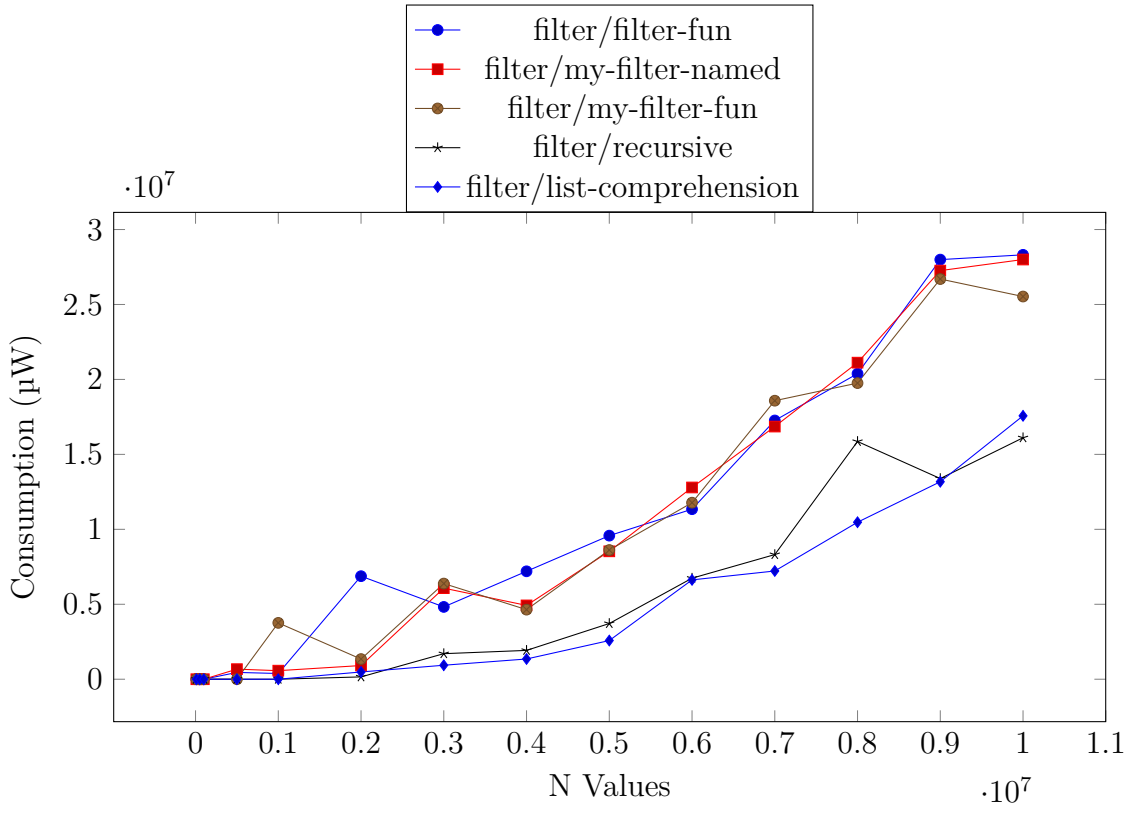


Figure A.24: Windows: Energy consumption values for different filter implementations and function calls.

Filter-map

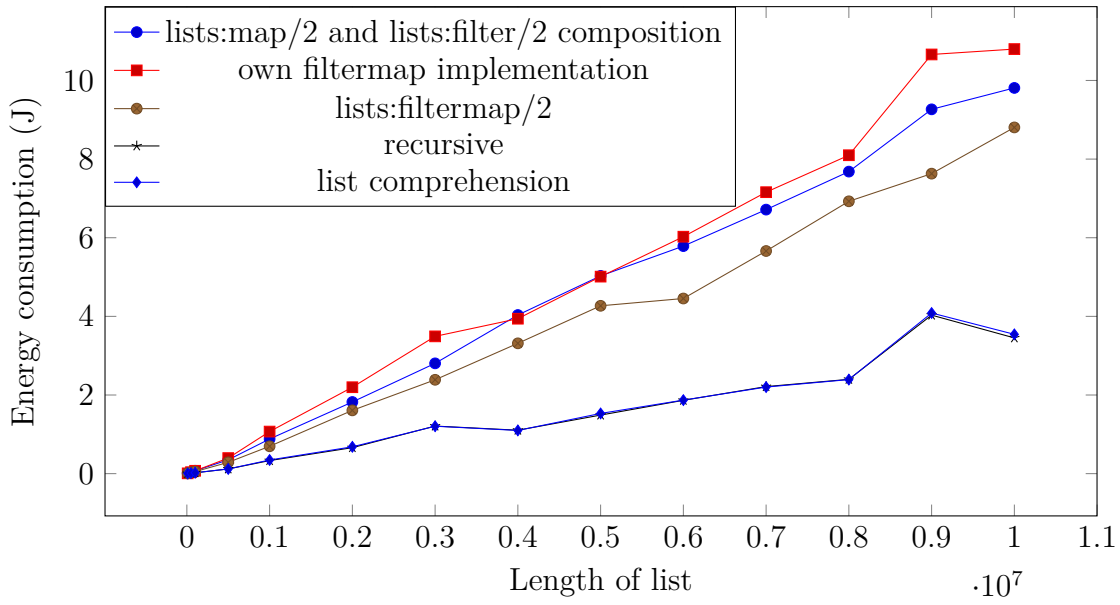


Figure A.25: Linux: Energy consumption values for some different filtermap implementations and function calls.

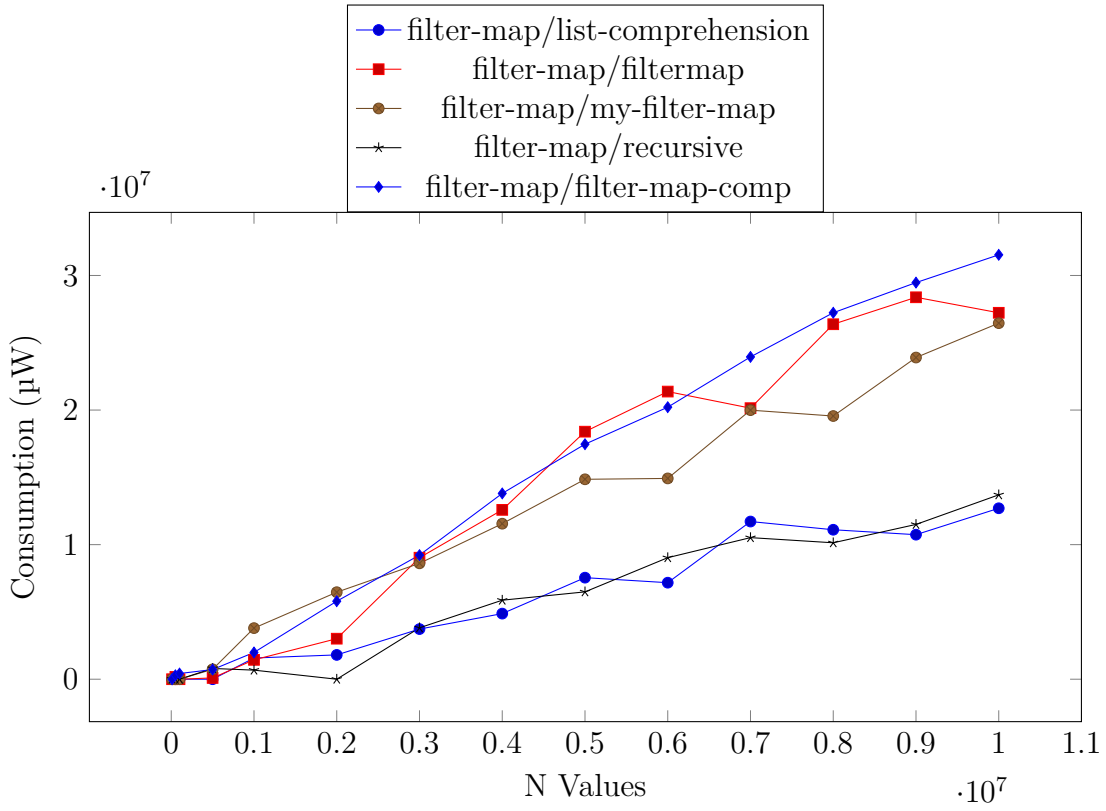


Figure A.26: Windows: Energy consumption values for some different filtermap implementations and function calls.

