sustainability

MDPI

Article

Design of a Standard and Programmatically Accessible Interface for Smart Meters to Allow Monitoring Automation of the Energy Consumed by the Execution of Computer Software

Alberto Ortega *, Abel Miguel Cano-Delgado, Beatriz Prieto and Jesús González

Department of Computer Engineering, Automation and Robotics, CITIC, University of Granada, 18010 Granada, Spain
* Correspondence: aoruiz@ugr.es

Abstract: Software has become more computationally demanding nowadays, turning out high-performance software in many cases, implying higher energy and economic expenditure. Indeed, many studies have arisen within the IT community to mitigate the environmental impact of software. Collecting and measuring software's power consumption has become an essential task. This paper proposes the design of a standard interface for any currently available smart meter, which is programmatically accessible from any software application and can collect consumption data transparently while a program is executed. This interface is structured into two layers. The former is a driver that provides an OS-level standard interface to the meter, while the latter is a proxy offering higher-level API for a concrete programming language. This design provides many benefits. It makes it possible to substitute the meter for a different device without affecting the proxy layer. It also allows the presence of multiple proxy implementations to offer a programmatic interface to the meter for several languages. A prototype of the proposed interface design has been implemented for a concrete smart meter and OS to demonstrate its feasibility. It has been tested with two experiments. Firstly, its correct functioning has been validated. Later, the prototype has been applied to monitor the execution of a high-performance program, a machine learning application to select the most relevant features of electroencephalogram data.

Keywords: energy metering system; software power consumption

Citation: Ortega, A.; Cano-Delgado, A.M.; Prieto, B.; González, J. Design of a Standard and Programmatically Accessible Interface for Smart Meters to Allow Monitoring Automation of the Energy Consumed by the Execution of Computer Software. *Sustainability* **2023**, *15*, 1900. https://doi.org/10.3390/su15031900

Academic Editor: Ke Xing and Bin Huang

Received: 22 December 2022
Revised: 13 January 2023
Accepted: 16 January 2023
Published: 19 January 2023

1. Motivation

Gone are the days when computers were used for a single purpose and existed solely for scientific, government, or banking applications. Today, there is at least one computer in any home, which can be used for a wide variety of applications, such as office suites, image processing solutions, video editing software, video broadcasting, or playing video games, to name some of the most common ones.

Many of the above programs are executed daily on personal computers, either as part of a profession or as part of a hobby. On the one hand, these everyday tasks can usually become quite intensive for the computer. On the other hand, the advancement of programming techniques in recent years, along with the development of more powerful computers, has resulted in more advanced, heavier, and resource-intensive software.

These trends can lead to an increase in the power consumption of computers and, taking into account the enormous number of personal computers in the world, their overall energy consumption is prone to grow too. In short: more intensive programs, executed on more powerful computers, which consume more energy, increase overall energy consumption. In fact, Andrae et al. predict in [1] that, in a worst-case scenario, information and communication technologies could consume up to 51% of the world's electricity by 2030 and contribute 23% of greenhouse gas production.

Therefore, the consumption caused by software is worth measuring and analyzing. Mainly, three approaches can be applied to obtain consumed energy, either for the execution of a program or for other scenarios [2–4]: estimating the consumption through a customized model, aggregating the power consumed by the different parts of a system, or metering the consumption of the whole system globally, with the latter being widely used to measure the power consumed by personal computers [5–15]. However, the collection of energy measurements for this approach becomes cumbersome since although device manufacturers provide human-friendly interfaces, neither of them meets any standard nor allows programmatic access to the meter, which makes it difficult or even impossible for programs to be aware of their consumed power while being executed [16]. This paper aims to design such an interface to allow any program to access its current energy consumption transparently, regardless of the smart meter connected between the computer and the wall outlet.

The rest of the paper is organized as follows: Section 2 presents the most relevant antecedents and current developments related to this work. Later, Section 3 describes the design of the proposed standard and programmatically accessible interface for smart meters. Section 4 details a prototype implementation of this proposal for a concrete OS and measurement device, a Linux OS, and an openZmeter, respectively. Next, Section 5 validates the interface prototype implementation, and finally, some conclusions and a future work proposal are drawn in Section 6.

2. State-of-the-Art

The energy consumed by any program can be estimated through a mathematical model specifically designed to calculate the power consumption derived from its execution on a concrete computer system. This kind of model must be designed specifically for each program, and their parameters (both program- and computer-related) must be adjusted using experimental measurements acquired through many executions. However, this approach presents one significant drawback. The slight energy variations caused by circumstances beyond the execution of a program are hard to estimate and, therefore, the consumption they calculate may be inaccurate. For example, the power consumption caused by the processor cooling system may be estimated by the maximum value of this parameter provided by the computer's constructor. Nevertheless, since the cooling system adapts its operation to the processor's temperature, which may vary during the program's execution, its consumption will be overestimated. Thus, it is usually assumed that each computer subsystem will produce a linear power consumption regarding its utilization. Notwithstanding, the aging of the cooling system's fan usually produces it slight deformations, which probably will increase its power consumption, motivating the use of more powerful techniques, such as machine learning [17]. Therefore, estimation models are commonly used for large computer systems, such as datacenters [18,19], rather than personal computers, because measuring their actual overall consumption is more complicated than estimating it with a reasonable error rate. Surveys on energy estimation models have also been reported in other fields, such as on GPUs [20], multicore processors [21], mobile devices [22,23], HPC systems [24], or for the execution of machine learning algorithms [16,17,25].

The direct metering and sub-metering approaches avoid the inaccuracy of energy estimation by measuring the actual system consumption. Sub-metering consists of measuring the consumption of the different computer subsystems (such as the CPU, GPU, memory, disk, network interface devices, power conversion, etc.) and then expressing the system's power consumption as the aggregation of the power drawn by its subcomponents. Separate metering hardware can be used to collect the consumption data of the different computer subsystems, including devices such as current sensors, current clamps, data acquisition cards, and microcontrollers. Measurements can be performed, for example, on the different output lines from the system's DC power supply connected to distinct parts of the computing system (motherboard, disk, etc.). However, it is not physically possible to obtain direct measurements of consumption of all the low-level components (inside the chips),

although it is possible to use counters and hardware registers that are included as utilities or interfaces by the processor manufacturers for thermal and power management to obtain indirect measurements of such components [26,27]. With this type of interface, it is possible, for example, to develop tools to control the operation (on/off) of fans or to monitor power consumption. The RAPL (running average power limit) interface, introduced by Intel in their Sandy Bridge processor architecture [28], is an example of such an interface. Notwithstanding, sub-metering presents some drawbacks too. Firstly, some measurements can only be obtained indirectly through specific interfaces for each subsystem, not being as accurate as actual measures obtained directly. Secondly, the metering infrastructure connected to the different computer subsystems may introduce some overhead. Thirdly, monitoring may not be possible directly or indirectly for some components of the computer. Thus, the aggregation of all the gathered measurements will result in an inaccurate estimation of the actual energy consumption caused by the program execution. Despite the above, the separate sub-meters allow extraction and integration of information from different subsystems to achieve a unified, real-time, and intelligent view of the responsibility of the distinct elements in the overall consumption [21,27].

