*electronics*

MDPI

*Article*

# Analysis of Energy Consumption and Optimization Techniques for Writing Energy-Efficient Code

**Javier Corral-García [1,*], Felipe Lemus-Prieto [1], José-Luis González-Sánchez [1] and Miguel-Ángel Pérez-Toledano [2]**

[1] CénitS–COMPUTAEX, Extremadura Supercomputing, Technological Innovation and Research Center, 10071 Cáceres, Spain; felipe.lemus@cenits.es (F.L.-P.); joseluis.gonzalez@cenits.es (J.-L.G.-S.)
[2] Computer Science Department, University of Extremadura, 10003 Cáceres, Spain; toledano@unex.es
* Correspondence: javier.corral@cenits.es; Tel.: +34-927-049-070

Received: 29 September 2019; Accepted: 16 October 2019; Published: 19 October 2019


**Abstract:** The unprecedented growth of connected devices, together with the remarkable convergence of a wide variety of technologies, have led to an exponential increase in the services that the internet of things (IoT) can offer, all aimed at improving quality of life. Consequently, in order to meet the numerous challenges this produces, the IoT has become a major subject of research. One of these challenges is the reduction of energy consumption given the significant limitations of some devices. In addition, although the search for energy efficiency was initially focused on hardware, it has become a concern for software developers too. In fact, it has become an intense area of research with the principal objective of analyzing and optimizing the energy consumption of software systems. This research analyzes the energy saving that can be achieved when using a broad set of techniques for writing energy-efficient code for Raspberry Pi devices. It also demonstrates that programmers can save more energy if they apply the proposed techniques manually than when relying on other automatic optimization options offered by the GNU compiler collection (GCC). Thus, it is important that programmers are aware of the significant impact these techniques can have on an application's energy consumption.

**Keywords:** Raspberry Pi; energy-efficient code; energy optimization; energy performance

## 1. Introduction

The internet of things (IoT) has experienced exponential growth in recent years, and it is increasingly improving the services it offers, thanks to the remarkable convergence of a wide variety of technologies such as sensors, wireless communications and Internet protocols, among others. Thus, IoT currently has an unlimited scope of application, and it is being implemented in all types of devices used in areas as diverse as homes, businesses, industries, transport systems, and even wearable technology and medical devices.

With the constant expansion of IoT applications into everyday use, single-board computers (SBC) have become key devices that are being extensively used in all types of projects, since, although its computing function is limited, they are highly suitable for implementing low-power mobile systems in an inexpensive way due to their compact design, excellent cost/performance ratio, and low energy consumption. Therefore, the amount of research into these types of devices has also grown exponentially in recent years.

Raspberry Pi (RPi) [1] boards are among the most highly demanded SBC devices (25 million units have been sold in the last seven years, without including official accessories, according to March 2019 data [1]). RPi are very cost-effective computers, with a high cost–benefit ratio and appropriate size (3.5 × 2.3 × 0.76 inches), whose performance cannot be compared to that of high end servers,

but which stands out due to their mobility and the computing power they offer per watt, allowing them to execute important computational tasks with low energy consumption and making them suitable for projects in which energy saving is a key issue.

Currently, RPi is being widely used in all kinds of research that looks for solutions in areas as diverse as: health [2,3], industry [4], automotive sector [5], building [6] and domotics [7], education [8] or chemistry [9], just to name a few.

Consequently, in order to meet the numerous challenges this produces, the IoT has become a major subject of research. One of these challenges is the reduction of energy consumption given the significant limitations of some devices. The concept of energy efficiency can be understood as the performance achieved in relation to the performance achievable in an optimal situation. Although it was initially focused on hardware, it has become a concern for software developers too [10]. In fact, it has become an intense area of research with the main objective of analyzing and optimizing the energy consumption of software systems.

In our previous work [11], we presented the analysis of a broad set of techniques for the efficient writing of programming code in RPi devices, all of them focused on reducing execution time without modifying neither the semantic of the program nor its results. The current research aims to analyze the energy saving that can be achieved when using each of these techniques, with the objective of demonstrating their effectiveness when writing energy-efficient code and confirming that, in this case, the automatic help offered by the compiler is also insufficient in terms of energy saving. Thus, this research evaluates the energy consumed when applying these techniques manually, compared to the results obtained when using the automatic help offered by the GNU compiler collection (GCC) [12]. It demonstrates that programmers can save more energy if they apply the proposed techniques manually than when relying on the optimization options provided by the compiler.

The rest of the paper is organized as follows: the next section outlines the scope of the research together with the objectives taken into account in the approach; Section 3 discusses related work; Section 4 contains a complete description of the infrastructure and methodology used during the experiments; Section 5 covers the recommended techniques to write energy-efficient code; the results are presented and discussed in Section 6, and current limitations and future work are addressed in Section 7. Finally, Section 8 concludes and summarizes the contributions of the paper.

## 2. Scope and Objectives

The main objective of the current approach is to demonstrate that the optimization techniques covered in [11] achieve not only significant reductions in execution times, but also meaningful gains in terms of energy saving. These improvements become especially important when they are applied to the code of programs that run uninterrupted throughout the year, something very common in IoT devices. In the same way, it is also intended that IoT programmers be aware of the important impact that even small and simple portions of code can have on the energy consumption of their applications.

Some approaches have demonstrated that there is a direct relationship between energy consumption and running time [10,13,14], so that decreasing execution time can imply better energy efficiency. This statement is based on two main points: firstly, the fewer instructions executed, the less energy is used; secondly, the faster a task is accomplished, the sooner the processor sleeps and saves power [15]. However, the fact that optimizations for speed may achieve important reductions in total energy costs cannot be considered a general rule, as energy consumption also depends on a power variable, modelled as follows:

$$E_{nergy}(J) = P_{ower}(W) \times T_{ime}(s). \quad (1)$$

For instance, in some circumstances, a code compiled for size optimization may be more efficient than a code compiled for speed, for example, due to better use of the cache [15].

In this sense, it is important to highlight the impact that programmer skills have on the energy efficiency of the code. Compilers often offer different levels of optimization that help reduce code size, improve memory utilization, speed up executions or optimize the number of input/output

operations. However, as demonstrated in [11] and in this research, if programmers apply several specific techniques manually they can achieve better performances, not only regarding execution time but also in energy consumption, with minimal effort by modifying the code. Therefore, the proposed techniques are suitable for both expert programmers and beginners in IoT programming. In addition, the results discussed in the current proposal can be extrapolated to other arm devices apart from the models analyzed here, especially those related to the IoT.