A.3 Parallel language constructs

Sending raw data structures

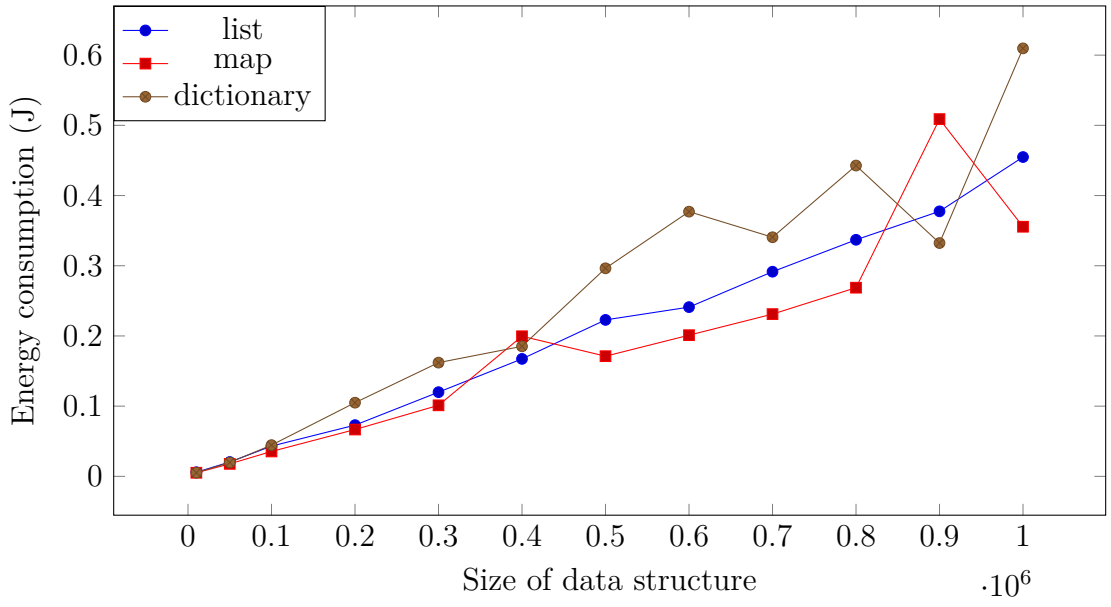


Figure A.27: Linux: Energy consumption values for sending data structures once.

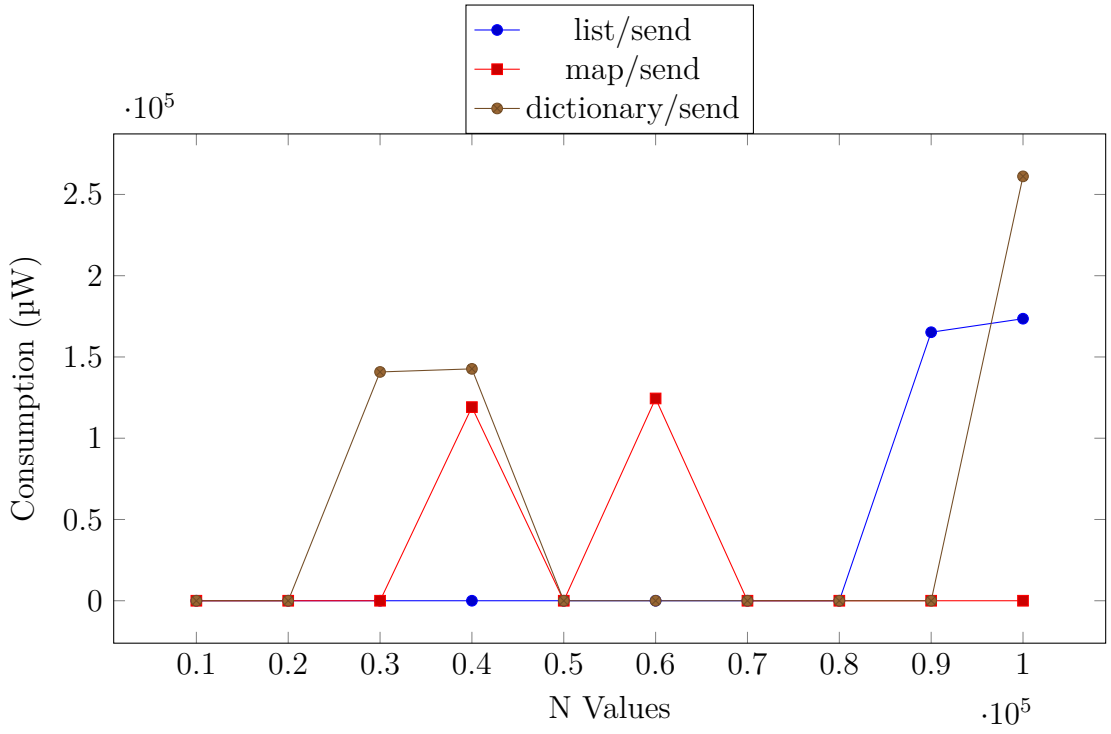


Figure A.28: Windows: Energy consumption values for sending data structures once.

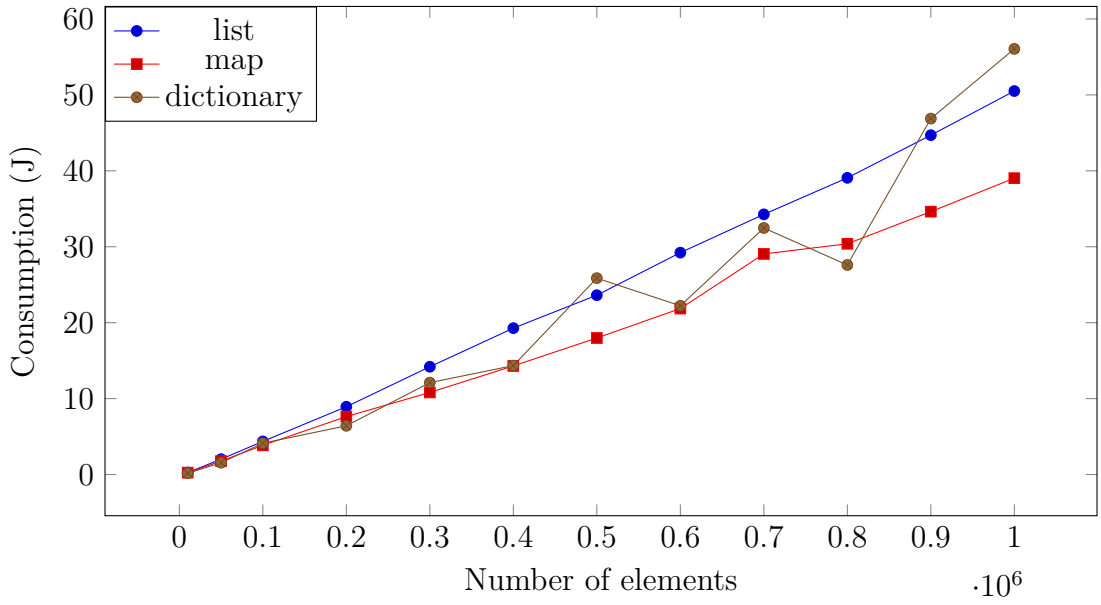


Figure A.29: Linux: Energy consumption values for sending data structures 100 times.