Finally, the direct or global metering approach involves using an external energy measurement device, connected in series with the computer power cable to the wall outlet, to obtain an accurate measurement of the actual energy consumed by the computer while executing a program. Different commercial measuring devices exist, along with many other custom approaches based on microcontrollers [6–12,16,25,29]. Direct metering is indeed used for the realization of the Green500 supercomputer ranking [30,31]. An example of a direct metering approach is proposed in [13], where the actual power consumption data of some algorithms executed on a Raspberry Pi are gathered through a NI USB-6210 DAS (data acquisition system), which provides accurate measures of the power and consumption over time. There are also more approaches based on using a DAS to obtain the energy measurements of running programs [14,15], but suffering all of them from the same drawbacks: DASs are rather expensive devices and can be accessed only through the interface provided by their manufacturer.

Using a conventional and cheaper energy metering device may ease energy consumption studies, although the manufacturer's interface may not always be versatile enough to perform downstream data analyses conveniently. For example, Prieto et al. use openZmeter (oZm) [32,33], an inexpensive and off-the-shelf energy metering device, to perform a computer-to-computer comparison of the power consumption of several programs [5]. However, the energy-related data gathered by oZm must be processed manually. The program to be measured must generate start and stop timestamps at the beginning and end of its execution, respectively. Then, once the program is run, the consumption data must be exported from the oZm Web interface to a spreadsheet (in Excel format). Finally, the start and stop timestamps must be located within the spreadsheet to identify the time interval exactly corresponding to the program execution to extract its energy measures.

It is clear, then, that measuring the energy consumed by the execution of a program is a complicated challenge that has been addressed in different ways in the literature with more or less precision. In addition, after analyzing the approaches described above, a direct method based on using any conventional and cheap metering device (as in [5]), able to measure different magnitudes accurately, is preferred, although there still exists a lack of any standard interface for such kind of device that allows programs to access the meter automatically and transparently, which motivates its design, described below.

3. Design of a Standard and Programmatically Accessible Interface for Smart Meters

This work proposes the design of a standard interface to allow automatic monitoring of the energy consumed by the execution of a program and even to let programs be aware of their energy consumption. This design is structured into two abstraction layers, as shown in Figure 1, which provides great flexibility. For example, it lets substitute the meter with a different one transparently, simply by replacing the driver layer with another one that

allows access to the new meter and respects the driver's standard interface. Implementing proxies for different programming languages (Python, Java, C, etc.) is also possible so that any program, regardless of its implementation language, can access information on its real-time energy consumption. In addition, even several proxies for different programming languages can co-exist in the computer.

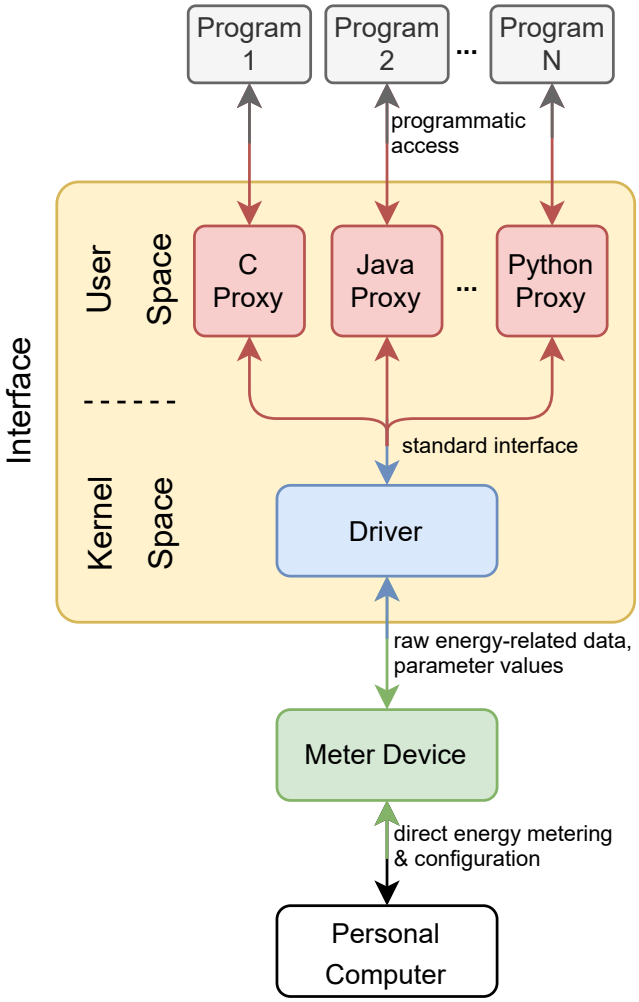

Figure 1. Design diagram of the standard and programmatically accessible interface for smart meters.

3.1. Driver Layer

The low-level layer, called the driver, integrates the measurement device with the PC's operating system (OS), as Figure 2 shows. It should be noted that this layer is not really a driver per se since the meter is a device separated from the PC. However, treating the meter as any other PC's peripheral would allow access to it via the interface provided by the OS to communicate with the PC's subsystems, facilitating a standard interface to the energy-related data programmatically accessible at the OS level. As any driver, it should also hide all the technical details of the meter and be in charge of all the communication issues between the meter and the computer (authentication credentials, connection protocols, etc.).

The driver should be tightly integrated with both the metering device and the PC's OS where the program is running. Thus, its implementation will depend on the chosen meter and the specific OS installed on the computer. A prototype implementation of the driver for a concrete meter and OS is discussed below in Section 4.2.

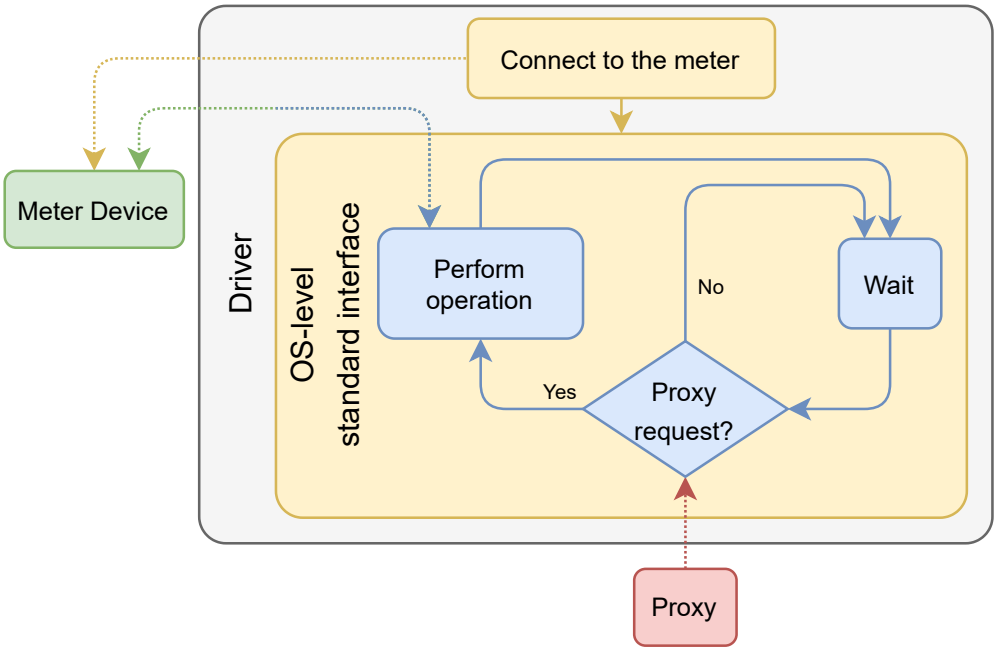

Figure 2. State diagram showing the general behavior of the driver.