All the techniques discussed in this approach are focused on C and C++ code, two of the most popular programming languages, along with others such as Java or Python, that were omitted in our previous work due to the need for them to be translated during runtime. In this sense, recent efforts have focused on compiled languages whose translation overhead is incurred just once, and thus, the expected improvements in execution time and energy saving can be greater. The results of the current proposal may be extrapolated to other programming languages, but their analysis has not yet been performed and will be addressed in further studies.

Although profilers and analysis tools were also discarded, they allow programmers to identify inefficiencies that affect the performance of their applications and obtain information about opportunities for energy-efficient improvements [16], often being the first step towards code optimization [17]. In fact, future work is expected to extend the current research by using these kinds of tools to analyze how some of the proposed techniques impact performance and energy efficiency in greater depth.

## 3. Related Work

In [11] we provided a detailed review of the existing literature, specially focused on improving source code optimization to achieve better execution times. The lack of proposals dedicated to writing efficient code for IoT devices was especially noteworthy. Diverse approaches to optimize C and C++ code for other operating systems and architectures were found instead [14,18–30]. Descriptions of all of them can be consulted in the aforementioned previous work.

Nevertheless, it is important to highlight that IoT may benefit from the existing literature on the usage of low-power arm processors for high-performance computing (HPC) where energy efficiency has become a key challenge to address Exascale needs [31], highlighting the importance of the relationship between performance and energy consumption in the development of HPC systems. A survey which includes a list of articles mainly focused on the use of Arm processors to reduce the energy consumption of HPC infrastructure can be found in [32]. For instance, the role played by hardware factors and by some software aspects in the energy-performance landscape of real-life HPC applications is analyzed in detail in [33], and a combination of both architecture and software improvements to optimize performance and energy consumption on arm-based platforms by using dynamic voltage and frequency scaling (DVFS) techniques, multithreading and vectorization is proposed in [34].

With regard to Raspberry Pi energy consumption, its efficiency has been evaluated in several works with the objective of providing software developers and users with indicators for understanding how much energy the device is consuming while running a software application [35,36]. For instance, a research about how Raspberry Pi power drain is affected by the key functionalities that could be performed by end-users on the platform can be found in [37]. This is compared against other types of common personal computers, and techniques and practices that could reduce energy consumption are recommended. In [38], the energy efficiency issue of an SBC-based cluster is addressed in the context of big data applications. They correlate energy utilization for the execution time of several benchmarks using workloads of different sizes.

Next, other proposals related to the scope of our research are described: an approach for power and energy usage for scientific calculation with and without General-Purpose Unit (GPU) acceleration on RPi devices can be found in [39]; energy and execution time of several wearable and mobile devices, including RPi Zero, are compared in [40] with a benchmark to discuss offloading techniques to use for

increasing quality of service (QoS) in IoT applications; a preliminary analysis and modeling of energy consumption of evolutionary algorithms in different devices, including RPi, is introduced in [41]; an estimation of energy consumption in transferring data using an IoT protocol over different QoS levels is presented in [42]; a linear IoT model to deploy processes and data to devices and servers in IoT (considering a RPi as a fog node) reducing the total energy consumption of nodes is introduced in [43]; a study about the service distribution in multi-layer IoT architecture to minimize the total energy consumption is presented in [44]; the evolution of the energy consumption of several RPi models is compared to alternative platforms in [45].

However, the above-mentioned proposals are focused on analyzing and optimizing the energy consumption of IoT and RPi devices, rather than on the energy saving that can be achieved when writing energy-efficient code for RPi boards. Thus, the current manuscript aims to demonstrate the effectiveness of 25 proposed techniques and confirm that, in many cases, the percentage of energy-saving obtained by applying these techniques manually is much higher than those achieved with the automatic optimizations offered by compilers such as GCC.

## 4. Materials and Methods

Five Raspberry Pi board models, whose features are shown in Table 1, were used to conduct the experiments, all of them Model B devices: second-generation (2B) model, third-generation models B and B+ (3B, 3B+), and the fourth-generation model with 1GB RAM (4B, released in September 2019). Model B was chosen over other RPi models because it had the fastest CPUs (apart from most connectivity features), whereas other models have limited computational power. Therefore B boards are usually more suitable for the kind of applications covered by this research [1].

**Table 1.** Raspberry Pi (RPi) comparison chart.

| Model | RPi 2 B | RPi 3 B | RPi 3 B+ | RPi 4 B |
|---|---|---|---|---|
| **SOC Type** | Broadcom BCM2836 | Broadcom BCM2837 | Broadcom BCM2837B0 | Broadcom BCM2711 |
| **CPU Clock** | 4 × Arm Cortex-A7, 900 MHz | 4 × Arm Cortex-A53, 1.2 GHz | 4 × Arm Cortex-A53, 1.4 GHz | 4 × Arm Cortex-A72, 1.5 GHz |
| **RAM** | 1 GB | 1 GB | 1 GB | 1 GB/2 GB/4 GB |
| **GPU** | Broadcom VideoCore IV | Broadcom VideoCore IV | Broadcom VideoCore IV | Broadcom VideoCore VI |
| **USB Ports** | 4 | 4 | 4 | 4 (2 × USB 3.0 + 2 × USB 2.0) |
| **Ethernet** | 100 Mbit/s base Ethernet | 100 Mbit/s base Ethernet | Gigabit Ethernet (max. 300 Mbps) | Gigabit Ethernet (no limit) |
| **Power over Ethernet** | No | No | Yes (requires separate PoE HAT) | Yes (requires separate PoE HAT) |
| **WiFi** | No | WiFi 802.11n | WiFi 802.11ac Dual Band | WiFi 802.11ac Dual Band |
| **Bluetooth** | No | 4.1 | 4.2 BLE | 5.0 BLE |
| **Video Output** | HDMI/3.5 mm Comp./DSI | HDMI/3.5 mm Comp./DSI | HDMI/3.5 mm Comp./DSI | micro-HDMI/3.5 mm Comp./DSI |
| **Audio Output** | $I^2$S/HDMI/3.5 mm Composite | $I^2$S/HDMI/3.5 mm Composite | $I^2$S/HDMI/3.5 mm Composite | $I^2$S/HDMI/3.5 mm Composite |
| **Camera Input** | 15 Pin CSI | 15 Pin CSI | 15 Pin CSI | 15 Pin CSI |
| **GPIO Pins** | 40 | 40 | 40 | 40 |
| **Memory** | MicroSD | MicroSD | MicroSD | MicroSD |

Just as in our previous work, Raspbian operating system was used (as the Raspberry Foundation [1] recommends) in its version Buster 4.19, with all its packages updated on 1 September 2019. All code was compiled using GCC 8.3.0 version (February 2019 released) contained in the last stable version of GCC in the $Raspbian 8.3.0-6+rpi1$ package. SanDisk Ultra microSDHC class 10 memory cards were also used in this case and the experiments were performed with an ambient temperature of 22 °C, reaching the RPi's maximum CPU temperature of 60 °C. In addition, automatic pauses of 3 min were scheduled between each test and after each experiment to avoid possible overheating problems during measurements. It is important to highlight that time results were similar to those shown in our previous work (conducted with Raspbian Stretch 4.14 and GCC 6.3.0 version).