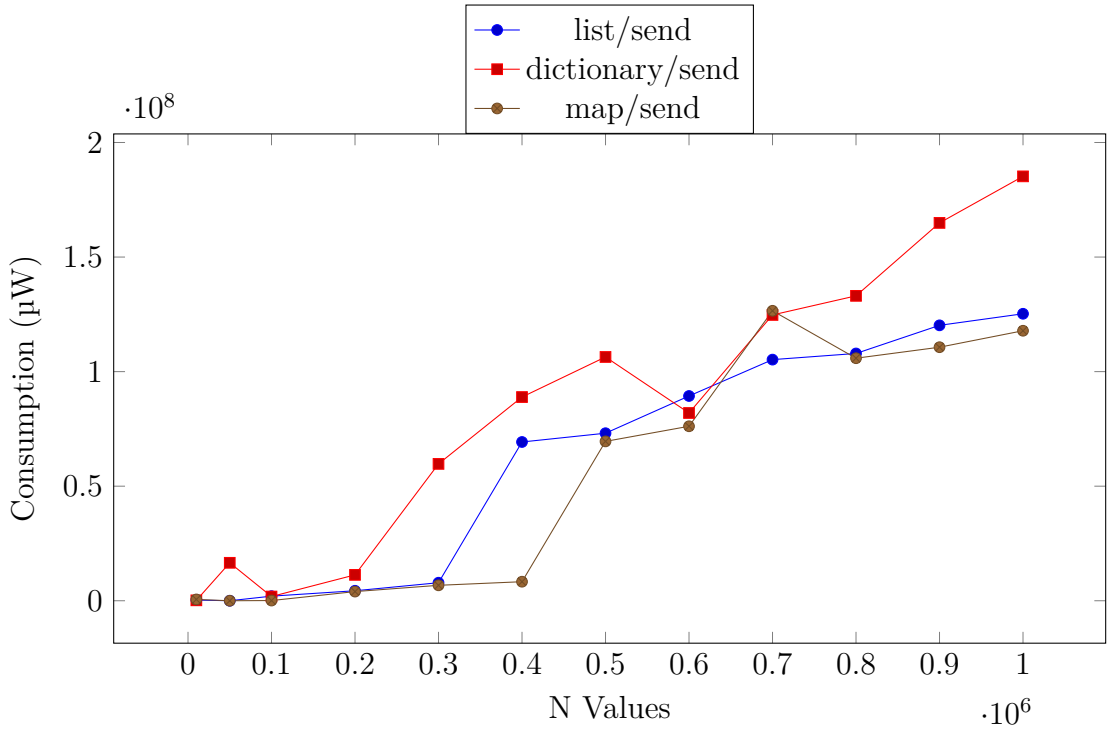


Figure A.30: Windows: Energy consumption values for sending data structures 100 times.

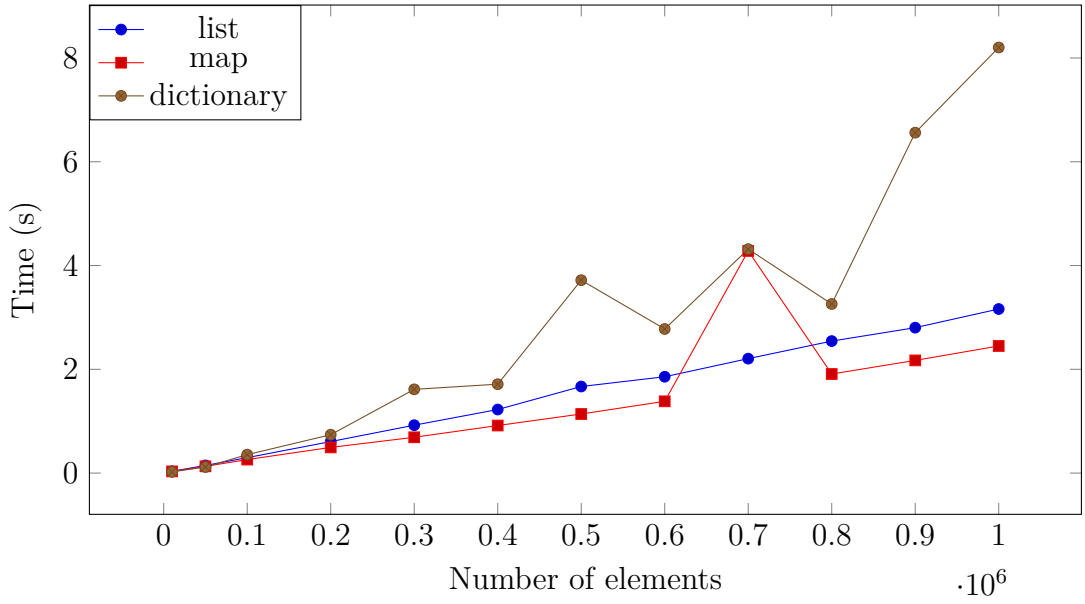


Figure A.31: Linux: Runtime values for sending data structures 100 times.