3.2. Proxy Layer

The high-level layer, or proxy, should rely on the driver to supply a standard API to access the energy-related measures from a specific programming language. Since this layer provides an API to access the OS-level interface provided by the driver for a concrete programming language, this API may be defined through a set of functions or by a proxy object, depending on the nature of the language. Moreover, several APIs could be implemented to support programmatic access to the meter from different languages on the same computer. In either case, a program has only to invoke a determined proxy function/method to obtain any energy measurement or get/modify any meter parameter, with all the proxy functions being designed to access the standard interface to the meter provided by its driver.

The proxy should also provide some functionalities at a higher level of abstraction, such as the continuous monitoring of a program's energetic behavior throughout its execution, whose data would make it possible to design and fit a model of the program. It could even provide access to any instantaneous energy measurement in real-time, allowing the design of schedulers that dynamically plan the program's execution to minimize its energy consumption, such as that proposed in [16]. Thus, any implementation of the proxy should supply at least the following functions/methods:

- *start(filename, magnitude_list)*: Begins a continuous measurement of the desired energetic magnitudes. The program should now switch to using two threads: one to run the main program or workload (main thread) and another to handle its energy measurement (measurement thread). The latter should execute an infinite measurement loop that periodically gathers the values of all the magnitudes in the list from the driver interface and remains asleep between measurements to minimize its impact on the main thread. Figure 3 shows how the continuous measurement of energy-related magnitudes should work internally. As a result, a file with the given name should be generated. It should store all the gathered data in CSV format, with the first column being a timestamp and the remaining ones containing the values for the measured magnitudes. These data could be used, for instance, to generate a model of the application's energetic behavior.
- *stop()*: Finishes the continuous measurement initiated by *start()*. It should terminate the measurement thread and close the CSV file.

- *get(magnitude/parameter)*: Returns the instantaneous value of any magnitude or configuration parameter. It could be used, for example, to feed an energy-aware scheduler that plans the execution of an application to minimize its energy consumption.
- *set(parameter, value)*: Sets a new value for a configuration parameter. It could be used to modify the continuous measurement sampling period of energy-related data.

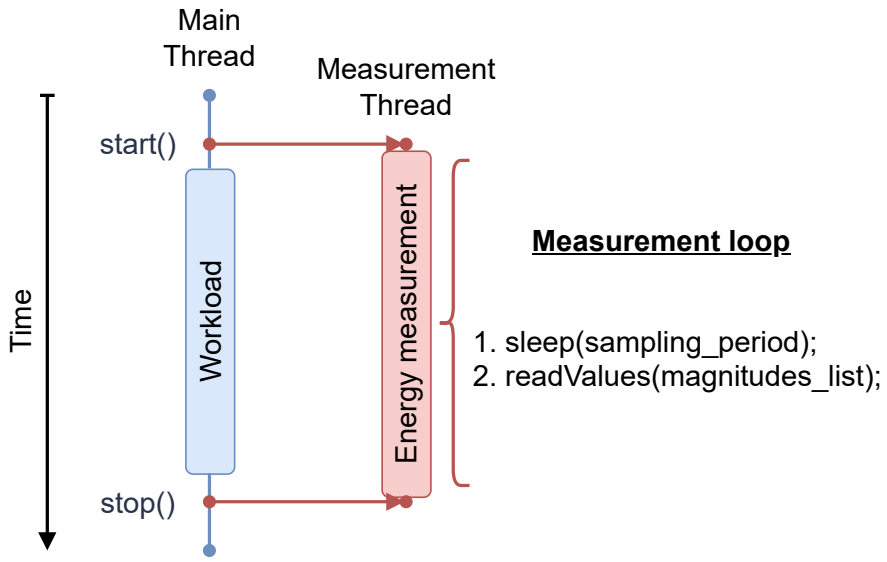

Figure 3. Continuous measurement of energy measurements in the proxy.

4. Prototype Implementation of the Standard and Programmatically Accessible Interface for Smart Meters

Once the design of the standard interface for smart meters has been presented, this section details the implementation of a prototype of it for Linux systems. The driver layer has been implemented to access an oZm meter while the proxy layer defines an API for the C programming language.

4.1. Openzmeter Overview

In 2011, researchers from the University of Almería designed an open-source smart meter and power quality analyzer characterized by its accuracy and reliability: an easy-to-use device named openZmeter (Figure 4). This device was intended to be installed in buildings to collect and process information regarding power supply and energy consumption [32,33] and, unlike other devices available on the market, the oZm has a more comprehensive range of functionalities, follows several international standards such as IEC 61000-4-30 and EN 50160, and offers a comparable measurement error compared to the competition.

As Figure 5 shows, the workflow of oZm starts with the current and voltage sensor, passes to an STM32 microdriver that acquires and preprocesses the samples, and communicates with a NanoPi NEO AIR board, which processes and stores the information, implements the server and deploys the web application. More specific details about oZm can be found in [32].

oZm provides a web-based simple but efficient data query system supported by a server running under OpenWRT Linux. Once connected to the network, and with the device to be measured connected to one of its possible three power outlets, it is ready to provide various types of energy-related data via the web interface. In addition, oZm can calculate the actual cost associated with the energy outlay corresponding to a specific period of use from different electricity tariffs. Figure 6 shows an example of the oZm web interface appearance.

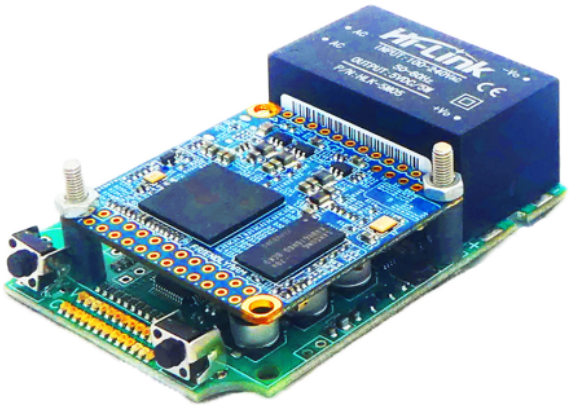

Figure 4. openZmeter device.