The rest of the details concerning the methodology employed to conduct the tests and measure the execution times can be found in the previous work [11]. The same steps have been followed to perform the experiments shown in the current work. It provides extensive information about the four levels of automatic optimization offered by the GCC compiler and the methodology designed to conduct

the experiments, including the general pseudocode followed in all the experiments, how the accuracy of the tests is ensured, and even the stopwatch class used for the measurements (with a resolution of around 10 milliseconds and a lack of overhead while running).

Very briefly, GCC offers four levels of automatic optimization [12]: default level *O*0, without optimization for execution time (only the cost of compilation is reduced); level *O*1, which offers optimization for code size and execution time; level *O*2, in which the compiler performs nearly all supported optimizations that do not involve a space-speed trade-off; and level *O*3, which offers the maximum level of optimization, but also significantly increases compilation time and may inhibit the ability to debug the program.

In addition, each test is repeated with at least one hundred million iterations (with the objective of obtaining a measurable value and also to discard the test function invocation cost) and ten different measurements are taken (in order to obtain ten execution time samples and reduce the impact of cold starts and cache effects, since random variations are cancelled out and the caches tend to converge on a single value, without outliers). In the same way, variations due to background tasks and context switching are avoided by starting the experiments through Secure Shell (SSH) network protocol, and running only the required services (the rest are reduced as much as possible before performing the experiments, as was described in [11]).

With regard to energy consumption, there are two main possibilities to measure it in an electronic device: connecting a measuring instrument in series to its supply circuit or using a clamp meter (with two jaws which open to allow clamping around the electrical cable to be measured). However, although the use of clamps avoids the disadvantages of placing an ammeter in a circuit, they are commonly designed to measure high-current circuits.

In addition, there are multiple ways to power RPi devices, of which the most common is to use the 5 V micro USB port and supply the energy with a power adapter or a portable battery. Supplying power through the Ethernet port is also possible, but this option is not provided as standard (as indicated in Table 1). Another option consists of using the General-Purpose Input/Output (GPIO) port of the Raspberry, connecting the 5 V pins to a source to feed energy directly to the board. This type of supply allows a multimeter to be connected in series, without having to modify the cable (that would be connected to the micro USB port) or the board's power connector.

In this way, data analysis of the energy consumed in each experiment was carried out using an UNI-T UT61E digital multimeter (UNI-T Co., Dongguan, China), together with a constant voltage source (DC power supply) to maintain stable power reliably. The positive pole of the source of current should be connected to the positive (red) lead of the multimeter, whereas its ground connection (of the power supply) is connected to pin number 6 of the GPIO (according to the technical specifications of RPi [1], pins 9, 14, 20, 25, 30, 34 and 39 would also be valid for the same purpose. In the same way, the negative lead of the multimeter should be connected to pin number 2 (or 4), thus closing the circuit (see Figure 1)).

The results were obtained with the analysis software provided by the multimeter manufacturer (UT61E Interface Program version 4.01), using a computer. The electrical energy consumed in each experiment was continuously monitored during the executions, with a sample rate of two measurements per second. In this way, current values were collected in real time and transmitted through a serial cable connected to a serial-to-usb converter plugged into the computer and stored into a log file which would later be analyzed to extract the results and calculate the average electric current. The meter was set to measure up to a maximum of 10 A, since the instantaneous current values were in a range between 200 mA and 800 mA during the experiments. In the meter, this range implies a resolution of 1 mA with a margin of error of $\pm(0.5\% + 10)$, and the measurement should be displayed and registered in amps. However, in the experiments (discussed below) the current is shown in milliamps for readability, in order to avoid the use of the five decimals that would be necessary to represent the standard deviation if the result were expressed directly in amps.

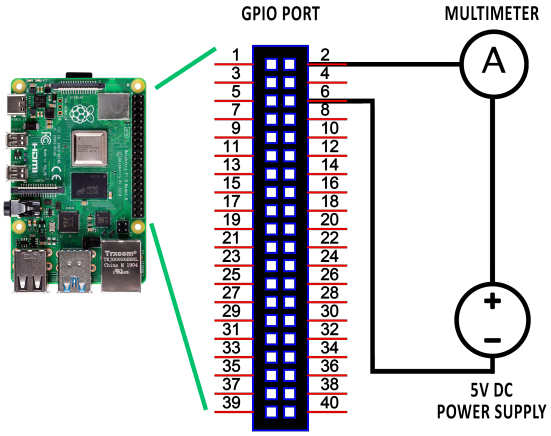


**Figure 1.** Current measuring circuit.

Electric power (measured in watts) is calculated by multiplying the voltage (5 V) by the electric current, modelled as follows:

$$P_{ower}(W) = V \times I, \tag{2}$$

where $V$ is voltage (in volts, V), $I$ is electric current (in amperes, A) and $P_{ower}$ is meant as average power during the code execution. Then energy (measured in joules) can be calculated with the following formula (mentioned in Section 2):

$$E_{nergy}(J) = P \times t \tag{3}$$

where $P$ is electric power (average power during the code execution, in watts, W) and $t$ is time (in seconds, s).

## 5. Techniques to Write Energy-Efficient Code

The following techniques were selected from the literature [18–27] with the objective of using them in an extensive range of programming routines and tasks. They have been analyzed for the current proposal with the aim of demonstrating that, in addition to their suitability for reducing execution times (as demonstrated in [11]), they can also be recommended for writing energy-efficient code for Raspberry Pi boards.

A complete description of each of these techniques, together with the source code of all the designed tests, can be found in [11] along with supplementary materials. Next, a brief excerpt is shown as follows:

T1 Bit fields: a bit field structure holds a sequence of a certain number of bits to improve the use of memory space. However, using an integer instead is recommended.

T2 Boolean return: returning boolean variables at the end of a function is not recommended. It is preferable to convert them into a single *unsigned int* along with the use of associated flags to perform the checks with a single logical operation.

T3 Cascaded function calls: it is advisable to avoid the use of cascaded function calls that return pointers or references.

T4 Row-major accessing: there are two ways of traversing a two-dimensional array. In C and C++ the leftmost index should be incremented first, in order to achieve better cache hit ratio, since data are stored in a contiguous by row-major order.

