# Green computing for Erlang*

Gharbi Youssef, István Bozó and Melinda Tóth

*ELTE, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary*

**Abstract**

Energy efficiency means to achieve the same result with less energy consumption. By reducing energy consumption, companies can potentially experience both cost savings and improved software performance. Erlang is heavily used in continuously operating server applications therefore the energy used by Erlang applications are quite significant. Any change, any reduction of the consumption of Erlang applications might have significant amount on the total energy used by the servers. This presents a measurement environment for Windows to be able to analyse the energy usage behaviour of Erlang applications. We used this environment to analyse various Erlang applications. We present our finding and compare the behaviour of the Erlang BEAM on Windows and Linux.

**Keywords**

energy consumption, Erlang, green computing, measurements

## 1. Introduction

Energy consumption is a major concern in every aspect of our life. Being energy efficient is a crucial factor of the modern manufacturing process [1]. 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 computers. For decades, people have been designing and producing ever-more energy-conscious hardware, but it has only recently became common practice to write energy-efficient software. Computers are being employed in more energy-critical systems as a result of their ubiquitous use and the Internet of Things (IoT).

Since environmental consciousness is growing in popularity [2], it is important to create software that is energy efficient. Cisco says that 90% of internet traffic goes through Erlang-controlled nodes [3]. With 5.07 billion internet users in the world growing by half a million daily [4], today comes the need to have efficient Erlang code patterns more than ever.

Our work aims to analyse the energy efficiency of Erlang applications by identifying language components that use less energy and to guide the developers to design and write more energy-efficient code. In our previous work [5, 6, 7] we created a toolchain to measure the energy consumption of Erlang language constructs and designed refactoring steps to transform the code to make it more efficient. The GreenErl framework focused on Linux-based measurements and analysis.

In this paper, we are presenting a framework [1] that is applicable to measure the energy consumption of Erlang software running on a Windows operating system. We built the framework on top of Scaphandre [8] and integrated it with the previous GreenErl framework [5, 9]. We repeated the measurements from the previous studies [10], analysed the results and compared the main findings with the Linux-based results.

This study aims to investigate the trends and behavior of Erlang/OTP on Windows and Linux as the size of input data structures increases. The following research questions guide our investigation:

1. Runtime Performance Trends: - What are the trends in the runtime performance of Erlang/OTP when processing increasingly larger data structures on both Windows and Linux?

2. Energy Consumption Comparison: - How does the energy consumption of Erlang/OTP on Windows compare to Linux as the size of input data structures increases?

Our objective is not to determine which operating system is optimal, but rather to observe and analyze how Erlang/OTP behaves and trends on each operating system under varying conditions. By addressing these questions, we aim to provide a comprehensive analysis of Erlang/OTP's performance and energy efficiency on different operating systems with increasing input sizes.

The rest of the paper is structured as follows. In Section 2 we present the context of our work including details on the original GreenErl framework and our investigations on selecting a proper measurement environment for Windows. Section 3 explains how the Scaphandre-based measurement environment was built for Windows. Section 4 describes our measurements and the analysis of the result. Finally, Sections 5 and 6 presents related works and concludes the paper.

## 2. Background

In this paper we would like to investigate the energy behaviour of Erlang applications. Therefore in this section we are summarising the main concepts and tools we rely on.

### 2.1. Green computing

Green computing is a study and practice of efficient and eco-friendly computing resources [11]. It involves developing, designing, engineering, producing, using, and disposing of computing modules and devices to reduce environmental hazards and pollution [12].

Green computing become increasingly important in recent years [13] as the world becomes more aware of the impact of technology on the environment [1, 14] and the utilization of mobile applications, embedded systems, and data center-based services increases [15, 16]. One way green computing can be achieved is through energy-efficient computing by designing computer systems that use less energy while still providing the same level of performance [17, 18]. Energy-efficient computing can be achieved through various

*3rd workshop on Resource AWareness of Systems and Society (RAW 2024)*

ikycue@inf.elte.hu (G. Youssef); bozoistvan@elte.hu (I. Bozó); tothmelinda@elte.hu (M. Tóth)

0009-0005-3183-0343 (G. Youssef); 0000-0001-5145-9688 (I. Bozó); 0000-0001-6300-7945 (M. Tóth)



[1] https://github.com/joegharbi/greenErl

methods such as using low-power processors [19], optimizing software code [20], and using virtualization technology [21]. Another way green computing can be achieved is through sustainable computing practices. This involves using environmentally friendly materials in the production of computer systems and reducing waste by recycling old computer systems [22, 23].

Green data centers are another aspect of green computing [24]. Data centers consume a large amount of energy due to their high-performance computing requirements. Green data centers aim to reduce this energy consumption 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 [25].

## 2.2. Erlang

Erlang [26] is a general-purpose, concurrent, functional, high-level programming language [27, 28] that was developed by Ericsson in the late 1980s. It is used to build massively scalable soft real-time systems with requirements of high availability [29, 30]. It can be used for a wide range of applications, some of its uses are in telecoms [31], banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system, the BEAM, has built-in support for concurrency, distribution and fault tolerance [30].

Erlang has several features that make it unique among programming languages and it also can serve as a runtime system [30]. 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.

Erlang is usually mentioned as Erlang/OTP. OTP is a set of Erlang libraries and design principles providing middleware 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, and debugging and release handling tools.

## 2.3. RAPL

Running Average Power Limit (RAPL) [32] presents in contemporary Intel and AMD processors, facilitating the measurement and regulation of power usage across various components such as CPU cores, memory, and the package. RAPL grants access to power and energy counters through model-specific registers (MSRs), which can be accessed by software applications.