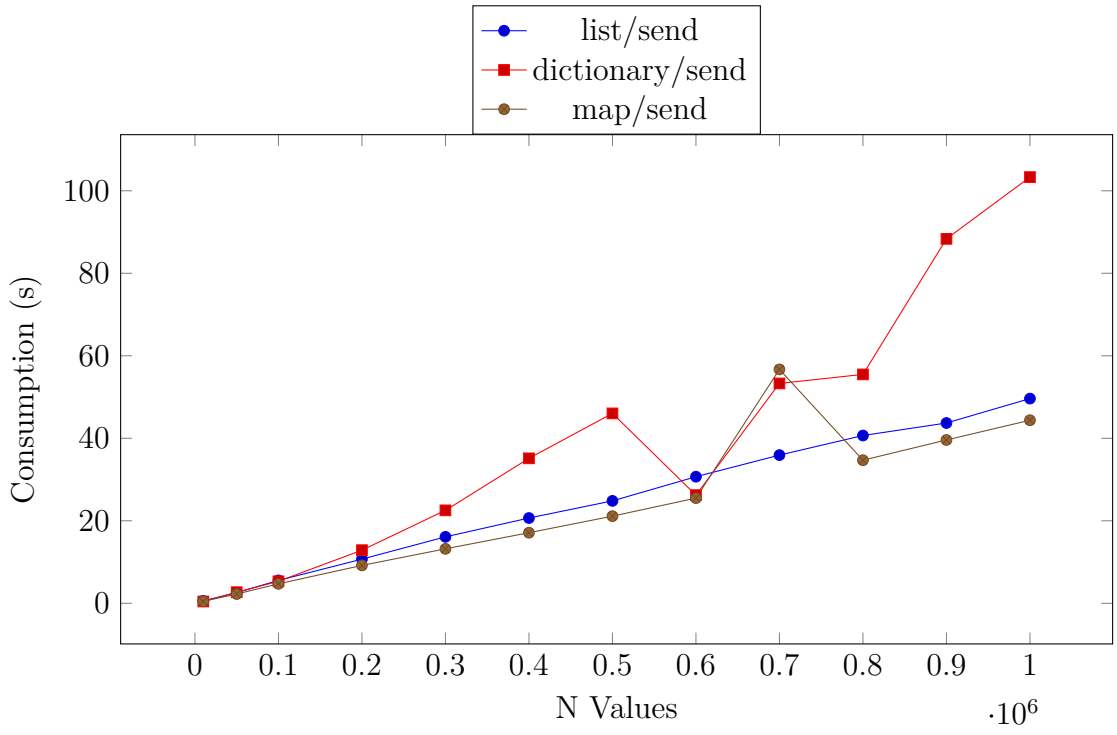


Figure A.32: Windows: Runtime values for sending data structures 100 times.

Sending data in pieces

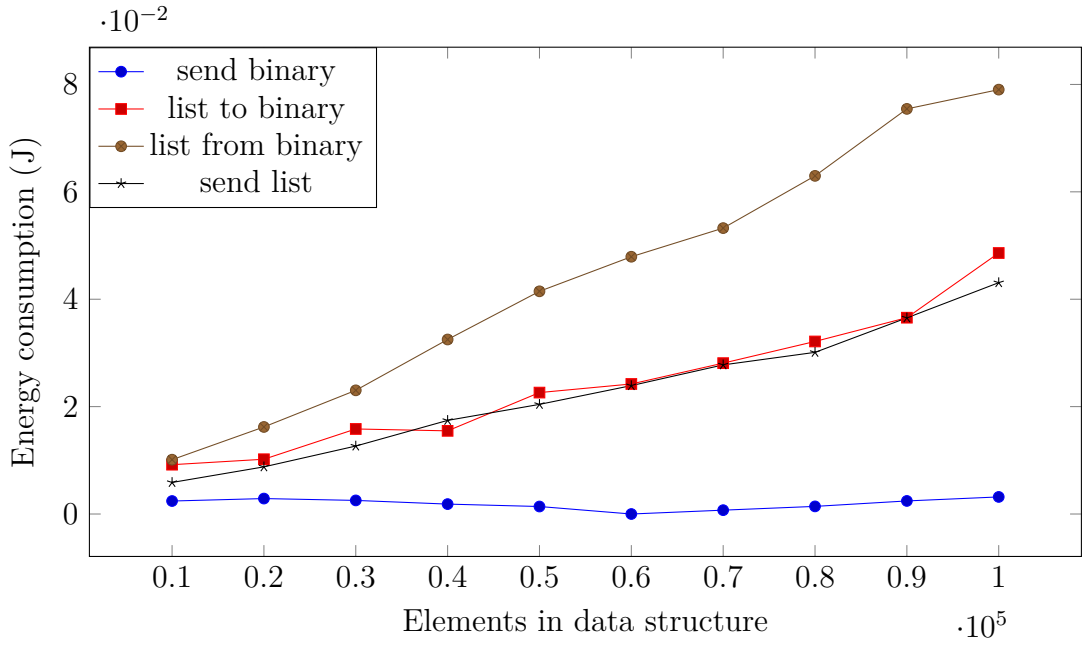


Figure A.33: Linux: Energy consumption values for list-binary operations.

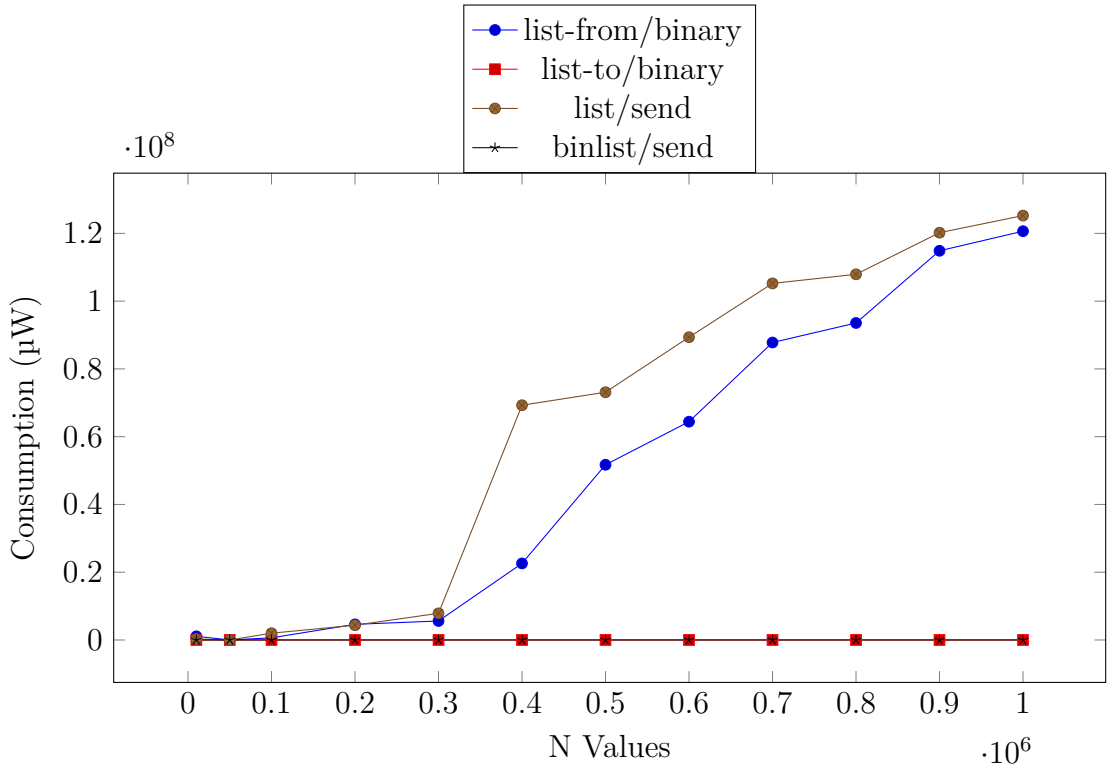


Figure A.34: Windows: Energy consumption values for list-binary operations.