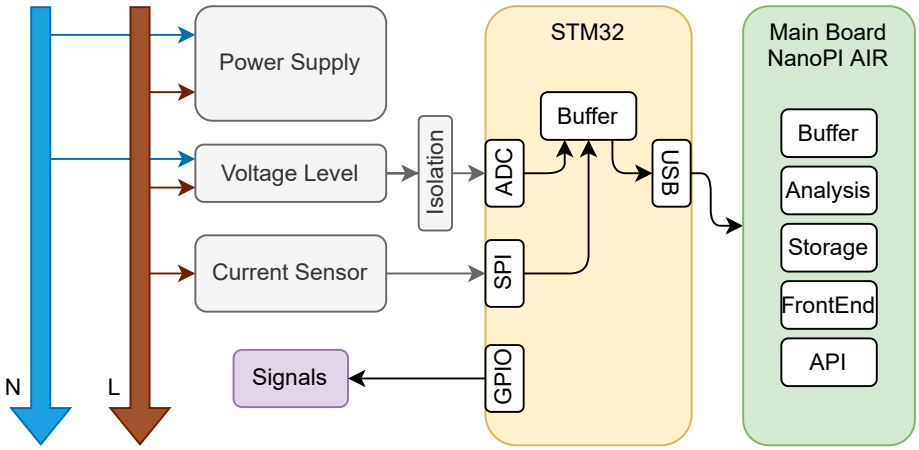

Figure 5. Flowchart for the oZm's electronic scheme.

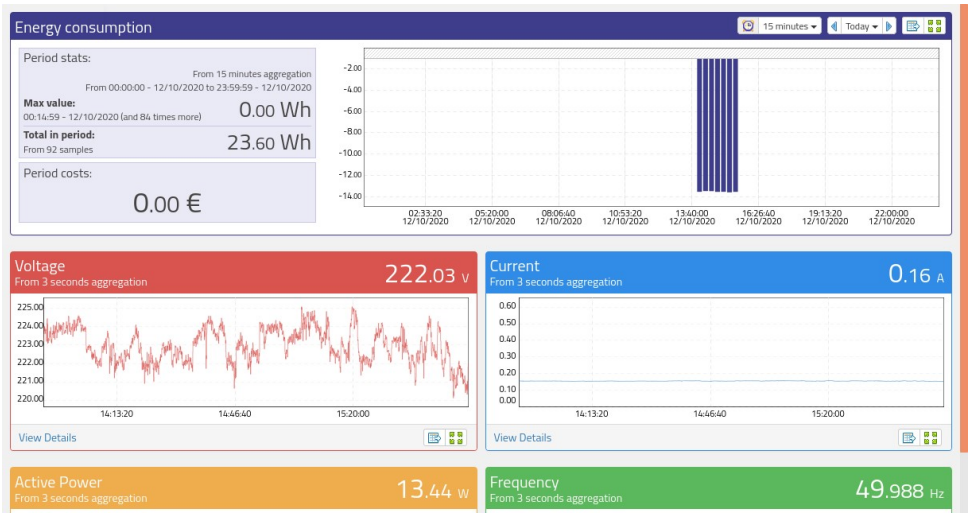

Figure 6. Screenshot of the oZm web interface.

Notwithstanding, it can be seen how this web-based graphical interface, quite useful for conventional users to monitor appliances or even whole homes visually, is not so appropriate for a computer program to query its real-time power consumption, for example. Therefore, it is necessary to have a programmatic and accurate solution for integration into programs, such as the one proposed in Section 3.

4.2. Driver Layer Prototype

This section describes the implementation of a driver prototype for the oZm device as a kernel module for Linux systems. Since this prototype aims to demonstrate the feasibility of the proposed design of a standard interface to provide programmatical access to smart meters, only a partial implementation of the driver has been carried out. As described in Section 3.1, the driver should facilitate a standard interface to access all the magnitudes measured by the meter. Thus, since the access to whatever measure provided by oZm is performed through this interface, implementing the access to just any of them and validating its correct operation can verify the design of the proposed interface design. Therefore, only access to the instantaneous power has been implemented in the prototype. Later, the rest of the magnitudes provided by oZm can easily be supported analogously.

As a Linux kernel module, it should be written in the C language and run in kernel space [34], a different place in memory with its own mapping, where software runs using the highest priority level offered by the microprocessor (supervisor mode). On the contrary, the proxy will run in user space, where applications run at the lowest priority level [35]. The driver is responsible for connecting to the meter, receiving commands from the proxy, and returning data to it. Thus, it should provide a standard but simple interface to the proxy. To this end, with it being one of the novel proposals in this work, the *sysfs* virtual file system was used.

4.2.1. Sysfs as OS-level Interface for the Meter

The *sysfs* [36] was introduced with the Linux 2.6 kernel. It is, in short, a virtual file system. As with any file system, it allows access to the information it handles through folders and files. However, instead of providing access to data already stored in physical devices (such as hard disks or pen drives) or data accessible via a network (remote file systems or cloud storage), it allows access to *kernel objects* by writing to and reading from specific virtual files organized among virtual folders. Since devices can be considered kernel objects, as their access by a program always is supervised by the OS, the *sysfs* allows both sending configuration parameters and getting data from a device by just writing to and reading from the corresponding virtual files in the *sysfs*, as Figure 7 shows. This mechanism is commonly used as a method of communication between the user (from both a program or the command console) and the kernel [37,38]. It offers two main benefits:

- Like any virtual file system, the Linux kernel creates the entire *sysfs* tree at system startup, invoking the different device drivers attached to it. The *sysfs* also disappears when the system shuts down, leading to an automatic cleanup of directories and files if the meter is removed.
- Any virtual file must only handle one single value. Users or programs can get the value of a concrete device parameter/input or set a new value for a parameter/output simply by reading or writing to its associated file, respectively.

Thus, the OS-level standard interface proposed to access any energy meter is composed of the *energy_meter* subdirectory in the *sysfs* and, inside, all the necessary files to send commands and receive information from the device. Each device's configurable parameter or possible input/output datum is linked to just one virtual file in the *sysfs*. This design is highly scalable, capable of supporting any functionality implemented by the device, and even able to support new functionalities that may be incorporated in future versions of the meter.

In addition, this approach provides the advantage that it is possible to communicate with the driver and, therefore, with the device, simply by reading or writing text files from the terminal console, with simple *shell* commands, for example, without the need to use the proxy.

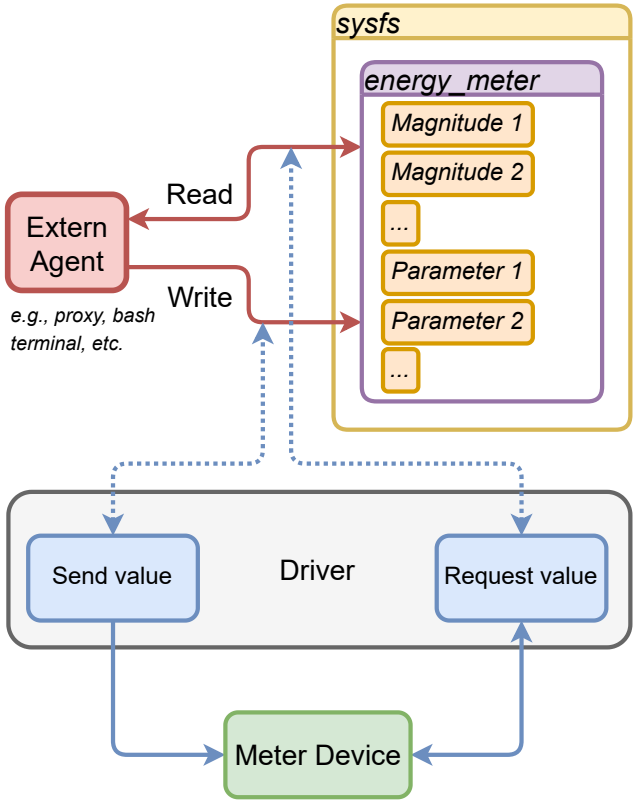

Figure 7. Usage of *sysfs* as OS-level interface for the meter.