T5 Constructor initialization lists: it is recommended to use initialization lists to set the initial values of the variables when defining constructors.

T6 Common subexpression elimination: it consists of avoiding the repetition of identical expressions which can be modified so that a single variable holds their computed values.

T7 Mapping structures: in the case of mapping tables, it is recommended to use loops to traverse them, instead of nested $if$ statements.
T8 Dead code elimination: this well-known and widely used technique consists of eliminating instructions that are totally unreachable or do not affect the result (e.g., never used variables).
T9 Exception handling: in order to significantly improve energy efficiency, some exceptions can be replaced by *continue* or *break* statements when they are located within loops.
T10 Global variables within loops: as far as possible, the use of assignments to global variables within loops should be avoided. Local variables can be used instead until the end of the loop is reached.
T11 Function inlining: in many cases, overhead caused by call and return statements can be avoided by directly inserting the code for the function instead of calling it. This technique is considered one of the most important code optimization techniques [21] and has significant advantages over the use of $\#define$ macros [19].
T12 Global variables: the heavy use of global variables is not recommended, due to their associated overhead. In addition, their values can be modified through the program, opening the possibility of collateral problems later.
T13 Constants inside loops: it is recommended to avoid continuous access to constants within loops, even the risk of poor readability.
T14 Initialization versus assignment: variables should be directly initialized, instead of using a subsequent assignment, with the objective of avoiding the assignment of the instantiation.
T15 Division by a power-of-two denominator: division expressions with power-of-two denominators can be changed by shift expressions.
T16 Multiplication by a power-of-two factor: as in the previous technique, multiplication expressions with power-of-two factors can also be changed by shift expressions.
T17 Integer versus character: for arithmetic operations integers should be used instead of characters because in C and C++ char values are converted to integers before operating.
T18 Loop count down: a loop can be traversed in the opposite direction when the order of its counter is not determinative. Traversing the loop in this way implies a faster process, because less instructions are required.
T19 Loop unrolling: some small loops can be unrolled with the objective of decreasing the number of iterations. However, one should be careful to use this technique without reducing the performance of the cache due to an increase in code size, among other reasons.
T20 Passing structures by reference instead of value: it is recommended to pass structures by reference on function calls whenever possible, with the objective of avoiding the overhead of copying all the structures when they are passed by value.
T21 Pointer aliasing: due to the possibility that the same address may be pointed to by two pointers, some optimizations offered by the compiler cannot be automatically applied. For instance, common subexpression should be eliminated manually by the programmer.
T22 Chains of pointers: when using chains of pointers, performance can be improved by the utilization of intermediate local variables to access the final links instead of following the entire chain every time.
T23 Pre-increment versus post-increment: in the case of data types with classes that overload increment operators, it is recommended to use the pre-increment instead of the post-increment operator to avoid the previous copy of the object.
T24 Linear search: *while* statements are recommended to control loop iterations when using linear search method, since they imply a single comparison ($while(list[i] != searched)$) instead of the usual two comparisons (one to control the iterations and another to find the desired key).
T25 Invariant IF statements within loops: it is advisable to take invariant conditional expressions out of the loops whenever possible, although this means an increase in the number of loops.

With the objective of demonstrating the effectiveness of these 25 techniques, and following the measurement algorithm that we introduced in [11], two specific tests were performed for each technique: one test with a standard code, and a second with the result of applying the corresponding efficient technique.

It is important to remember that although all the experiments were based on C++ code, they apply equally to C, and the objective of this research is to demonstrate that the proposed techniques should be taken into account in order to write energy-efficient code.

## 6. Experimental Results and Discussion

Tables 2–5 show the energy saving results related to the application of the proposed techniques in the different RPi devices (2B, 3B, 3B+ and 4).

Optimization levels 1 and 2 have been omitted in the tables, since their values do not improve the results and they are not particularly representative for the objectives of the current approach. Nevertheless, complete results of the experiments conducted can be consulted in the Supplementary Materials.

**Table 2.** Energy results and percentages of improvement in Raspberry Pi 2B.

| | Technique | Energy Consumed (Without Optimization) | | | Energy Consumed (Optimization Level 3) | | |
|---|---|---|---|---|---|---|---|
| | | Standard (nJ) | Efficient (nJ) | Improvement (%) | Standard (nJ) | Efficient (nJ) | Improvement (%) |
| 1 | Bit fields | 121.96 | 66.07 | 45.82 | 51.22 | 48.54 | 5.24 |
| 2 | Boolean return | 56.72 | 52.39 | 7.62 | 28.04 | 27.99 | 0.18 |
| 3 | Cascaded function calls | 1121.10 | 573.22 | 48.87 | 705.75 | 116.99 | 83.42 |
| 4 | Row-major accessing | 13,890.62 | 13,943.61 | 0.00 | 1951.87 | 522.36 | 73.24 |
| 5 | Constructor initialization lists | 1487.91 | 1462.10 | 1.73 | 1418.83 | 1353.56 | 4.60 |
| 6 | Common subexpression elimination | 98.03 | 74.06 | 24.46 | 48.72 | 47.12 | 3.28 |
| 7 | Mapping structures | 732.29 | 1117.32 | 0.00 | 638.66 | 627.11 | 1.81 |
| 8 | Dead code elimination | 44.74 | 34.12 | 23.74 | 23.64 | 23.53 | 0.44 |
| 9 | Exception handling | 13,250.96 | 58.27 | 99.56 | 12,868.40 | 14.13 | 99.89 |
| 10 | Global variables within loops | 745.19 | 572.42 | 23.18 | 119.66 | 121.63 | 0.00 |
| 11 | Function inlining | 77.21 | 49.79 | 35.52 | 28.02 | 27.96 | 0.19 |
| 12 | Global variables | 2126.28 | 1659.54 | 21.95 | 980.14 | 822.56 | 16.08 |
| 13 | Constants inside loops | 673.31 | 473.71 | 29.64 | 315.76 | 316.25 | 0.00 |
| 14 | Initialization versus assignment | 80.44 | 39.64 | 50.73 | 76.50 | 13.95 | 81.76 |
| 15 | Division by a power-of-two denominator | 49.98 | 49.81 | 0.36 | 27.98 | 27.86 | 0.42 |
| 16 | Multiplication by a power-of-two factor | 49.96 | 49.83 | 0.25 | 27.92 | 27.81 | 0.37 |
| 17 | Integer versus character | 81.54 | 73.90 | 9.37 | 27.82 | 28.05 | 0.00 |
| 18 | Loop count down | 2061.13 | 2503.40 | 0.00 | 511.80 | 503.34 | 1.65 |
| 19 | Loop unrolling | 1058.70 | 678.36 | 35.92 | 121.24 | 105.82 | 12.72 |
| 20 | Passing structures by reference | 572.92 | 61.22 | 89.31 | 547.43 | 13.89 | 97.46 |
| 21 | Pointer aliasing | 164.54 | 154.48 | 6.12 | 99.02 | 94.18 | 4.88 |
| 22 | Chains of pointers | 112.10 | 83.87 | 25.19 | 34.17 | 34.06 | 0.29 |
| 23 | Pre-increment versus post-increment | 3706.04 | 3805.12 | 0.00 | 991.48 | 990.06 | 0.14 |
| 24 | Linear search | 3378.41 | 2740.79 | 18.87 | 969.45 | 530.29 | 45.30 |
| 25 | Invariant IF statements within loops | 2990.67 | 2067.03 | 30.88 | 185.21 | 185.20 | 0.01 |