In Linux environments, RAPL is widely utilized for energy-conscious tasks such as monitoring, profiling, and improving the energy efficiency of software and hardware. However, RAPL lacks native support in Windows systems [33], thereby constraining the utility of RAPL-based solutions for Windows users and developers. A project named Windows-RAPL-Driver [34] seeks to bridge this gap by furnishing a Windows driver enabling the gathering of RAPL metrics from physical computers. This driver exposes RAPL data via a device file accessible to user-space applications.

## 2.4. GreenErl framework for Linux

The former work on Erlang green computing [5, 9] made several important contributions, including the development of GreenErl, a tool designed to measure the energy consumption of Erlang programs. The GreenErl framework was developed using RAPL [35] to read the energy consumption. 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:

- **Rapl-read.c** is responsible for reading the energy consumption. The original source file was modified [32] to measure the energy consumption of Erlang functions. Now it splits 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.
- The **Erlang module**, `energy_consumption.erl`, 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.
- 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.

These components 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.

GreenErl was used [5] 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.

The effect of higher-order functions on energy consumption was also investigated and replacing higher-order function calls with either a list comprehension or a specialized recursive function was suggested to decrease energy consumption.

The energy cost of different parallel language constructs were also examined, 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, some refactorings were proposed 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 property list 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. The RefactorErl [36] framework was used to implement these transformations [5, 7].

### 2.5. Selecting the right environment for Windows

A significant challenge in reducing the energy usage of tech services is accurately and openly measuring it. However, most current tools and methods are either too expensive, too complicated, or not straightforward enough to be widely used by providers and users. For example, some tools need special hardware or software that does not work on all devices. Others use indirect estimates or averages that do not show the real power use of a particular service or device.

The GreenErl framework was developed using RAPL [35] to read the energy consumption from the designated registers. Since Windows does not have APIs that allow access to the needed model-specific registers [33], 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 do not want to redefine new function descriptions to have the highest similar environmental comparison. We also need to consider that some functions are 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 make the comparison.

We explored various Windows utilities, such as Powercfg, CPU-Z, Intel Power Gadget, and Scaphandre, to identify a tool that satisfies our specific requirements. These criteria included the ability to accurately measure energy consumption at the process level, maintain a rapid sampling rate, detect even low levels of energy consumption, and facilitate data storage in file (or other reusable, persistent storage) formats. Following this, we proceeded to adapt the GreenErl framework for compatibility with Windows, preserving common elements while integrating the chosen tool into the system. We then rigorously tested the framework and finalized the configuration parameters for optimal performance. Subsequently, we conducted the measurement procedures and carefully analysed the results obtained.

A recent open-source tool called Scaphandre [8, 37] endeavours to tackle these issues by offering a straightforward, dependable, and transparent approach to assessing and minimizing the energy consumption of technological services.

The fundamental principle behind Scaphandre's functionality lies in integrating two data sources: the proportion of resources utilized by a process and the total power consumption of the system. The latter relies on RAPL. We used Windows-RAPL-Driver [34] to provide the required functionality,

## 3. The Scaphandre based energy usage measuring tool-chain for Windows

Scaphandre presents numerous advantages over other watt meters concerning platform compatibility, measurement precision (with sampling rates down to one nano-second), and data output versatility (JSON format). It is operable on both Windows and Linux systems and compatible with processors supporting the RAPL interface. Additionally, Scaphandre can assess the power consumption of any process or application on both physical and virtual hosts. It offers the capability to transmit or expose power consumption data to various data analysis or monitoring tools for further processing and visualization, including JSON, Grafana, and wrap10. Consequently, Scaphandre emerges as a superior choice for power consumption measurement across diverse scenarios, including the context of this work, and holds promise for potential future enhancements.

To be able to run Scaphandre on Windows we needed to first install Windows-RAPL-Driver [34] to allow access to metrics and then Scaphandre [38][2].

Upon selecting Scaphandre as the designated tool, we proceeded to examine the pre-existing Unix-like OS GreenErl framework [5], experimenting with various implementations to determine the necessary edits and integration steps required. In this section, we will detail each component, outlining the adjustments made and ultimately presenting the finalized implementation outcome.

**GreenErl for Windows implementation** Our initial aim in adapting the GreenErl framework for Windows was to maintain consistent functionality and input specifications with the original Linux-based version. This entailed utilizing the existing Erlang modules as inputs for the Erlang measure module without necessitating modifications, wherever feasible, as the objective was to conduct a comparative analysis of energy consumption between the two operating systems. The process began with the removal of redundant functions and parameters from the Erlang measure module to create a streamlined command-line version of GreenErl tailored

[2]Throughout the installation process, we faced many challenges since the framework was in an early phase of development. For those who will try the framework in the future, there is new documentation and explanation of the Windows-RAPL-Driver installation on the official website [39] supporting Windows.