4.2.2. Driver Installation

As with any other loadable Linux kernel module, the configuration parameters of the oZm driver layer should be stored within one or several files in the PC's*/etc/modprobe.d* folder, along with the rest of the configuration files used by *modprobe*, the Linux tool used to load a module dynamically to the Linux kernel. These files should be protected by the Linux file permissions system to avoid security issues. So an installation script has been developed for this task. Since oZm is connected to the same router as the PC, as Figure 8 shows, both an IP address and a TCP port are necessary to establish a connection with the meter, so the installation script asks these parameters to the system's administrator and stores them in the *ozm-connection.conf* file, within the */etc/modprobe.d* folder. A login and password are also needed to access the oZm server, so the installation script gathers these data and stores them in the *ozm-credentials.conf* file, also in the */etc/modprobe.d* folder.

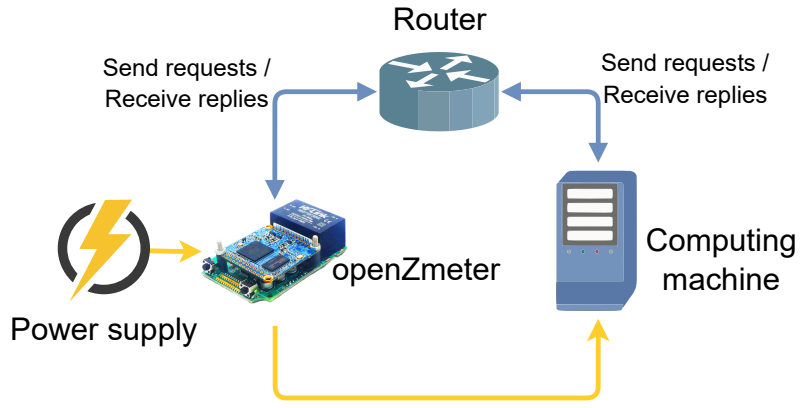

Figure 8. System interconnection.

Finally, since the driver is implemented as a kernel module, it leverages the dynamic kernel module support (DKMS) framework services, a tool created by the Dell Linux engineering team to facilitate the management of kernel modules [39]. With this support, the driver will continue working even after a Linux kernel update as the DKMS compiles the driver source codes against the new kernel automatically to ensure that it will be correctly loaded by the kernel when rebooting the OS after the update.

4.2.3. Driver Initialization

When the computer boots, the Linux kernel loads the meter driver, which attempts to connect to oZm using the data provided by the *ozm-connection.conf* and *ozm-credentials.conf* files. If the login is successful, the driver obtains a session ID, which is stored in memory for later use, so the driver will not have to log in again next time. Otherwise, if the login is unsuccessful, the driver remains loaded in memory but will attempt to log in again when any measurement or parameter configuration is required. Finally, the driver creates the *energy_meter* subdirectory within the *sysfs* and, inside, all the files needed to send commands and receive information from the device. As mentioned above, the driver prototype only provides access to the instantaneous power measured by oZm, so only the virtual file *inst_power* is created.

4.2.4. Driver Operation

Once the kernel has loaded the driver, the proxy only has to read or write any virtual file to obtain or modify the meter's corresponding datum, as shown in Figure 7. So when a read is executed on any virtual file, the kernel calls the driver to get the requested value from oZm and return it. On the contrary, when writing to a virtual file, the driver is invoked by the kernel to send the written value to the device. If any error is generated while the driver tries to get or set a value, its associated file in the *sysfs* will keep a negative code, according to the POSIX standard [40].

4.3. Proxy Layer Prototype

The proxy layer should implement the API detailed in Section 3.2 for a concrete programming language. As commented above, even several proxy instances could be developed to support the programmatic access to the meter from different programming languages. However, as the purpose of this section is to produce a prototype to validate the standard interface design proposed in this paper, only one proxy has been developed for the C programming language since the programs used in Section 5 for the experimental validation of the prototype are written in this language. Moreover, this proxy should be based on the standard interface provided by the driver for the OS where the programs are run, Linux in this case. Thus, the proxy functions are implemented to read or write the corresponding virtual files managed by the driver in the *sysfs*, as described in Section 4.2.1.

Regarding the *start* and *stop* functions, on the one hand, the POSIX threads library has been used for their implementation since Linux is POSIX compatible. On the other hand, as the minimum sampling period of oZm is 200 ms, a sampling period of 500 ms has been set for the measurement thread created by the *start* function to avoid reading contention. Listing 1 shows how the proxy prototype should be used to monitor the energetic behavior of a program.

Finally, since the driver prototype only provides the oZm instantaneous power measures through the *inst_power* virtual file in the *sysfs*, the proxy prototype's *get* and *set* functions can only access this magnitude by the moment. Nevertheless, as soon as access to the rest of the magnitudes is supported by the driver, they would be already available for the proxy simply by reading their corresponding virtual files. Notwithstanding, monitoring just the instantaneous power allows the validation of the proposed standard interface design, as discussed in the next section.

Listing 1. Example of use of the oZm proxy to monitor the energy consumption of an application.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libozm.h> // oZm Proxy

int main(int argc, char** argv)
{
    char *filename = "oZm_data.csv";
    char *magnitudes_list[1] = {"inst_power"};

    printf("Starting a measure with oZm:");
    start(filename, magnitudes_list);
    // Run the workload;
    run_workload();
    stop();
    printf("Measurement finished");