**Table 3.** Energy results and percentages of improvement in Raspberry Pi 3B.

| | Technique | Energy Consumed (Without Optimization) | | | Energy Consumed (Optimization Level 3) | | |
|---|---|---|---|---|---|---|---|
| | | Standard (nJ) | Efficient (nJ) | Improvement (%) | Standard (nJ) | Efficient (nJ) | Improvement (%) |
| 1 | Bit fields | 105.17 | 58.49 | 44.38 | 47.40 | 46.06 | 2.83 |
| 2 | Boolean return | 61.20 | 47.56 | 22.30 | 23.94 | 24.08 | 0.00 |
| 3 | Cascaded function calls | 998.58 | 550.24 | 44.90 | 592.80 | 97.03 | 83.63 |
| 4 | Row-major accessing | 12,681.55 | 12,696.56 | 0.00 | 1903.67 | 267.23 | 85.96 |
| 5 | Constructor initialization lists | 1211.80 | 1162.43 | 4.07 | 1191.65 | 1150.58 | 3.45 |
| 6 | Common subexpression elimination | 90.73 | 69.58 | 23.31 | 48.40 | 46.34 | 4.26 |
| 7 | Mapping structures | 754.68 | 1022.39 | 0.00 | 649.29 | 715.81 | 0.00 |
| 8 | Dead code elimination | 39.63 | 28.93 | 27.02 | 21.20 | 20.77 | 2.02 |
| 9 | Exception handling | 10,676.17 | 51.53 | 99.52 | 10,861.19 | 12.86 | 99.88 |
| 10 | Global variables within loops | 703.81 | 522.70 | 25.73 | 147.87 | 144.62 | 2.20 |
| 11 | Function inlining | 71.35 | 45.14 | 36.73 | 23.98 | 24.10 | 0.00 |
| 12 | Global variables | 1743.59 | 1449.88 | 16.84 | 1023.41 | 925.87 | 9.53 |
| 13 | Constants inside loops | 635.06 | 442.52 | 30.32 | 281.49 | 272.23 | 3.29 |
| 14 | Initialization versus assignment | 76.89 | 36.17 | 52.96 | 71.22 | 12.66 | 82.23 |
| 15 | Division by a power-of-two denominator | 44.38 | 44.50 | 0.00 | 23.99 | 23.90 | 0.39 |
| 16 | Multiplication by a power-of-two factor | 44.42 | 44.37 | 0.11 | 23.97 | 23.94 | 0.11 |
| 17 | Integer versus character | 75.89 | 67.23 | 11.41 | 23.83 | 23.81 | 0.09 |
| 18 | Loop count down | 1919.36 | 2402.28 | 0.00 | 549.88 | 393.33 | 28.47 |
| 19 | Loop unrolling | 993.81 | 623.35 | 37.28 | 94.94 | 105.22 | 0.00 |
| 20 | Passing structures by reference | 592.78 | 56.22 | 90.52 | 498.65 | 12.76 | 97.44 |
| 21 | Pointer aliasing | 153.20 | 143.67 | 6.22 | 90.72 | 86.33 | 4.83 |
| 22 | Chains of pointers | 108.18 | 80.63 | 25.47 | 31.60 | 31.50 | 0.32 |
| 23 | Pre-increment versus post-increment | 3476.96 | 3564.70 | 0.00 | 1065.11 | 1046.03 | 1.79 |
| 24 | Linear search | 3112.78 | 2473.12 | 20.55 | 865.26 | 578.88 | 33.10 |
| 25 | Invariant IF statements within loops | 2839.55 | 1922.75 | 32.29 | 129.04 | 129.64 | 0.00 |

**Table 4.** Energy results and percentages of improvement in Raspberry Pi 3B+.

| | Technique | Energy Consumed (Without Optimization) | | | Energy Consumed (Optimization Level 3) | | |
|---|---|---|---|---|---|---|---|
| | | Standard (nJ) | Efficient (nJ) | Improvement (%) | Standard (nJ) | Efficient (nJ) | Improvement (%) |
| 1 | Bit fields | 149.45 | 82.73 | 44.65 | 66.54 | 64.53 | 3.03 |
| 2 | Boolean return | 86.61 | 67.22 | 22.39 | 33.68 | 33.76 | 0.00 |
| 3 | Cascaded function calls | 1401.71 | 769.40 | 45.11 | 827.05 | 134.82 | 83.70 |
| 4 | Row-major accessing | 18,009.01 | 18,051.36 | 0.00 | 2658.73 | 375.46 | 85.88 |
| 5 | Constructor initialization lists | 1689.81 | 1617.21 | 4.30 | 1651.05 | 1591.84 | 3.59 |
| 6 | Common subexpression elimination | 128.22 | 97.78 | 23.74 | 67.65 | 64.60 | 4.51 |
| 7 | Mapping structures | 1068.11 | 1421.71 | 0.00 | 906.92 | 995.19 | 0.00 |
| 8 | Dead code elimination | 55.53 | 40.30 | 27.42 | 29.71 | 28.95 | 2.54 |
| 9 | Exception handling | 14,790.17 | 72.01 | 99.51 | 15,265.70 | 17.88 | 99.88 |
| 10 | Global variables within loops | 987.25 | 730.92 | 25.96 | 205.91 | 200.43 | 2.66 |
| 11 | Function inlining | 100.34 | 63.23 | 36.98 | 33.79 | 33.96 | 0.00 |
| 12 | Global variables | 2427.07 | 2012.82 | 17.07 | 1426.08 | 1288.63 | 9.64 |
| 13 | Constants inside loops | 887.88 | 617.33 | 30.47 | 392.18 | 379.02 | 3.36 |
| 14 | Initialization versus assignment | 107.27 | 50.60 | 52.83 | 99.65 | 17.75 | 82.18 |
| 15 | Division by a power-of-two denominator | 62.61 | 62.64 | 0.00 | 33.84 | 33.75 | 0.28 |
| 16 | Multiplication by a power-of-two factor | 62.92 | 62.79 | 0.21 | 33.87 | 33.69 | 0.52 |
| 17 | Integer versus character | 106.43 | 94.60 | 11.12 | 33.56 | 33.64 | 0.00 |
| 18 | Loop count down | 2683.04 | 3353.65 | 0.00 | 761.17 | 541.75 | 28.83 |
| 19 | Loop unrolling | 1386.73 | 868.13 | 37.40 | 132.23 | 147.56 | 0.00 |
| 20 | Passing structures by reference | 751.48 | 79.48 | 89.42 | 692.32 | 17.61 | 97.46 |
| 21 | Pointer aliasing | 215.79 | 201.78 | 6.49 | 126.51 | 120.05 | 5.11 |
| 22 | Chains of pointers | 152.42 | 113.12 | 25.79 | 44.42 | 44.31 | 0.25 |
| 23 | Pre-increment versus post-increment | 4843.87 | 4946.66 | 0.00 | 1474.57 | 1442.73 | 2.16 |
| 24 | Linear search | 4346.57 | 3483.12 | 19.87 | 1201.38 | 799.32 | 33.47 |
| 25 | Invariant IF statements within loops | 3995.83 | 2700.42 | 32.42 | 180.29 | 180.58 | 0.00 |