A.4 Algorithmic skeletons

Spawning different number of workers

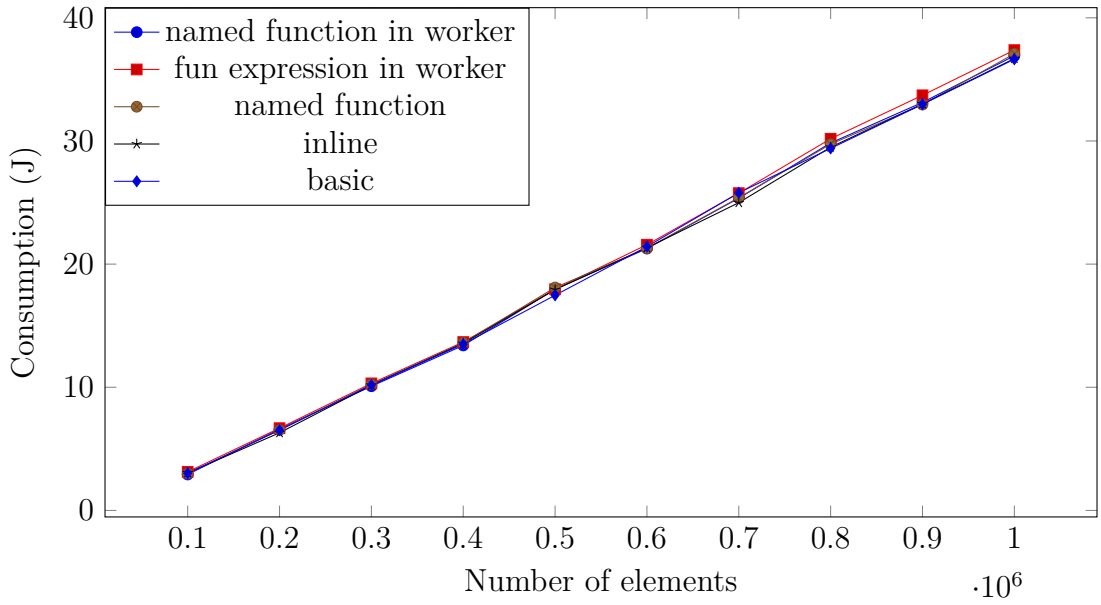


Figure A.35: Linux: Energy consumption values for different task farm implementations.

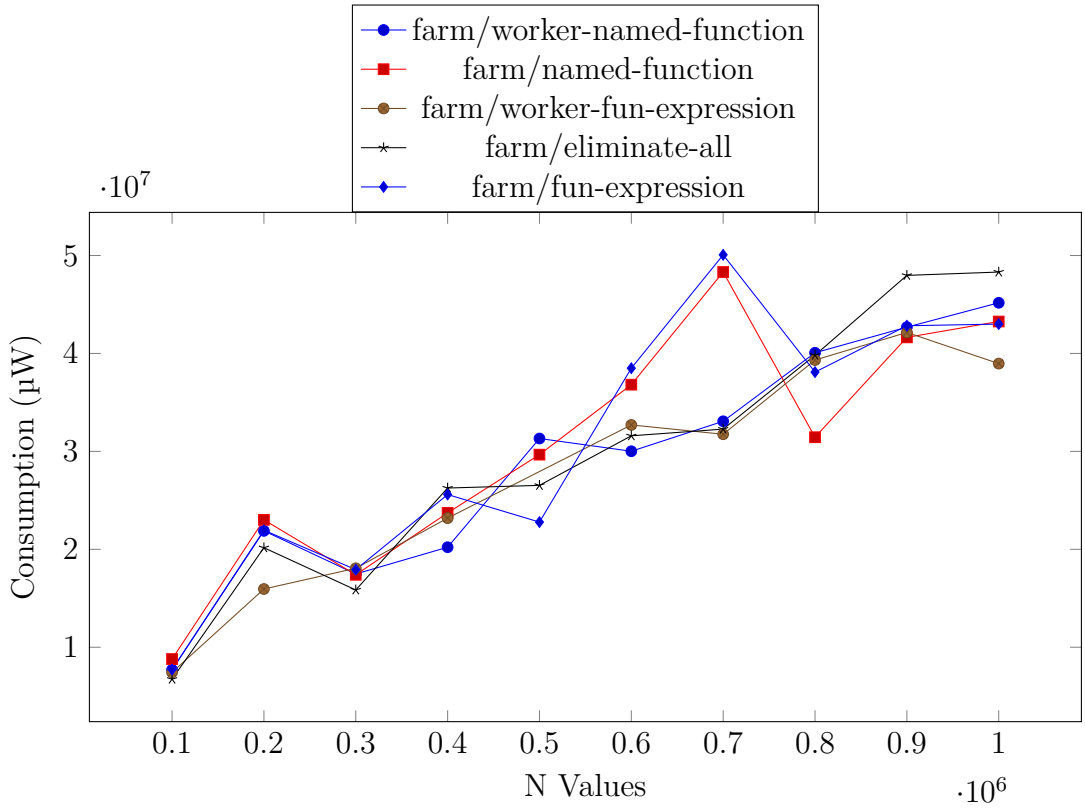


Figure A.36: Windows: Energy consumption values for different task farm implementations.

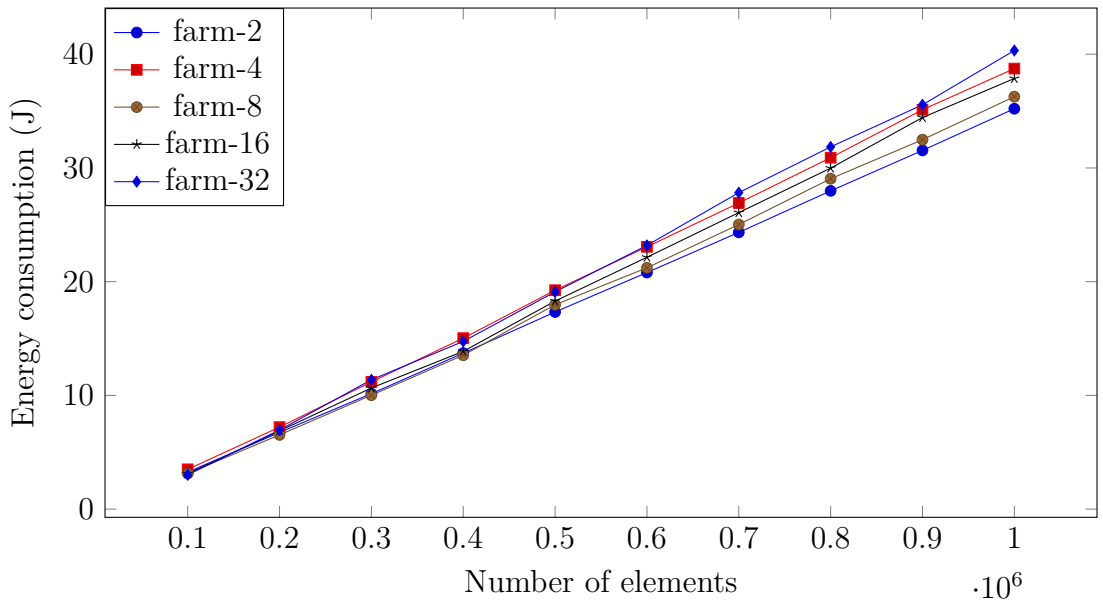


Figure A.37: Linux: Energy consumption values for different number of workers.