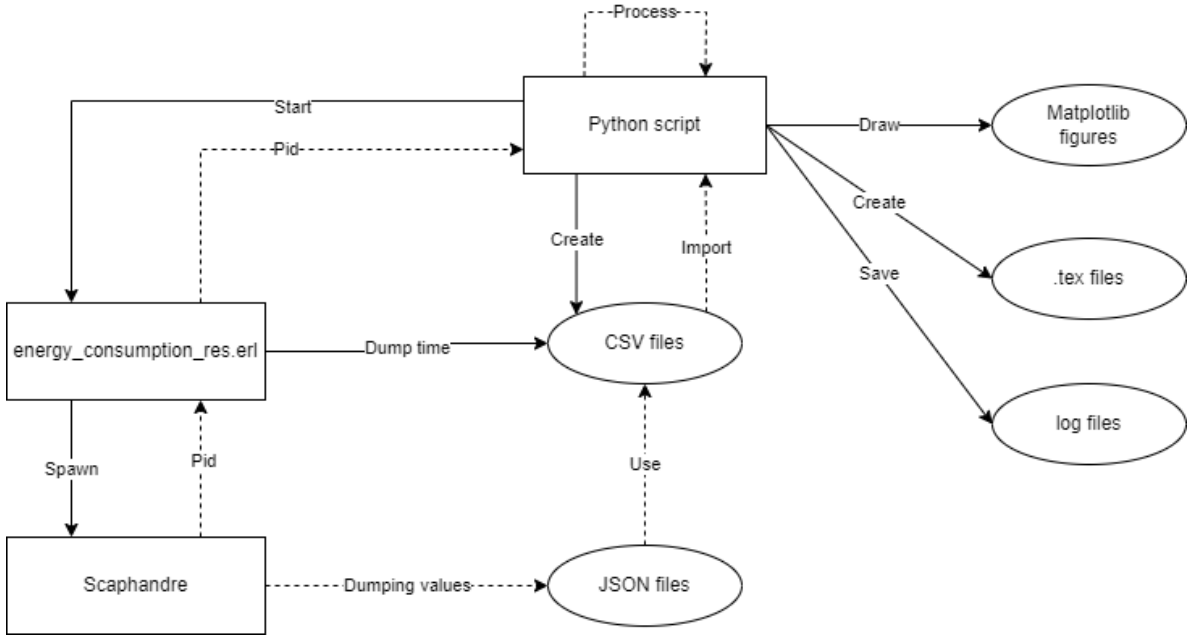


**Figure 1:** Workflow diagram of GreenErl on Windows

for Windows (without energy consumption data). Subsequently, integration of the Scaphandre component ensued, alongside modifications to the Python script to facilitate execution of the Erlang module. Finally, adjustments were made to the Python script to visualize and export results in latex format. The revised implementation thus comprised three key components: an Erlang module, a Python script, and a Scaphandre command for data collection.

**Erlang Module** The Erlang module, denoted as `energy_consumption_res.erl`, exposes a single function `measure/3`. This function receives three arguments: `{Module, Functions, InputDescs}`, `Count`, `ResultPath`. Its purpose is to measure the energy consumption across multiple functions, each executed with various input sets and repetitions. Employing recursive invocation, it iterates through the list of input descriptions and utilizes the `measureFunctions/3` function to assess each function with respective input argument lists. Additionally, it verifies whether the module under measurement features a function named `generate_input`, which, if available, generates input argument lists from provided descriptions. Otherwise, it treats the input descriptions directly as argument lists.

The input description follows the format `{Value, Count}`, where `Value` represents the input value and `Count` signifies the number of repetitions for that specific value within the function. We introduced this capability to operate optionally if included in the input description, recognizing that certain functions execute swiftly with minimal energy consumption (approximately 0 Watts). In cases where the pattern `{Value, Count}` is absent, the default `Count` specified in the `measure/3` parameter is utilized.

**Python script** Similar to the modifications made to the Erlang module, we streamlined the graphical user interface (GUI) components for Windows compatibility, as depicted in the measurement (Figure 2) and visualization (Figure 3) figures. Leveraging the *TkInter* library for Python, we developed an intuitive GUI allowing users to select the Erlang file containing the module for assessment, specify functions and inputs for measurement, and designate file paths for the Erlang module and result folders. Upon input completion, measurements can be queued for execution. Furthermore, we resolved path disparities between Windows and Linux, removed redundant packages, and updated the *LaTeX* exporter. A function was introduced to compute consumption values from JSON files, accepting arguments for folder path, count, input, and process ID (`Pid`) to ensure precise process-level consumption measurements. This function also handles errors, logging them if encountered. The visualization component (Figure 3) facilitates exporting *LaTeX* files and visualizing results, requiring only the path to CSV files or folders. The interaction between these components is illustrated in Figure 1.

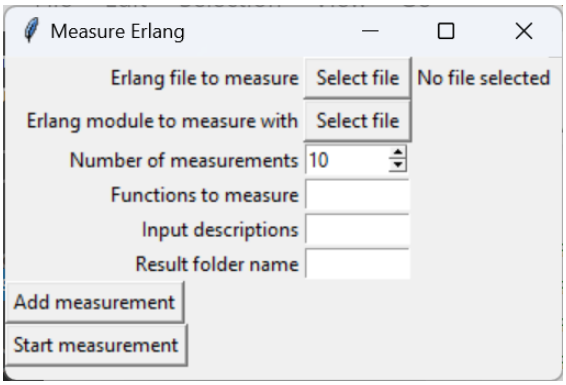


**Figure 2:** Python GUI to run the tool

**The use of Scaphandre** We integrated the Scaphandre tool into the Erlang measuring module. Tracking the module

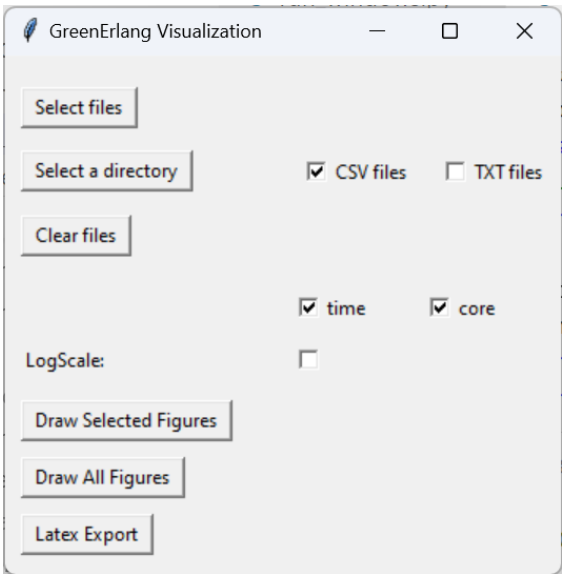


**Figure 3:** Python GUI to visualize

function and input descriptions was facilitated by specific names, as illustrated in Figure 4. This code segment handles the setup for generating the input JSON file name and constructing the Scaphandre command. To execute Scaphandre, we run the `wmic process call create` command from Windows Management Instrumentation (WMI) [40], and parse the output of the command to extract the PID of the newly spawned Scaphandre process. This PID is crucial for terminating the Scaphandre process once the function completes.