The general pseudocode followed in all the experiments is shown in Algorithm 1. As can be observed it is a linear time algorithm with complexity $O(n)$, where $n$ is the number of iterations of the loop. As described in Section 4, each test is repeated one hundred million times. Thus, the initial measurements should be divided by the number of iterations (to extract the data for a single execution). The internal complexity of each test varies depending on each technique, although, as shown in the source code of the tests (available at [46]) the complexity is constant or linear in most cases except in the row-major accessing technique (T4) due to the double nested loop used to traverse the two-dimensional array generating a quadratic complexity ($O(n^2)$) since it is a square matrix.

**Table 5.** Energy results and percentages of improvement in Raspberry Pi 4B.

| | Technique | Energy Consumed (Without Optimization) | | | Energy Consumed (Optimization Level 3) | | |
|---|---|---|---|---|---|---|---|
| | | Standard (nJ) | Efficient (nJ) | Improvement (%) | Standard (nJ) | Efficient (nJ) | Improvement (%) |
| 1 | Bit fields | 93.08 | 40.84 | 56.12 | 29.38 | 29.57 | 0.00 |
| 2 | Boolean return | 43.99 | 34.30 | 22.02 | 18.64 | 18.65 | 0.00 |
| 3 | Cascaded function calls | 755.31 | 806.87 | 0.00 | 707.72 | 77.15 | 89.10 |
| 4 | Row-major accessing | 18,990.96 | 16,691.18 | 12.11 | 2377.83 | 516.61 | 78.27 |
| 5 | Constructor initialization lists | 944.10 | 925.12 | 2.01 | 1430.88 | 1282.51 | 10.37 |
| 6 | Common subexpression elimination | 62.98 | 59.27 | 5.88 | 27.98 | 31.84 | 0.00 |
| 7 | Mapping structures | 880.96 | 1133.15 | 0.00 | 814.32 | 855.69 | 0.00 |
| 8 | Dead code elimination | 24.48 | 18.36 | 24.97 | 18.72 | 17.53 | 6.38 |
| 9 | Exception handling | 9795.13 | 44.62 | 99.54 | 9510.51 | 19.93 | 99.79 |
| 10 | Global variables within loops | 486.16 | 437.48 | 10.01 | 131.38 | 131.05 | 0.25 |
| 11 | Function inlining | 56.82 | 32.16 | 43.41 | 18.58 | 18.71 | 0.00 |
| 12 | Global variables | 1300.58 | 1163.04 | 10.58 | 912.82 | 822.38 | 9.91 |
| 13 | Constants inside loops | 403.26 | 338.58 | 16.04 | 231.32 | 229.93 | 0.60 |
| 14 | Initialization versus assignment | 54.48 | 25.63 | 52.97 | 53.83 | 17.78 | 66.98 |
| 15 | Division by a power-of-two denominator | 29.27 | 32.00 | 0.00 | 18.55 | 19.40 | 0.00 |
| 16 | Multiplication by a power-of-two factor | 29.51 | 31.98 | 0.00 | 18.62 | 20.54 | 0.00 |
| 17 | Integer versus character | 64.53 | 58.91 | 8.71 | 18.40 | 18.62 | 0.00 |
| 18 | Loop count down | 3735.86 | 3788.18 | 0.00 | 564.20 | 552.82 | 2.02 |
| 19 | Loop unrolling | 1922.48 | 917.66 | 52.27 | 143.33 | 126.70 | 11.61 |
| 20 | Passing structures by reference | 409.63 | 45.26 | 88.95 | 378.96 | 18.63 | 95.08 |
| 21 | Pointer aliasing | 103.04 | 105.81 | 0.00 | 71.62 | 62.04 | 13.37 |
| 22 | Chains of pointers | 78.29 | 89.63 | 0.00 | 25.91 | 26.54 | 0.00 |
| 23 | Pre-increment versus post-increment | 7309.77 | 7349.94 | 0.00 | 1077.51 | 1068.81 | 0.81 |
| 24 | Linear search | 2382.02 | 1684.48 | 29.28 | 641.00 | 591.53 | 7.72 |
| 25 | Invariant IF statements within loops | 4031.94 | 3760.77 | 6.73 | 170.33 | 170.32 | 0.00 |

**Algorithm 1** Test execution

```
1: procedure
2:     start time
3:     for j ← 0, 100000000 do
4:         execute test
5:     end for
6:     get time
7: end procedure
```

In addition, as mentioned above, each procedure was executed ten times, in order to obtain the average result of those ten measurements. The standard deviation was also calculated in order to quantify the dispersion of the measurements. In this sense, it is noteworthy that the standard deviations obtained (that can be consulted in the Supplementary Materials) were very small (mean deviation of 0.28%), indicating that the results tended to be close to the averages shown in the tables.

In addition, as mentioned in the previous section, two tests were implemented for each technique. Thus, the tables show the energy consumed in nanojoules of both tests (standard code and energy-efficient code) for each technique, applying and not applying the automatic optimization offered by the compiler (optimization levels three and zero respectively). Tables also show the percentage of improvement between the execution of the standard code and the efficient one.

With the objective of enhancing the understanding of the data, Figures 2 and 3 are provided to summarize the previous tables.