    return(0);
}
```

5. Experimental Results

Once the design of the standard interface for smart meters has been exposed and a prototype based on oZm for Linux systems has been implemented, it is time to validate it. To this end, two experiments have been performed: the former analyses its essential operation while the latter uses it to monitor the power consumption of a machine learning application.

Both test programs have been run on a PC under Rocky Linux 8.5 equipped with an Intel Core i9 12900K CPU, which provides eight high-performance cores able to reach 5.1 GHz when all of them are computing and eight efficiency cores that can run at 3.9 GHz in turbo mode. All the cores operate at 3.2 GHz as a baseline, ramping up to turbo mode as soon as they start to compute. Moreover, since the high-performance cores can execute up to 2 threads per core, a total of 24 threads can be executed in parallel by the CPU. The computer also is equipped with an Nvidia 330 GeForce RTX 3080 Ti GPU with 80 Compute Units (CU) running at 1.665 MHz and 12 GB of GDDR6X memory.

5.1. Validation of the Proposed Interface Design

This experiment has been designed to validate whether the proposed interface design operates appropriately. The hypothesis to be tested is that as the sum of the CPU cores' frequencies increases, so does the power consumed by the PC. A well-known and also computationally highly demanding program has been chosen for this task: the Linpack benchmark [41]. The proposed methodology consists of stressing an increasing number of threads with Linpack to analyze how the OS handles the CPU cores and their frequencies and the correlation of the sum of all the CPU cores' frequencies with the active power consumed by the computer.

This experiment provides a couple of results, with the first being how the OS manages CPU cores. Analyzing Figure 9, it can be observed how the sum of frequencies increases much more steeply from running Linpack with 1 to 16 threads than from executing it with 17 to 24 threads. Since the CPU supplies eight high-performance cores able to run two threads simultaneously at a higher frequency and another eight efficiency cores, it can be deduced that the first 16 threads are executed on the high-performance cores, and the last 8 threads are kept on the efficiency cores. Likewise, Figure 10 plots some statistics (minimum, first quartile, median, third quartile, maximum, and outlier values) summarizing the behavior of the instantaneous power measured while executing the Linpack benchmarck on the computer with different numbers of threads, uncovering that the OS switches the measurement thread to an efficiency core when more than 16 threads are executing Linpack, reducing the overall consumed power. On the other hand, it can be observed how the sum of CPU cores' frequencies and the PC's consumed power follow the same trend, as expected, which validates the energy metering system functioning.

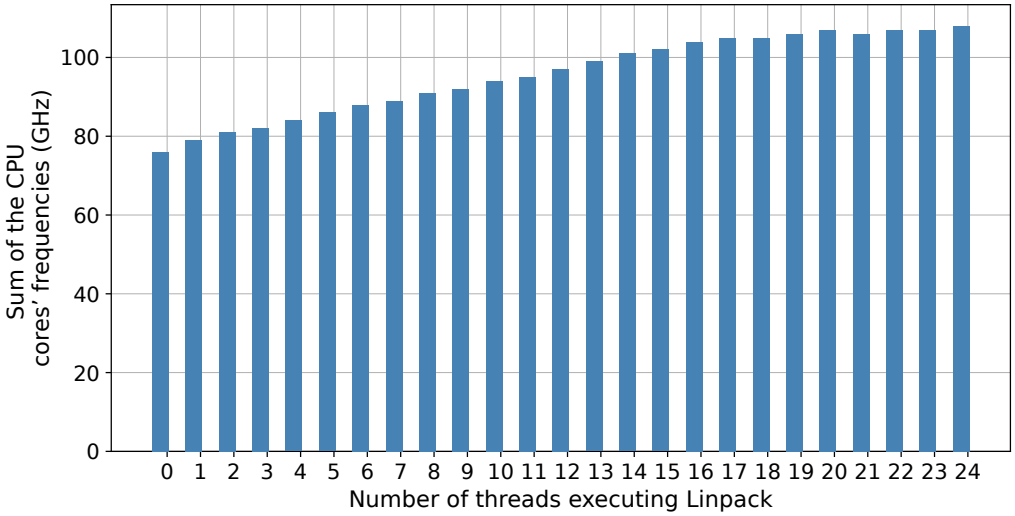

Figure 9. Sum of the CPU cores' frequencies as the number of threads executing Linpack increases.

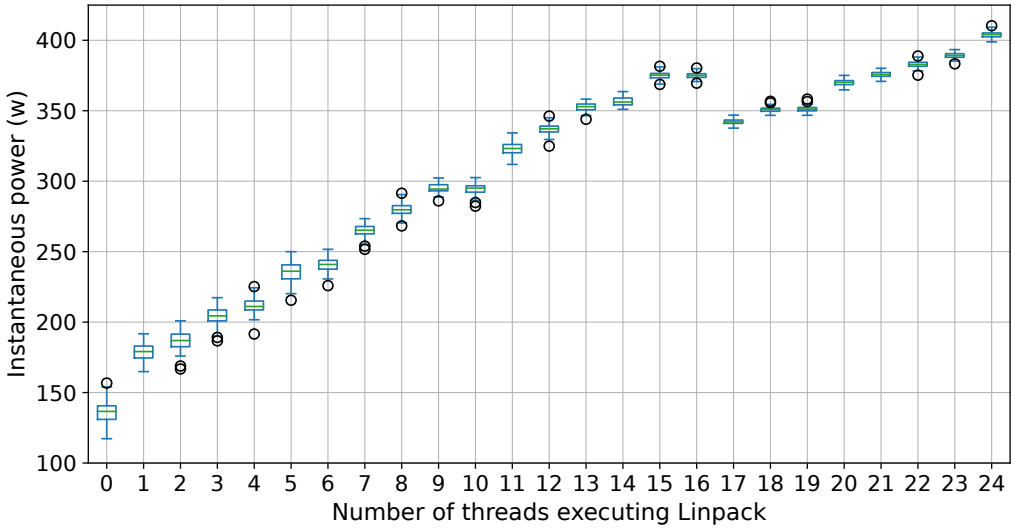

Figure 10. PC's energy consumption as the number of threads executing Linpack increases.

5.2. Heavy Workload Measurement

Once its correct operation has been verified, the standard interface design proposed in this work has also been used to measure the energy consumption of quite a demanding program: a machine learning application. This application allows running its computational load on both the CPU and GPU. Therefore, it is possible to observe the difference in power consumption between both hardware configurations to keep on testing the feasibility and performance of the system.

The test program is the parallel multi-objective evolutionary algorithm proposed in [25], designed to solve feature selection problems for high-dimensional datasets, such as the classification of electroencephalograms. When it runs only on the CPU, it uses all the available threads (24). Alternatively, when executed on the GPU, all the computing units (80) are used, so each computing unit is leveraged at its maximum performance. Figure 11 shows the results obtained, where the instantaneous power consumed by the execution of the program on both the CPU and the GPU are plotted. The mean value over a window of 10 measurements is also displayed to ease the comparison of both alternatives. On the one hand, it can be appreciated how the power consumption suffers almost constant ups and downs for the CPU execution. This behavior is due to the fact that the program is not entirely parallel and each time it changes its execution from parallel to sequential and

vice versa, an abrupt change in power consumption is produced. Maximum peaks of up to 307 watts and average power of 243 watts are observed, with a total execution time of 7923 s. On the other hand, power consumption jumps also occur for the GPU execution but are much less pronounced. In this case, maximum peaks of up to 353 watts and average power of 333 watts are observed, with a total runtime of 951 s.

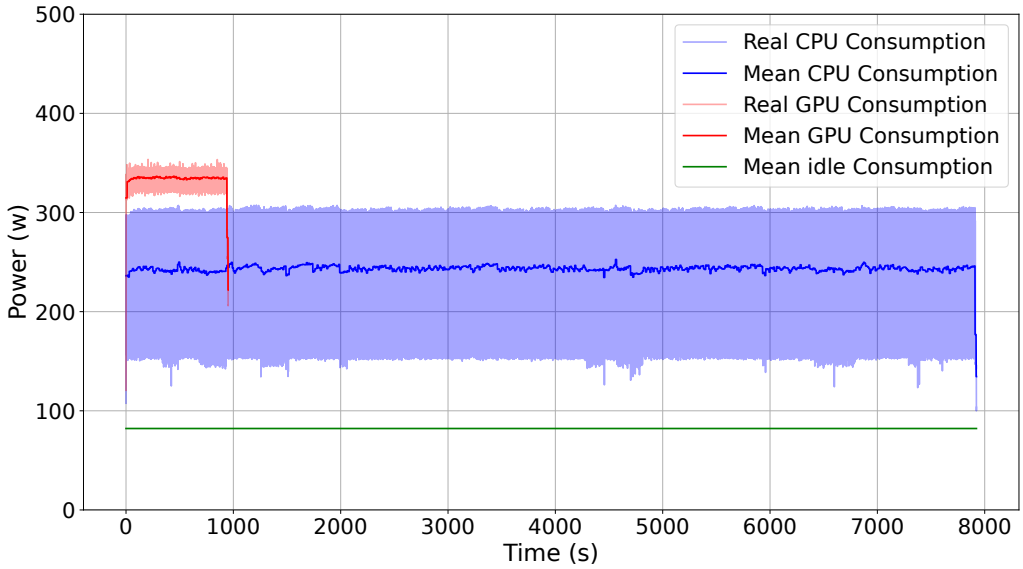

Figure 11. Power consumption of the parallel multi-objective evolutionary algorithm for the CPU and GPU executions.

At first glance, it could be said that the program's execution on the GPU is much less expensive in terms of energy than its execution on the CPU, but it would not be possible to know how much energy has been saved. However, thanks to the prototype of the standard interface for oZm implemented in this work, the instantaneous power consumption of both executions has been collected, integrated, and analyzed to obtain the total energy consumed for each one of its executions. In this case, after calculating the numerical integration of the active power consumed by the program for both the CPU and GPU execution, a total of 535.30 Wh were spent for the CPU run, while only 88.23 Wh were for the GPU execution. Thus, it can be concluded that the GPU execution is 8.33 times faster and spends 6.07 times less energy than the CPU execution.

6. Conclusions and Future Work