We utilize the JSON exporter feature offered by Scaphandre [41] to record the results into a JSON file. We specify the file path along with its name. For instance, as illustrated in Figure 5, given a module named `map`, a function named `recursive`, and an input description of `100,000`, the resulting JSON filename would be `map_recursive_100000.json`. Subsequently, this JSON file was employed by our Python script to compute the function's energy consumption.

Scaphandre offers the default option to collect the consumption data of the top 20 consumers. However, as depicted in Figure 5, we specified to gather data for the top 100 consumers explicitly. Through extensive experimentation with various consumer counts, we found that configuring it to 100 effectively covered all Erlang functions requiring measurement.

The final crucial feature for optimizing measurements and ensuring coherent results was determining the sampling rate. By default, Scaphandre sets this feature to two seconds [41], which proved inadequate for our requirements, particularly given the swift execution of some functions necessitating a larger sampling rate. After experimenting with various sampling rates, we opted for the highest available option, which is nano-second sampling, as depicted in Figure 5. Nano-second sampling provided a significantly higher granularity of detail in the generated graphs. While this increased granularity introduced potential fluctuations, we determined that the benefits outweighed the drawbacks, as it consistently yielded better results overall.

For experimental validation while determining the optimal configurations, we conducted two distinct implementations, each repeated 10 times. The consistent success observed across these iterations serves as compelling evidence of the framework's robustness and stability. This outcome confirms its readiness for future measurements, highlighting its reliability for continued use.

## 4. Analysing energy consumption

We performed various measurements on Windows to determine the energy behaviour of Erlang applications.

### 4.1. The measuring environment

The investigation into energy consumption across various functions was performed using the GreenErl framework for Windows, as outlined in Section 3. Each experiment was repeated 10 times, a choice made based on previous measurements that indicated a high level of consistency through analysis of standard deviation and range. Additionally, the runtime of the functions was measured. The data collected from these experiments were plotted and subjected to detailed descriptive statistical analysis.

All measurements in this research were performed on a DELL XPS 13 9370 equipped 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. We used Erlang/OTP 23.3.4 for the measurements. The previous Linux measurements were conducted on a Toshiba Satellite L50 equipped with an Intel Core i7-4700MQ @ 2.4 GHz, 12GB of RAM, running Ubuntu 16.04 LTS.

To ensure the accuracy and reliability of our results, we implemented several additional measures to create a controlled and consistent environment:

**Minimization of Processes** All non-essential processes and applications were terminated prior to conducting the measurements to eliminate any background activities that could introduce noise and affect the performance analysis.

**Hardware Consistency** The measurements were conducted on a DELL XPS 13 9370 with the following specifications: Intel CoreTM i5-8250U CPU @ 1.60GHz 1.80 GHz, 8GB of DDR3 RAM, 64-bit operating system, x64-based processor. The Windows environment ran Windows 11 Pro Version 22H2. Erlang/OTP 23.3.4 was used for the measurements.

**Operating System Services** Only essential operating system services were active during the measurements to avoid any extraneous influence on the results.

**Network and Connectivity** The WiFi was turned off, and the laptop was set to airplane mode during all measurements to prevent network-related processes from affecting the results.

**System Initialization** Each measurement was initiated on a fresh startup to ensure a clean state, free from any residual processes or temporary files that could impact the results.

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

**Figure 4:** A code snippet from the Erlang measure module

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

**Figure 5:** JSON export example

**Power Settings** The power settings were configured to 'balanced' mode to provide a consistent energy consumption environment without favoring performance or power saving excessively.

**Battery and Power Management** The laptop was fully charged and plugged into the AC during the measurements. Screen brightness was set to the minimum level. The laptop was set to test mode as required by Scaphandre, and the screen and sleeping features were disabled both on the battery and when plugged in.

**User Interaction** The laptop was not used during the measurements to avoid any additional load or interference.

By implementing these measures, we aimed to create a controlled and consistent environment for conducting the performance analysis, thereby enhancing the reliability of our findings.

These detailed conditions ensured that our measurements were conducted under minimal interference, providing a clear and accurate representation of the software programs' energy consumption and performance.

## 4.2. Types of measurements

The preceding study [5] conducted an original investigation into the energy consumption of complex algorithms in Erlang. Specifically, it compared different implementations of the N-queens problem and sparse matrix multiplication [6], employing various techniques including higher-order functions, recursion, list comprehensions, lists, and arrays. Additionally, other aspects of the research [5] focused on more fundamental language elements rather than solely addressing complex problems.

In this measurement, we focus on comparing different implementations of the same language element. Our detailed results can be found in this thesis [10], as it is not feasible to include all figures here. Additionally, all GreenErl for Windows framework components can be found in the same repository [3]. Based on the previous findings, we selected the following areas for this analysis:

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

**Data structures** The objective of this study was to evaluate the energy efficiency of various data structures for storing key-value pairs on Windows. We 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, we 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 we 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

**Finding an element** Among the observations, we found that map implementations exhibited greater efficiency and faster execution times. In contrast, all list implementations consumed more energy and took longer to complete. This finding is illustrated in the Figure 6.

**Higher-order functions** We also investigated the influence of higher-order functions (HOFs) on energy consumption. Based on the findings shown in Figure 7, both named function and lambda expression, exhibit consistently elevated energy consumption. Conversely, using list comprehension proves to be relatively more efficient, regardless of whether a named function or lambda expression is utilized.