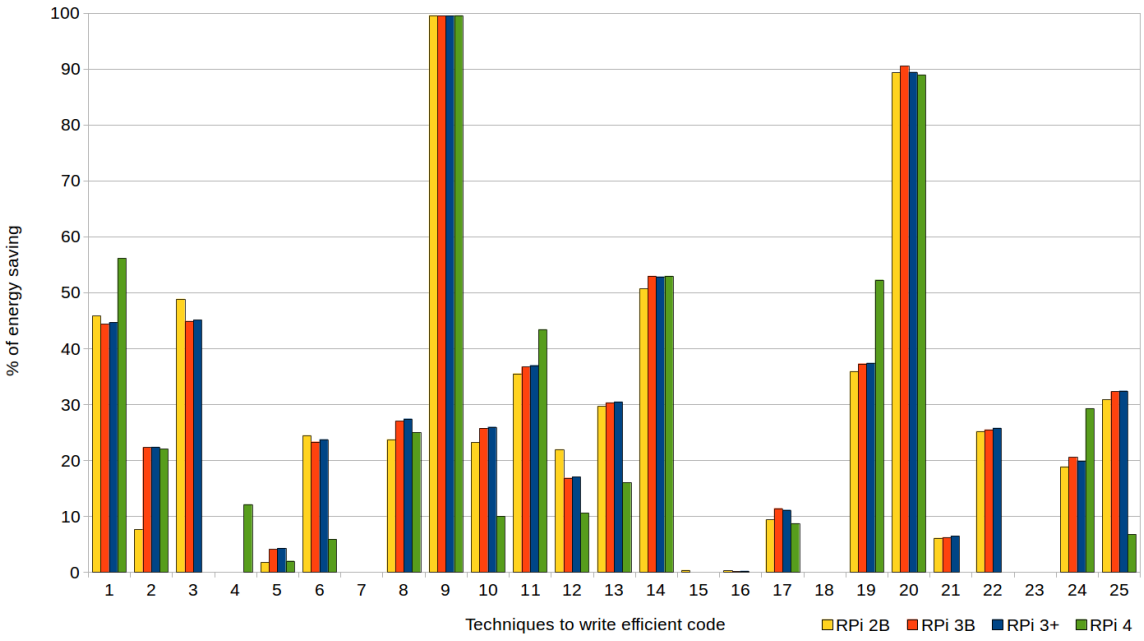


**Figure 2.** Percentages of energy saving achieved by writing energy-efficient code (without compiler optimization).

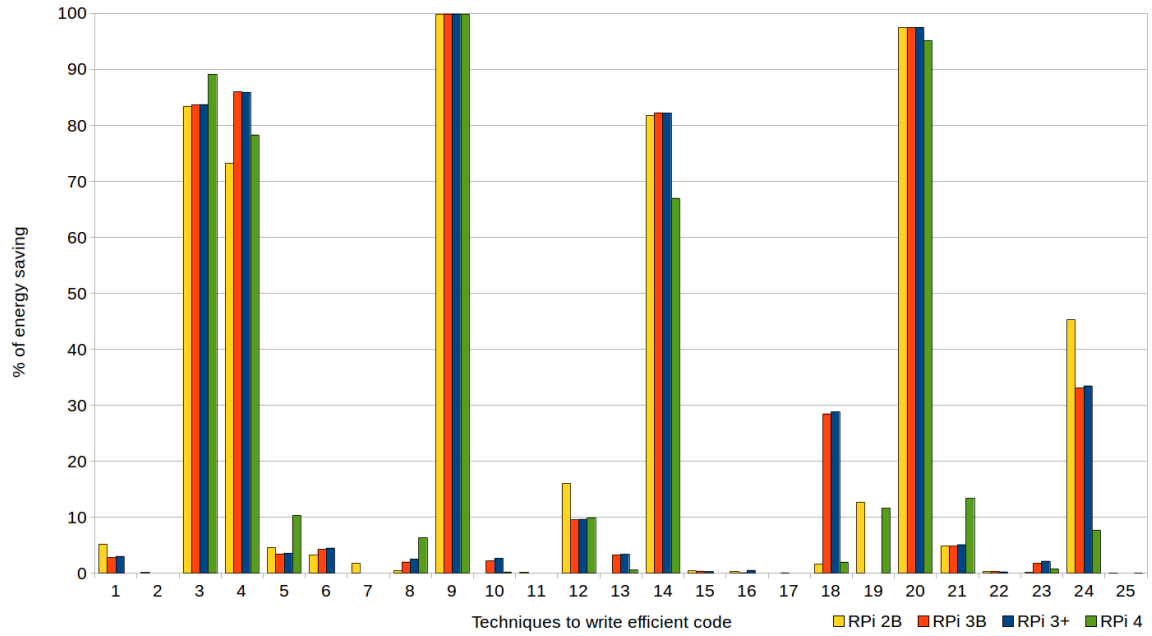


**Figure 3.** Percentages of energy saving achieved by writing energy-efficient code (with compiler optimization level 3).

Both figures show on the horizontal axes the numbers that identify each of the applied techniques, whereas the percentages of energy saving achieved (with respect to the standard code versions) when applying each technique are shown on the verticals. The four models of RPi used in the experiments are represented: 2B (yellow bars), 3B (orange), 3B+ (blue) and 4 (green). Figure 2 shows the results when using optimization level zero (0), that is, no automatic optimization of the compiler is used, whereas Figure 3 represents the data obtained when the tests are compiled with optimization level 3.

It is important to take into account the results obtained with the optimization level zero, since in these cases, unlike in the rest of the optimization levels offered by the compiler, the ability to debug the programs remains unaltered and the expected results can be obtained. Thus, as shown in Figure 2, the use of the proposed techniques is especially recommended for cases in which the debugging processes takes considerable time.

In terms of energy saving, the percentages of improvement obtained are similar to those achieved in terms of decreasing execution time in our previous work [11]. The RPi 4, which did not exist when our previous work was published, demonstrates that the more advanced the Raspberry Pi model is, the shorter its execution times are.

As can be seen in Figures 2 and 3 and Tables 2–5, the application of these techniques achieve energy savings of up to 99.56%. Especially noteworthy are the improvements obtained when applying two widely known techniques: exception handling (T9) and passing structures by reference instead of by value (T20), as in the analysis of execution time conducted in [11].

In general, the improvement percentages are similar when applying a technique in the four models of RPi, although there are some significant differences produced mainly by the RPi 4 tests. For instance, in the case of the optimization level zero (0) (see Figure 2), the following techniques achieve a greater energy saving (up to 10 percentage points more) with the RPi 4 than with the other models: bit fields (T1); row-major accessing (T4); function inlining (T11); loop unrolling (T19); and linear search (T24).