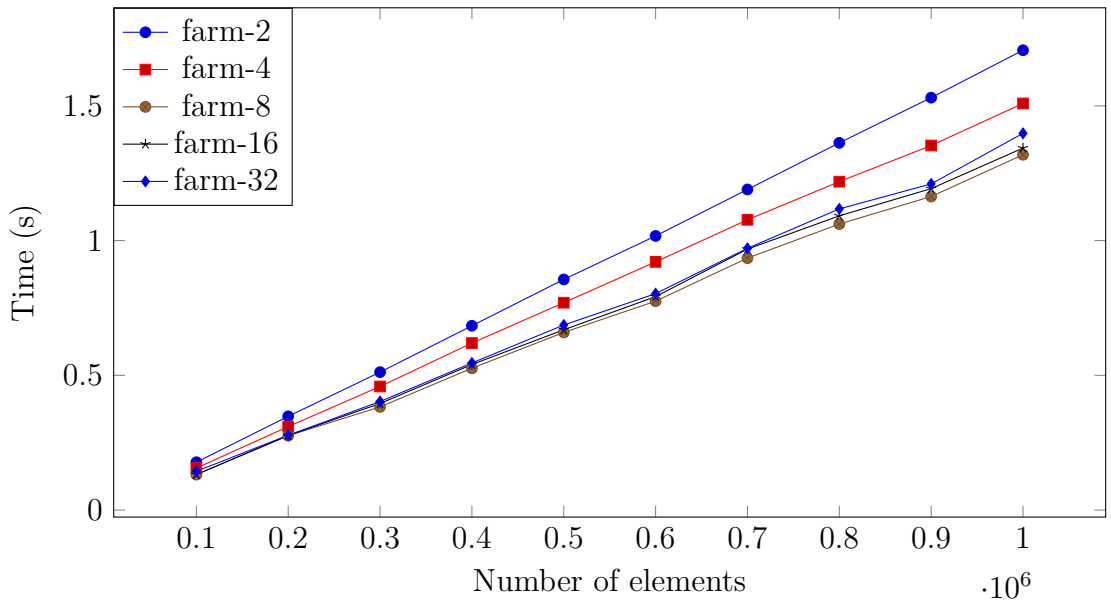


Figure A.38: Linux: Runtimes for different number of workers.

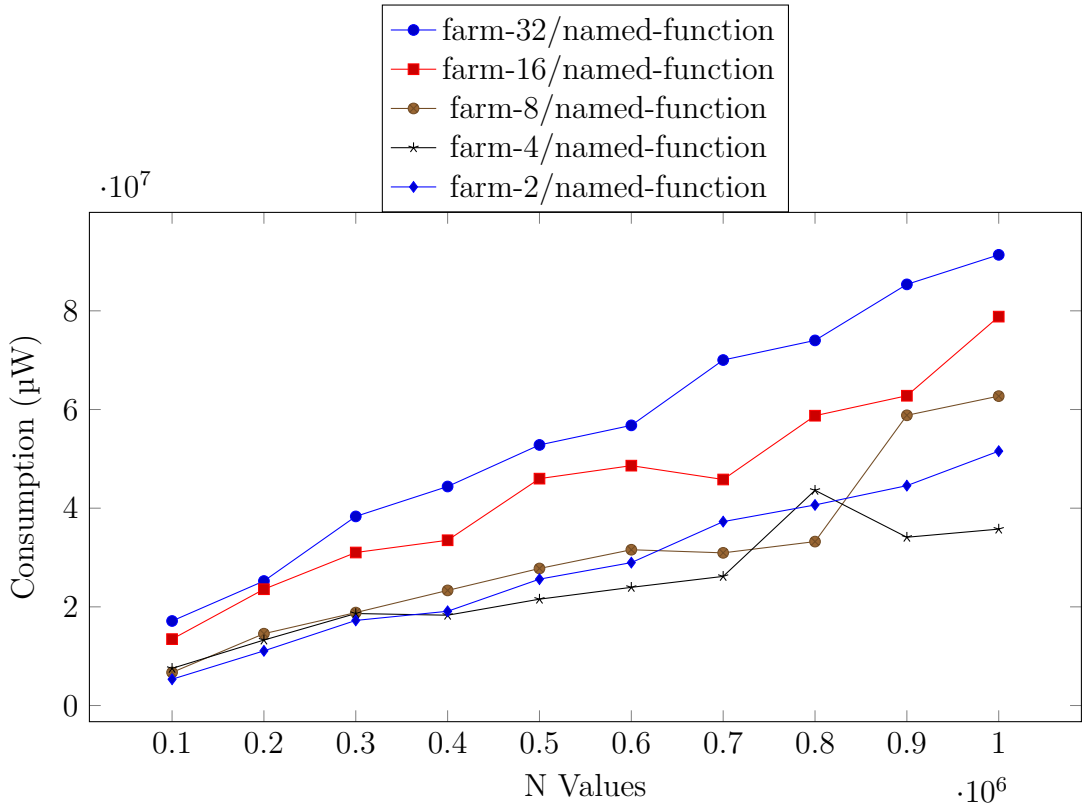


Figure A.39: Windows: Energy consumption values for different number of workers.

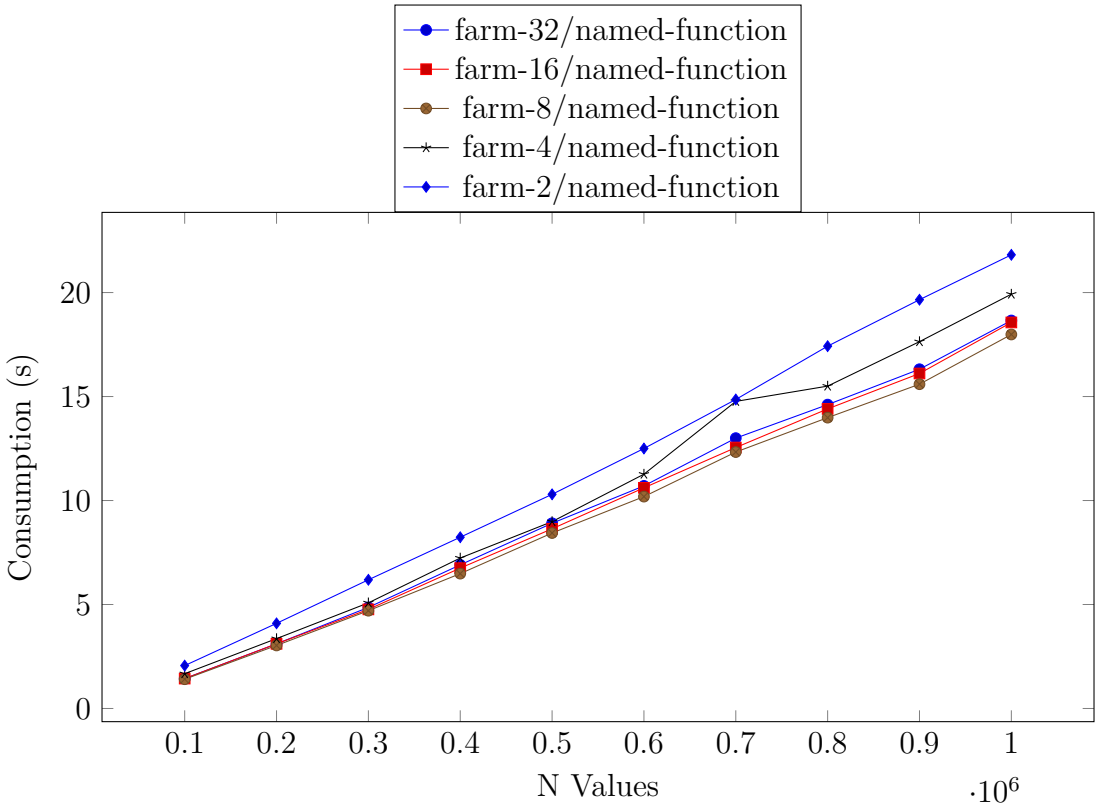


Figure A.40: Windows: Runtimes for different number of workers.