**Parallel language constructs** Erlang effortlessly supports extensive concurrency through its simple primitives for process creation and communication. Each process operates within its own isolated memory space, exclusively holding its data. To share data between processes, it must be transmitted as messages. This research measures not only the energy consumption associated with creating, modifying, and converting various data structures but also the

[3] https://github.com/joegharbi/greenErl

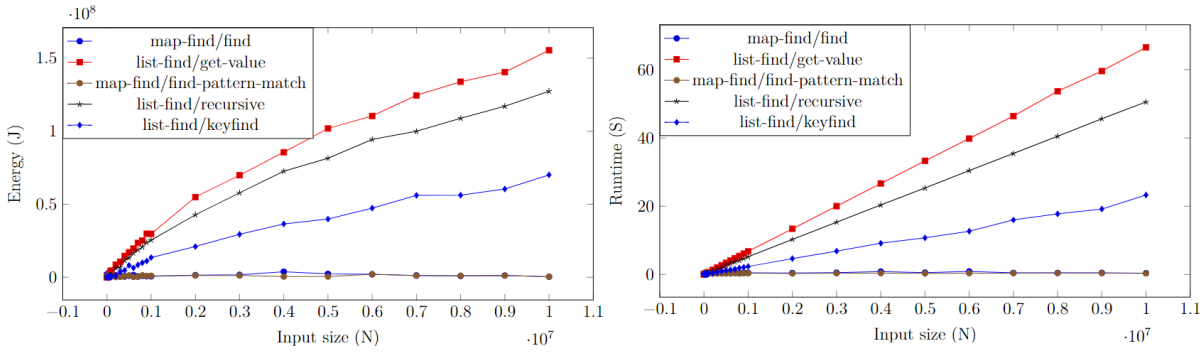


**Figure 6:** Finding an element using GreenErl on Windows

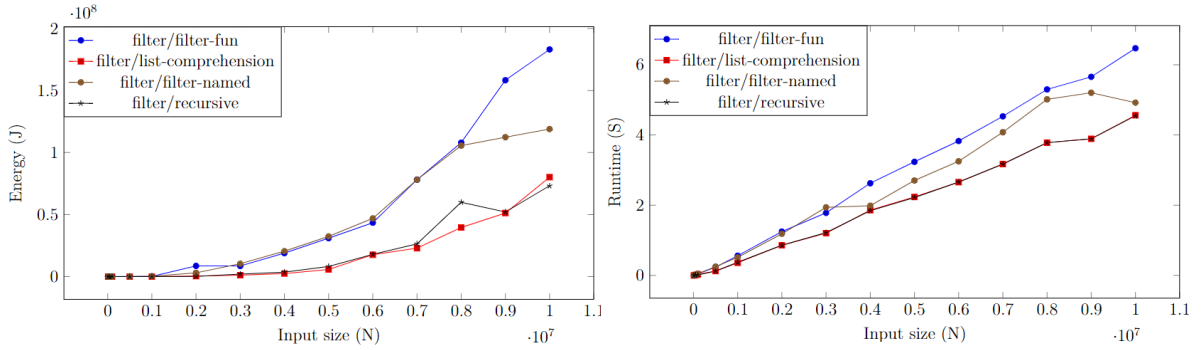


**Figure 7:** Higher-order functions using GreenErl on Windows

overhead of sending these structures between processes. We aimed to determine whether optimizing data representation before transmission and upon reception could minimize energy consumption.

We analysed the `map/send` function initiates a process that interacts with the main process by exchanging a list of key-value pairs along with a confirmation message. Similarly, the `list/send` and `dictionary/send` functions communicate a list and a dictionary, respectively. We found that sending a list of tuples proves to be the least efficient in terms of energy consumption. Conversely, sending maps demonstrates the highest efficiency, followed by the dictionary (Figures 8 and 9).

**Algorithmic skeletons** This study aimed to understand how different parallel algorithmic skeletons impact energy consumption. These parallel programming patterns are extensively used due to their ability to simplify the complexities of parallel and distributed applications, highlighting the importance of investigating their energy usage implications. In this research, we evaluated fundamental skeletons such as farm and pipeline skeletons. Additionally, considering the potential combinations of these patterns, we also assessed various compositions of skeletons.

For instance, the task farm is a widely used parallel programming pattern to parallelise the evaluation of a function for all elements of an input structure. To implement it, three key members with distinct roles are defined. The dispatcher is responsible for managing the input and distributing elements to the workers. Each worker executes the function on assigned inputs and forwards the results to the collector before requesting a new element. Finally, the collector aggregates the results. It is important to note that this skeleton does not maintain the order of the elements.

## 4.3. Comparing energy consumption

We aimed to compare the energy behaviour of Erlang applications on Linux and Windows operating systems. Therefore, we will explore some of the trends between the two operating systems.

**Data structures** A noticeable distinction was observed between Linux and Windows environments when constructing a dictionary as shown in Figure 10. Linux exhibited a peak in energy consumption compared to other data structures, whereas Windows demonstrated a much lower peak. Interestingly, the remaining functions displayed consistent behaviour across both operating systems.

On the other hand, we observed that most functions exhibited similar behaviour. For instance, when deleting an element from a list, as illustrated in Figure 11.