Throughout this article, a standard interface for any currently available smart meter, programmatically accessible from any software application, has been designed, implemented, and validated. Although an oZm has been used to test the implementation of the proposed interface design, it should be pointed out that it allows use of any metering device, since the driver layer isolates it from the proxy layer and, thus, from any programmatic access to the meter. The proposed interface design is also modular since it allows the replacement of any of its components with a new implementation of the same layer, letting substitute the device and its driver or add a proxy for another programming language transparently.

Since the driver layer provides an OS-level standard interface to the meter, the proposed interface design is highly scalable too. For instance, the implemented prototype offers a *sysfs*-based interface. Thus, access to any configuration parameter of electric magnitude can be performed by simply reading or writing to a virtual file. The actualization of the meter or even the change for another different device would imply only a modification on the driver, which could eventually add more virtual files to support new magnitudes. On the other hand, although only a proxy for the C language has been developed in the prototype of the proposed interface, as the proxy basically offers an API to access the driver's

OS-level standard interface from a concrete programming language, several proxies could be developed and even co-exist in the same computer to provide programmatic access to the meter for different languages, e.g., MATLAB, Python, Java, etc.

Furthermore, the continuous monitoring of a program execution provided by the proxy makes possible the analysis of different electric magnitudes, the calculation of statistics, and even the detection of the program parts that are not leveraging all the potential of the computer.

As a future work, a third layer designed to integrate the measures taken from the different nodes of a computing cluster would make it possible to analyze and optimize the energetic behavior of distributed applications. The proposed smart meter standard interface design with programmatical access also makes viable the development of a scheduler, such as that proposed in [16], to dynamically manage the computing resources used by a program, which would help meet a specific energy budget during its execution.

Author Contributions: Conceptualization, J.G. and A.O.; methodology, J.G. and B.P.; software, A.M.C.-D. and A.O.; validation, A.M.C.-D., A.O., X.X. and B.P.; writing—original draft preparation, A.M.C.-D. and A.O.; writing—review and editing, J.G. and B.P.; supervision, J.G. and B.P.; project administration, J.G.; funding acquisition, J.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by both the Spanish Ministry of Science, Innovation and Universities, and the ERDF fund, grant number PGC2018-098813-B-C31.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data discussed in this study are available on request from the corresponding author.

Acknowledgments: Special thanks to Francisco G. Montoya and Eduardo Viciana for support with the low-level details of oZm and Juan José Escobar for allowing us to use his feature selection application for the second experiment.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Andrae, A.S.G.; Edler, T. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* **2015**, *6*, 117–157. [CrossRef]
2. Sipma, J.; Broc, J.S.; Skema, R. *Comparing Estimated versus Measured Energy Savings. Topical Case Study of the Epatee Project*; European Union's Horizon 2020 Programme; 2019. Available online: https://epatee.eu/sites/default/files/files/epatee_topical_case_study_comparing_estimated_vs_measured_energy_savings.pdf (accessed on 28 November 2022).
3. Umar, T. Making future floating cities sustainable: A way forward. *Proc. Inst. Civ. Eng. Urban Des. Plan.* **2020**, *173*, 214–237. [CrossRef]
4. Umar, T. Key factors influencing the implementation of three-dimensional printing in construction. *Proc. Inst. Civ. Eng. Manag. Procure. Law* **2021**, *174*, 104–117. [CrossRef]
5. Prieto, B.; Escobar, J.J.; Gómez-López, J.C.; Díaz, A.F.; Lampert, T. Energy Efficiency of Personal Computers: A Comparative Analysis. *Sustainability* **2022**, *14*, 12829. [CrossRef]
6. Tamkittikhun, N.; Tantidham, T.; Intakot, P. AC power meter design based on Arduino: Multichannel single-phase approach. In Proceedings of the 2015 International Computer Science and Engineering Conference (ICSEC), Chiang Mai, Thailand, 23–26 November 2015; pp. 1–5. [CrossRef]
7. Automated Smart Metering; Visalatchi, S.; Sandeep, K.K. Smart energy metering and power theft control using arduino & GSM. In Proceedings of the 2017 2nd International Conference for Convergence in Technology (I2CT), Mumbai, India, 7–9 April 2017; pp. 858–961. [CrossRef]
8. Kumar, A.; Thakur, S.; Bhattacharjee, P. Real Time Monitoring of AMR Enabled Energy Meter for AMI in Smart City-An IoT Application. In Proceedings of the 2018 IEEE International Symposium on Smart Electronic Systems (iSES), Hyderabad, India, 17–19 December 2018; pp. 219–222. [CrossRef]
9. Prathik, M.; Anitha, K.; Anitha, V. Smart Energy Meter Surveillance Using IoT. In Proceedings of the 2018 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS), Chennai, India, 22–23 February 2018; pp. 186–189. [CrossRef]
10. Abate, F.; Carratù, M.; Liguori, C.; Paciello, V. A low cost smart power meter for IoT. *Measurement* **2019**, *136*, 59–66. [CrossRef]

11. Faisal, M.; Karim, T.F.; Pavel, A.R.; Hossen, M.S.; Lipu, M.H. Development of Smart Energy Meter for Energy Cost Analysis of Conventional Grid and Solar Energy. In Proceedings of the 2019 International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST), Dhaka, Bangladesh, 10–12 January 2019; pp. 91–95. [CrossRef]
12. Kumar, L.A.; Indragandhi, V.; Selvamathi, R.; Vijayakumar, V.; Ravi, L.; Subramaniyaswamy, V. Design, power quality analysis, and implementation of smart energy meter using internet of things. *Comput. Electr. Eng.* **2021**, *93*, 107203. [CrossRef]