Measuring a more complex computation

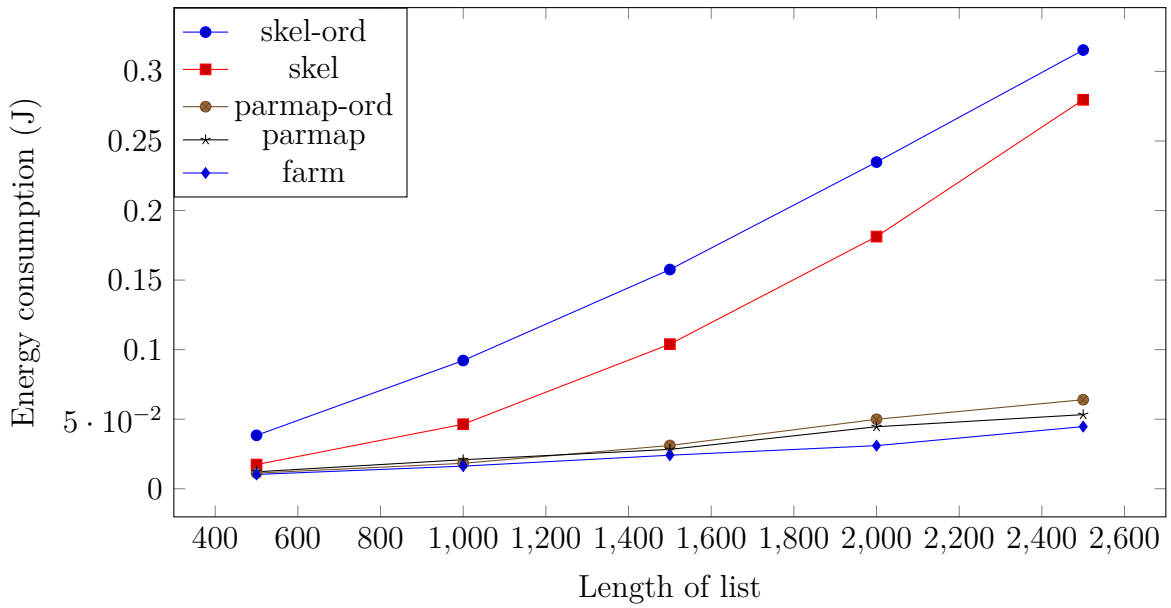


Figure A.41: Linux: Energy consumption of calculating Fibonacci 1.

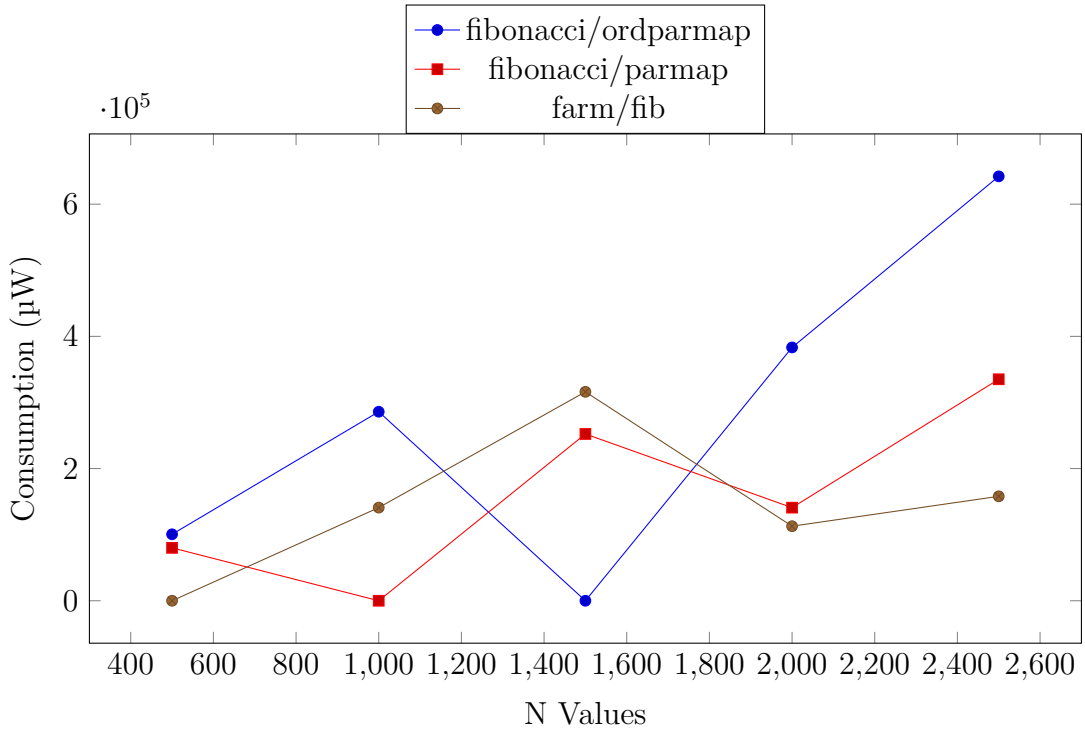


Figure A.42: Windows: Energy consumption of calculating Fibonacci 1.

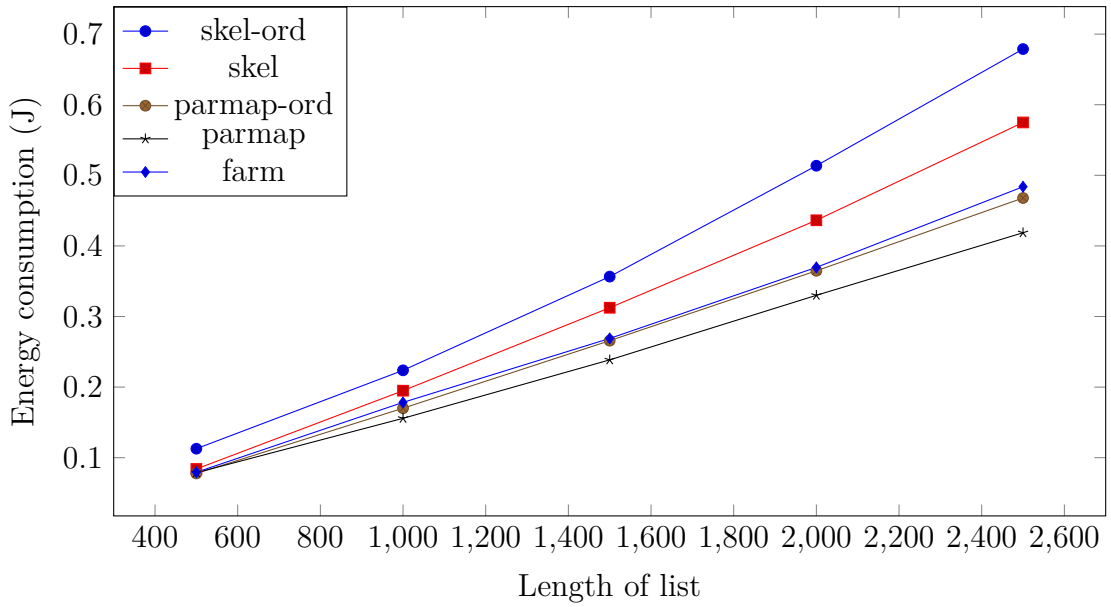


Figure A.43: Linux: Energy consumption of calculating Fibonacci 15.