**Inference** The Linux kernel for servers has been specifically optimized for high throughput and performance [42], prioritizing processing efficiency over power consumption. Consequently, Linux distributions tend to consume more power and battery life compared to Windows and Mac operating systems [43]. Although the aim of this investigation is not to prove which Operating System is more efficient as

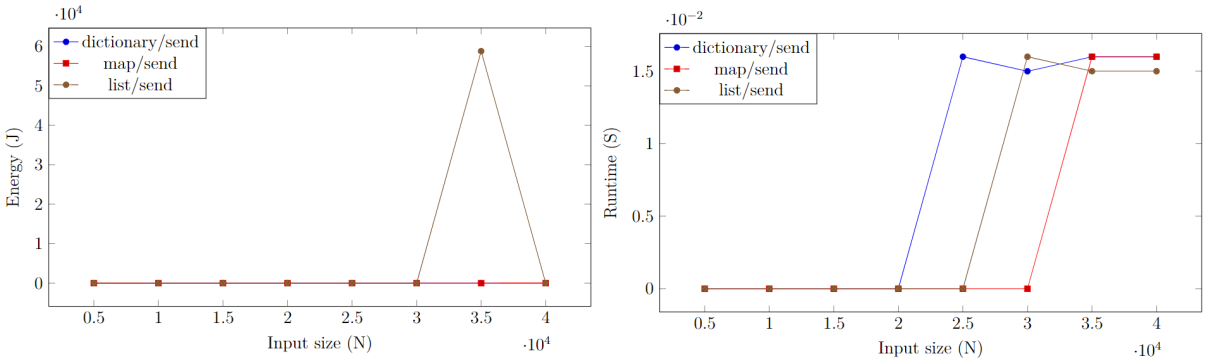


**Figure 8:** Sending of elements from a container

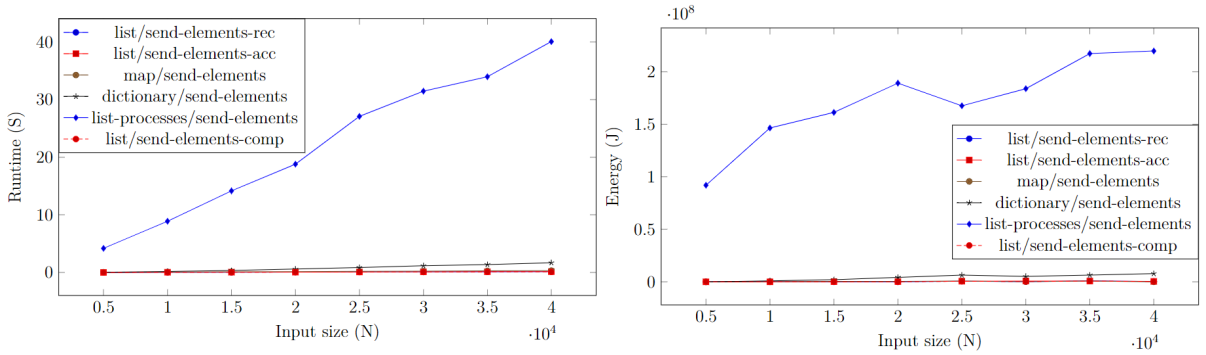


**Figure 9:** Sending of elements from a container

stated in the introduction, nevertheless our findings showed that Windows behaved in a more efficient way than Linux.

As previously mentioned, comprehensive results are documented in this thesis [10].

## 5. Related work

We are investigating the related works from different perspectives. We are summarising the studies related to functional programming and related to comparisons of operating systems and programming languages.

**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 [44] 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

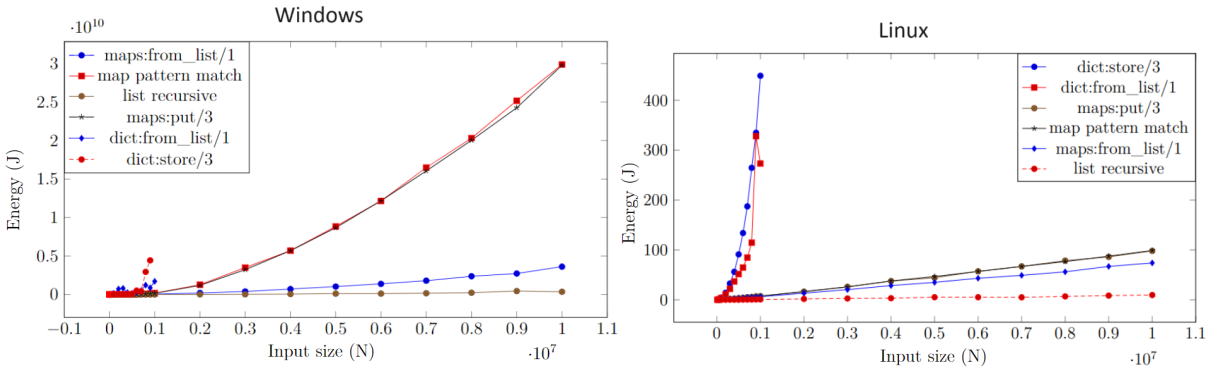


**Figure 10:** Different behavior of dictionary on Linux and Windows

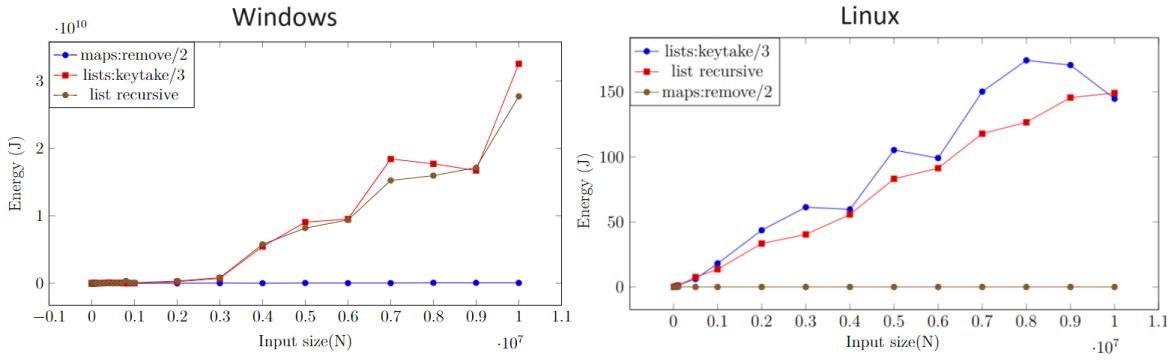


**Figure 11:** Different behavior of deleting an element on Linux and Windows

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 [45] 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 [46] 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 [47] 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.