13. Rashid, M.; Ardito, L.; Torchiano, M. Energy Consumption Analysis of Algorithms Implementations. In Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Beijing, China, 22–23 October 2015; pp. 1–4. [CrossRef]
14. Bunse, C.; Höpfner, H.; Mansour, E.; Roychoudhury, S. Exploring the Energy Consumption of Data Sorting Algorithms in Embedded and Mobile Environments. In Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware, Taipei, Taiwan, 18–20 May 2009; pp. 600–607. [CrossRef]
15. Potlapally, N.R.; Ravi, S.; Raghunathan, A.; Jha, N.K. Analyzing the Energy Consumption of Security Protocols. In Proceedings of the 2003 International Symposium on Low Power Electronics and Design, Seoul, Republic of Korea, 25–27 August 2003; Verbauwhede, I., Roh, H., Eds.; ACM: New York, NY, USA, 2003; pp. 30–35. [CrossRef]
16. Escobar, J.J.; Ortega, J.; Díaz, A.F.; González, J.; Damas, M. Energy-aware Load Balancing of Parallel Evolutionary Algorithms with Heavy Fitness Functions in Heterogeneous CPU-GPU Architectures. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e4688. [CrossRef]
17. García-Martín, E.; Rodrigues, C.F.; Riley, G.; Grahn, H. Estimation of energy consumption in machine learning. *J. Parallel Distrib. Comput.* **2019**, *134*, 75–88. [CrossRef]
18. Dayarathna, M.; Wen, Y.; Fan, R. Data Center Energy Consumption Modeling: A Survey. *IEEE Commun. Surv. Tutor.* **2016**, *18*, 732–794. [CrossRef]
19. Rong, H.; Zhang, H.; Xiao, S.; Li, C.; Hu, C. Optimizing energy consumption for data centers. *Renew. Sustain. Energy Rev.* **2016**, *58*, 674–691. [CrossRef]
20. Bridges, R.A.; Neena, I.; Mintz, T.M. Understanding GPU Power: A Survey of Profiling, Modeling, and Simulation Methods. *ACM Comput. Surv.* **2017**, *149*, 41. [CrossRef]
21. Bertran, R.; Gonzalez, M.; Martorell, X.; Navarro, N.; Ayguade, E. Decomposable and responsive power models for multicore processors using performance counters. In Proceedings of the 24th ACM International Conference on Supercomputing ; Boku, T., Nakashima, H.; Mendelson, A., Eds.; ACM: New York, NY, USA, 2010; pp. 147–158. [CrossRef]
22. Ahmad, R.W.; Gani, A.; Hamid, S.H.A.; Xi, F.; Shiraz, M. A Review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *J. Netw. Comput. Appl.* **2015**, *58*, 42–59. [CrossRef]
23. Hoque, M.A.; Siekkinen, M.; Khan, K.N.; Xiao, Y.; Tarkoma, S. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* **2016**, *48*, 39. [CrossRef]
24. O'brien, K.; Pietri, I.; Reddy, R.; Lastovetsky, A.; Sakellariou, R. A Survey of Power and Energy Predictive Models in HPC Systems and Applications. *ACM Comput. Surv.* **2018**, *50*, 37. [CrossRef]
25. Escobar, J.J.; Ortega, J.; Díaz, A.F.; González, J.; Damas, M. Time-energy analysis of multilevel parallelism in heterogeneous clusters: The case of EEG classification in BCI tasks. *J. Supercomput.* **2019**, *75*, 3397–3425. [CrossRef]
26. Noureddine, A.; Rouvoy, R.; Seinturier, L. A Review of Energy Measurement Approaches. *ACM Sigops Oper. Syst. Rev.* **2013**, *47*, 42–49. [CrossRef]
27. Fahad, M.; Shahid, A.; Manumachu, R.R.; Lastovetsky, A. A Comparative Study of Methods for Measurement of Energy of Computing. *Energies* **2019**, *12*, 2204. [CrossRef]
28. Khan, K.N.; Hirki, M.; Niemi, T.; Nurminen, J.K.; Ou, Z. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*, 9. [CrossRef]
29. Gómez-López, J.C.; Escobar, J.J.; Díaz, A.F.; Damas, M.; Gil-Montoya, F.; González, J. Boosting the Convergence of a GA-based Wrapper for Feature Selection Problems on High-dimensional Data. In Proceedings of the GECCO '22: Proceedings of the Genetic and Evolutionary Computation Conference Companion ; Fieldsend, J.E., Ed.; ACM: New York, NY, USA, 2022; pp. 431–434. [CrossRef]
30. Top500. Energy Efficient High Performance Computing Power Measurement Methodology. Available online: https://www.top500.org/static/media/uploads/methodology-2.0rc1.pdf (accessed on 26 August 2022).
31. Top500. Green500. Available online: https://www.top500.org/lists/green500/ (accessed on 26 August 2022).
32. Viciana, E.; Alcayde, A.; Montoya, F.G.; Baños, R.; Arrabal-Campos, F.M.; Zapata-Sierra, A.; Manzano-Agugliaro, F. OpenZmeter: An Efficient Low-Cost Energy Smart Meter and Power Quality Analyzer. *Sustainability* **2018**, *10*, 4038. [CrossRef]
33. Viciana, E.; Alcayde, A.; Montoya, F.G.; Baños, R.; Arrabal-Campos, F.M.; Manzano-Agugliaro, F. An Open Hardware Design for Internet of Things Power Quality and Energy Saving Solutions. *Sensors* **2019**, *19*, 627. [CrossRef] [PubMed]
34. Mochel, P. The Linux Kernel Driver Model. The Linux Kernel Documentation. Available online: https://docs.kernel.org/driver-api/driver-model/overview.html (accessed on 25 November 2022).
35. Corbet, J.; Rubini, A.; Kroah-Hartman, G. *Linux Device Drivers*, 3rd ed.; O'Reilly Media: Sebastopol, CA, USA 1998.
36. Mochel, P. The sysfs Filesystem. In *Linux Symposium*; The Linux Foundation: San Francisco, CA, USA, 2005; Volume 1, pp. 313–326. Available online: https://www.kernel.org/doc/ols/2005/ols2005v1-pages-321-334.pdf (accessed on 3 May 2022).

37. Maliye, S.; Krishnaswamy, S.; Gajula, H. Quick access of sysfs entries through custom system call. In Proceedings of the 2016 International Conference on Microelectronics, Computing and Communications (MicroCom), Durgapur, India, 23–25 January 2016; pp. 1–4. [CrossRef]
38. Wang, B.; Wang, B.; Xiong, Q. The comparison of communication methods between user and Kernel space in embedded Linux. In Proceedings of the International Conference on Computational Problem-Solving, Li Jiang, China, 3–5 December 2010; pp. 234–237. Available online: https://ieeexplore.ieee.org/document/5696027 (accessed on 22 November 2022).
39. Domsch, M.; Lerhaupt, G. Dynamic Kernel Module Support: From Theory to Practice. In *Linux Symposium*; The Linux Foundation: San Francisco, CA, USA, 2004; Volume 1, pp. 187–202. Available online: https://www.kernel.org/doc/ols/2004/ols2004v1-pages-187-202.pdf (accessed on 18 November 2022).
40. The Linux Man-Pages Project. ERRNO(3)—Linux Programmer's Manual. Available online: https://man7.org/linux/man-pages/man3/errno.3.html (accessed on 25 May 2022).
41. Top500. The Linpack Benchmark. Available online: https://www.top500.org/project/linpack/ (accessed on 25 November 2022)