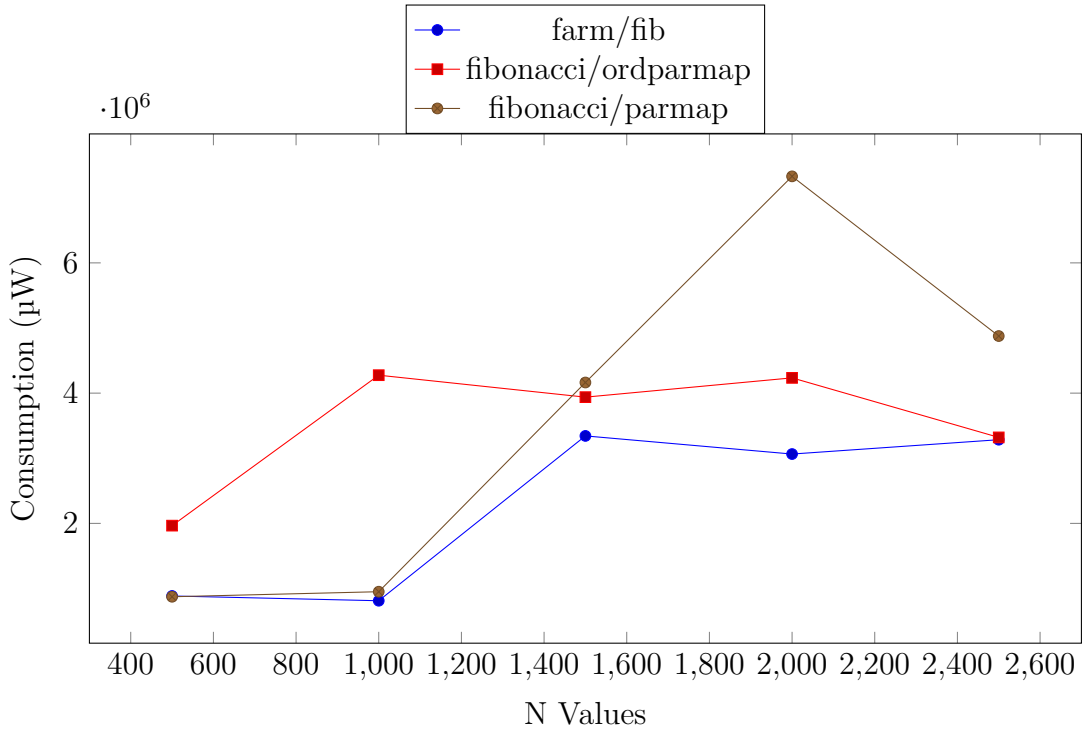


Figure A.44: Windows: Energy consumption of calculating Fibonacci 15.

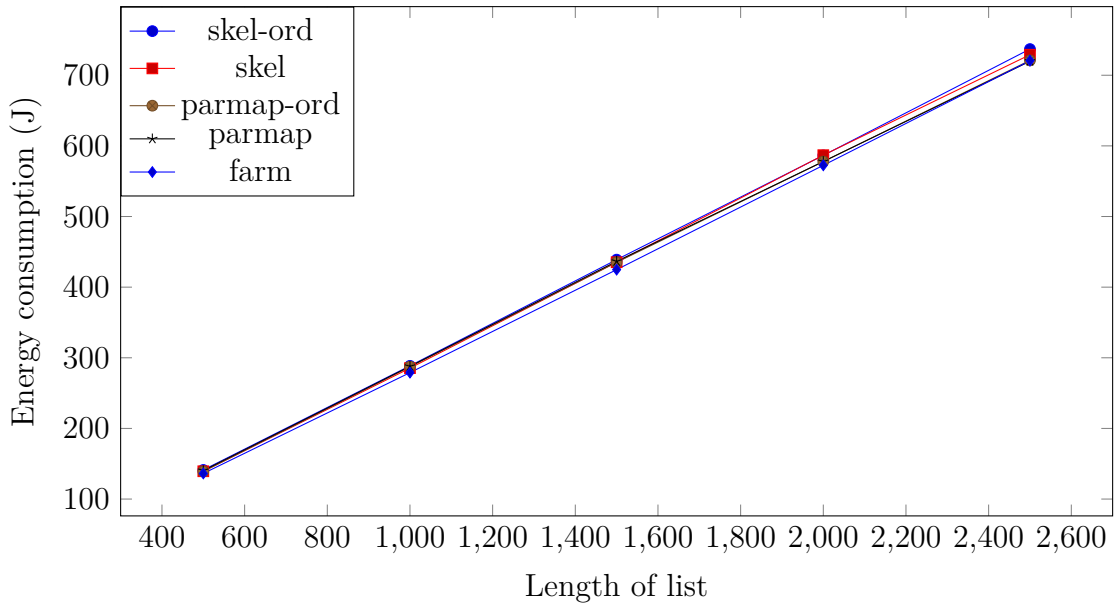


Figure A.45: Linux: Energy consumption of calculating Fibonacci 30.

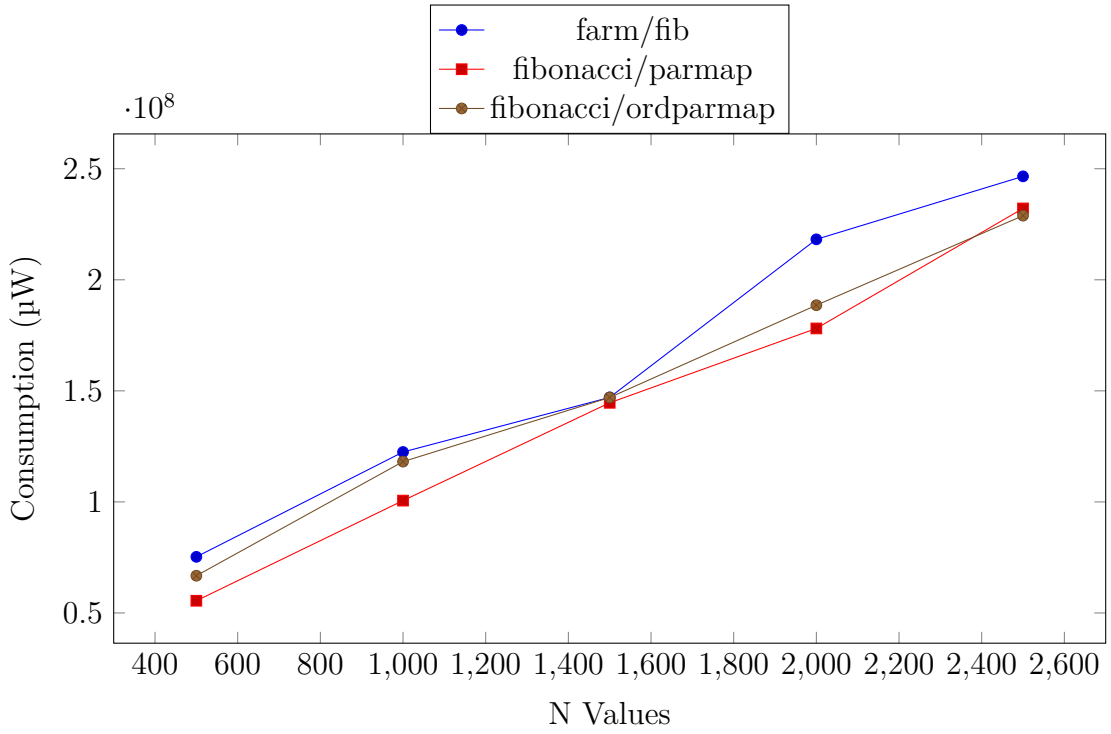


Figure A.46: Windows: Energy consumption of calculating Fibonacci 30.