**Programming languages efficiency** Rui Pereira et al [48] 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 we found in this paper, 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 [5].

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 [6] 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 parallelisation solutions can result in the worst consumption due to the fact of spawning different processes which

was the case in our results mentioned in this paper and the thesis containing the details of the work [10].

More recent research that incorporates hardware measurement setups to evaluate energy consumption is also noteworthy. Koedijk and Oprescu's 2022 study, "Finding significant differences in the energy consumption when comparing programming languages and programs," [49] presented at the International Conference on ICT for Sustainability (ICT4S), identified substantial variations in energy usage across different programming languages. Their results indicated that certain languages, such as C and Rust, tend to be more energy-efficient compared to higher-level languages like Python and JavaScript. They also found significant variations in energy consumption based on language choice and programming techniques. While C and C++ consistently exhibit lower energy consumption across the tested programs, they note that energy usage can vary depending on compiler optimization settings. This study's findings provide valuable insights and a more detailed understanding of the energy efficiency landscape, underscoring the importance of considering hardware-level data and compiler optimizations in energy consumption analysis for software development.

**Functional languages** Luis Gabriel Lima et al [50, 51]. conducted studies specifically targeting the energy efficiency of functional programming languages, focusing on Haskell. Their research, "On Haskell and energy efficiency" [50], and "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language" [51], examined Haskell's energy consumption characteristics. They evaluated various Haskell programs to understand the impact of different language features on energy usage. The studies revealed that Haskell, due to its purely functional nature, often exhibited favorable energy consumption patterns compared to other programming languages. They also found that small changes can make a big difference in terms of energy consumption. These findings underscore the importance of considering functional languages like Haskell when analyzing energy efficiency in software development.

## 6. Conclusion

Green computing is important for all kinds of systems, from handheld devices to large-scale data centres. It can help reduce greenhouse gas emissions 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 of our research team. Previously, the work was only focused on Unix-like operating systems, thus this thesis was a first step into bridging the gap between these different platforms.

As a first step, we started by looking for a tool to fill the gap between the two operating systems since RAPL is not available in Windows. Thus we used Scaphandre since it checked all of the boxes for our constraints. We integrated the tool with the previous GreenErl framework and measured different Erlang functions.

There were some nice findings about Erlang on Windows. We measured the energy consumption and runtime of some data structure implementations such as lists, maps and dictionaries. We evaluated creating, converting, updating, searching and deleting elements from those data structures where we 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. We 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, We also measured sending data in different forms. Lastly, we assessed some algorithmic skeletons such as task farms where we concluded it is better to spawn as many workers as the number of your physical processes.

As for the comparison section, we 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 behaviour while being the highest consumer in Linux except for 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 [52] and Scaphandre.

**Future work** As Scaphandre is still in an early stage, we 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.

## Acknowledgments

This work is partially supported by CERCIRAS COST Action CA19135 funded by COST Association.

Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

## References

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

[2] A. A. Chien, Computing's grand challenge for sustainability, Commun. ACM 65 (2022) 5. URL: https://doi.org/10.1145/3559163. doi:10.1145/3559163.

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

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

[5] G. N. 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.

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

[7] G. Nagy, A. A. Mészáros, I. Bozó, M. Tóth, Tools supporting green computing in erlang, in: Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Erlang 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 30–35. URL: https://doi.org/10.1145/3331542.3342570. doi:10.1145/3331542.3342570.

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

[9] J. T. C. Ortiz, Green computing in Erlang, Master's thesis, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, 2017.

[10] Y. Gharbi, Green computing for Erlang, Master's thesis, Eötvös Loránd University, Budapest, Hungary (Faculty of Informatics, Department of Programming Languages and Compilers), 2023. URL: https://github.com/joegharbi/greenErl/blob/main/papers/YoussefGharbi_IKYCUE_THESIS.pdf.

[11] I. Ray, Green computing, 2012. URL: https://www.researchgate.net/publication/270570843_Green_Computing.

[12] B. Saha, Green computing, International Journal of Computer Trends and Technology (IJCTT) 14 (2014) 46–50. doi:10.14445/22312803/IJCTT-V14P112.

[13] L. Ardito, G. Procaccianti, M. Torchiano, A. Vetrò, Understanding green software development: A conceptual framework, IT Professional 17 (2015) 44–50. doi:10.1109/MITP.2015.16.

[14] S. Podder, A. Burden, S. Kumar Singh, R. Maruca, How green is your software?, 2020. URL: https://hbr.org/2020/09/how-green-is-your-software.

[15] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An empirical study of practitioners' perspectives on green software engineering, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 237–248. URL: https://doi.org/10.1145/2884781.2884810. doi:10.1145/2884781.2884810.

[16] N. Gholipour, E. Arianyan, R. Buyya, Recent Advances in Energy Efficient Resource Management Techniques in Cloud Computing Environments, Springer, 2021, p. 29.

[17] B. S, Predicate path expressions, in: Proceedings of the UGC Sponsored National Conference on Advanced Networking and Applications, Advanced Networking and Applications (IJANA), 2015, pp. 107–112. URL: https://www.ijana.in/Special%20Issue/file24.pdf.

[18] A. Kaur, D. Gupta, D. Verma, Making cloud computing more efficient, International Journal of Advanced Research in Computer Science and Software Engineering 4 (2014).

[19] J. Shamir, Energy efficient computing exploiting the properties of light, in: J. Ojeda-Castaneda, M. J. Yzuel, R. B. Johnson (Eds.), Tribute to H. John Caulfield, volume 8833, International Society for Optics and Photonics, SPIE, 2013. URL: https://doi.org/10.1117/12.2024487. doi:10.1117/12.2024487.

[20] G. Konduri, J. Goodman, A. Chandrakasan, Energy efficient software through dynamic voltage scheduling, in: 1999 IEEE International Symposium on Circuits and Systems (ISCAS), volume 1, 1999, pp. 358–361 vol.1. doi:10.1109/ISCAS.1999.777877.

[21] H. Castro, G. Sotelo, C. O. Diaz, P. Bouvry, Green flexible opportunistic computing with virtualization, in: 2011 IEEE 11th International Conference on Computer and Information Technology, 2011, pp. 629–634. doi:10.1109/CIT.2011.105.

[22] T. V. Kumar, P. Kiruthiga, Green computing - an eco friendly approach for energy efficiency and minimizing e-waste 3 (2014) 356–359. URL: https://www.academia.edu/6943675/Green_Computing_An_Eco_friendly_Approach_for_Energy_Efficiency_and_Minimizing_E_Waste.

[23] L. Goldberg, The advent of "green" computer design, Computer 31 (1998) 16–19. doi:10.1109/2.708445.

[24] D. Çavdar, F. Alagoz, A survey of research on greening data centers, in: 2012 IEEE Global Communications Conference (GLOBECOM), 2012, pp. 3237–3242. doi:10.1109/GLOCOM.2012.6503613.

[25] J. Torres, D. Carrera, K. Hogan, R. Gavalda, V. Beltran, N. Poggi, Reducing wasted resources to help achieve green data centers, 2008 IEEE International Symposium on Parallel and Distributed Processing (2008). URL: https://ieeexplore.ieee.org/abstract/document/4536219. doi:10.1109/ipdps.2008.4536219.

[26] F. Cesarini, S. Thompson, Erlang programming, O'Reilly, 2009. URL: http://shop.oreilly.com/product/9780596518189.do.

[27] F. Cesarini, S. J. Thompson, Erlang Programming, O'Reilly Media, Inc., United States of America, 2009.

[28] J. Armstrong, Programming Erlang: Software for a Concurrent World, The Pragmatic Bookshelf, United States of America, 1993.

[29] J. Armstrong, The development of erlang, in: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97, Association for Computing Machinery, New York, NY, USA, 1997, p. 196–203. URL: https://doi.org/10.1145/258948.258967. doi:10.1145/258948.258967.

[30] J. Armstrong, A history of erlang, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, Association for Computing Machinery, New York, NY, USA, 2007, p. 6–1–6–26. URL: https://doi.org/10.1145/1238844.1238850. doi:10.1145/1238844.1238850.

[31] J. Armstrong, S. Virding, Erlang - an experimental telephony programming language, in: International Symposium on Switching, volume 3, 1990, pp. 43–48. doi:10.1109/ISS.1990.765711.

[32] V. M. Weaver, Reading RAPL energy measurements from Linux, http://web.eece.maine.edu/~vweaver/projects/rapl/, 2015.

[33] tools/power/rapl — firefox source docs documentation, 2023. URL: https://firefox-source-docs.mozilla.org/performance/tools_power_rapl.html#windows.

[34] Hubblo, Windows driver to get RAPL metrics, 2023. URL: https://github.com/hubblo-org/windows-rapl-driver.

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

[36] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, T. M., M. Tóth, Refactorerl - source code analysis and refactoring in erlang, in: Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2, Tallin, Estonia, 2011, pp. 138–148.

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

[38] Hubblo, Scaphandre: Energy consumption metrology agent, 2023. URL: https://github.com/hubblo-org/scaphandre.

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

[40] WMI command-line (WMIC) utility - Win32 apps, 2023. URL: https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmic.

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

[42] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, Energy-aware task and interrupt management in linux, in: Proceedings of the Linux Symposium, volume 2, 2008. URL: http://www.cs.columbia.edu/~nahum/w6998/papers/ols-2008-srinivasan-energy.pdf.

[43] S. Najmuddin, Z. Atal, R. A. Ziar, Comparative analysis of power consumption of the linux and its distribution operating systems vs windows and mac operating systems., Kardan journal of engineering and technology (2021). URL: https://kardan.edu.af/data/public/files/KJET-%203.1-2021-06%20Linux%20Kernel19012022120259.pdf. doi:10.31841/KJET.2021.21.

[44] H. T. Al-Rayes, Studying main differences between linux & windows operating systems, IJECS: International Journal of Electrical and Computer Sciences 12 (2012) 25–31.

[45] B. Prieto, J. J. Escobar, J. C. Gómez-López, A. F. Díaz, T. Lampert, Energy efficiency of personal computers: A comparative analysis, Sustainability 14 (2022) 12829. URL: https://www.mdpi.com/2071-1050/14/19/12829. doi:10.3390/su141912829.

[46] Top500, 2013. URL: https://www.top500.org/.

[47] S. Najmuddin, Z. Atal, R. A. Ziar, Comparative analysis of power consumption of the linux and its distribution operating systems vs windows..., 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.

[48] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. 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.

[49] L. Koedijk, A. Oprescu, Finding significant differences in the energy consumption when comparing programming languages and programs, in: 2022 International Conference on ICT for Sustainability (ICT4S), 2022, pp. 1–12. doi:10.1109/ICT4S55073.2022.00012.

[50] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, J. P. Fernandes, On haskell and energy efficiency, Journal of Systems and Software 149 (2019) 554–580. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302747. doi:https://doi.org/10.1016/j.jss.2018.12.014.

[51] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, J. P. Fernandes, Haskell in green land: Analyzing the energy behavior of a purely functional language, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, 2016, pp. 517–528. doi:10.1109/SANER.2016.85.

[52] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, Z. Ou, Rapl in action: Experiences in using rapl for power measurements, ACM Trans. Model. Perform. Eval. Comput. Syst. 3 (2018). doi:10.1145/3177754.