In contrast, the percentage of energy saving with this model is much lower (even non-existent) in the following cases (when compared to the results of the other three models): cascaded function calls (T3); common subexpression elimination (T6); global variables within loops (T10); global

variables (T12); constants inside loops (T13); pointer aliasing (T21); chains of pointers (T22); and invariant IF statements within loops (T25).

With regard to optimization level 3 (see Figure 3), the RPi 4, achieves better energy results than the rest of the models when applying these techniques: cascaded function call (T3); constructor initialization lists (T5); dead code elimination (T8); and pointer aliasing (T21). Whereas, the improvement is smaller for the following techniques: bit fields (T1); common subexpression elimination (T6); global variables (T12); initialization versus assignment (T14); loop count down (T18); and linear search (T24).

Returning to the energy savings obtained without the help of the automatic optimizations of the compiler (see Figure 2), and now considering all the RPi models used, there are several techniques that produce no or very limited improvement. This means their energy efficient code did not consume less energy than the corresponding standard code. This occurs when applying the following techniques: row-major accessing (T4); mapping structures (T7); division by a power-of-two denominator (T15); multiplication by a power-of-two factor (T16); loop count down (T18); and pre-increment versus post-increment (T23). The other nineteen techniques demonstrated their efficiency in writing energy-efficient code when compiler optimization is set to default (i.e., with level zero), achieving an average energy saving of 33.66%, considering the four different models of RPi.

When using optimization level 3 together with the application of the energy-efficient techniques, the automatic help offered by the compiler achieve significant results in most of the tests performed (see Figure 3). However, writing efficient code manually is still recommended with regards to the following techniques, which achieve an average energy saving of 51.33%: cascaded function calls (T3); exception handling (T9); initialization versus assignment (T14); loop count down (T18); passing structures by reference (T20); and linear search (T24). Next, these techniques are analyzed in greater depth using RPi models 3B+ and 4, because these devices achieve better execution times. In addition, RPi 3B+ will remain in production until at least January 2026, and RPi 4 has been recently released (in September 2019).

### *6.1. Cascaded Function Calls (T3)*

Results achieved when applying this technique have demonstrated here that avoiding cascaded function calls that return pointers or references is recommended. The role of the programmer is especially important for this technique, since he or she, unlike the compiler, are able to know if the reference returned by a function remains unchanged during the execution of a particular code fragment. This can be observed in the tests (see Listing 1) which were developed to analyze the percentage of energy saving achieved when using this technique. The standard test consists of a loop that traverses an array and calls a function in each iteration. In the energy-efficient test, just one call is made outside the loop and the value returned by the function is stored in an auxiliary variable which is checked inside the loop in each iteration.

**Listing 1.** Cascaded function calls. Standard and energy-efficient code.

```
void standardCode(Type *argument){
  for (int i=0; i<N; i++)
    if (argument->function()==1)
      a[i] = 0;
}
```

```
void energyEfficientCode(Type *argument){
  int variable = argument->function();
    for (int i=0; i<N; i++)
      if (variable==1)
        a[i] = 0;
}
```

As shown in Figure 4, energy savings of up to 95% are achieved thanks to this technique, with an improvement of 89.04% when using an array of just 20 elements (that is, with just 20 loop iterations). In addition, energy savings vary depending on the size of the array. The improvement of 51.37% reached when using an array size of 3 elements (just three loop iterations) is also noteworthy.

From 100 or more calls, the energy saving stabilizes at around 90%. This demonstrates the effectiveness of a technique which should be applied whenever the above conditions are met.

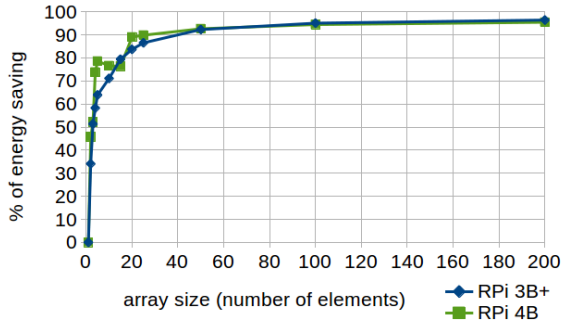


**Figure 4.** Cascaded function calls. Percentage of energy saving according to the number of calls to the function (number of iterations of the loop) in RPi 3B+ and 4B.

### *6.2. Row-Major Accessing (T4)*

Some languages support two-dimensional arrays stored in a contiguous by column-major order, whereas others such as C and C++ use a row-major one, that is, row-by-row, so that the element $x + 1$ is stored right next to $x$. Therefore, two-dimensional arrays should be traversed in the order they were stored, that is, the leftmost index should be incremented first, with the objective of achieving higher rates of cache hits during the accesses. In standard code the array is traversed by iterating over each column first (the outer loop processes the columns whereas the rows are processed by the inner one). Instead, row-major iteration is used for traversing the energy-efficient code (see Listing 2).

The percentages of energy saving achieved by applying this technique grow depending on the array size by up to 85.52%, so that greater percentages of improvement are obtained as the size of the vector increases, principally due to the work increment of the cache (see Figure 5). In this way, as the test is developed by using arrays that represent square matrices, the improvement is influenced by the size of both dimensions. Therefore, a deeper analysis of the effects produced by variations in array sizes will be addressed in future work.

**Listing 2.** Row-major accessing. Standard and energy-efficient code.

```
int standardCode(){
  for (int j=0; j<N; j++)
    for (int i=0; i<N; i++)
      array[i][j] = 0;
}
```

```
int energyEfficientCode(){
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      array[i][j] = 0;
}
```

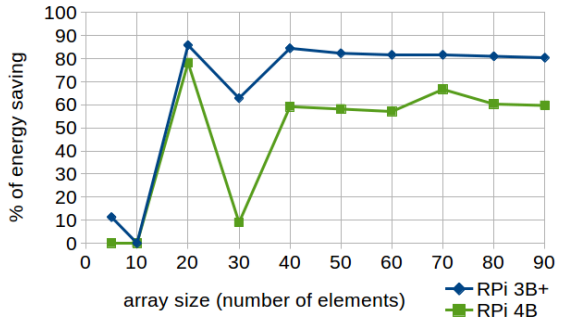


**Figure 5.** Row-major accessing. Percentage of energy saving according to the matrix size in RPi 3B+ and 4B.

The difference between the results of both models, due to fact that the Cortex-A72 of the model 4 features a larger cache which leads to a reduction in the miss penalty when applying standard code, is also noteworthy. In this way, from 40 or more elements, the percentage stabilizes around an improvement of 80% in RPi 3B+ and 60% in RPi 4. This fact also strongly influences the results obtained with an array size of 30 elements, where the cache hit ratio increases.

### *6.3. Exception Handling (T9)*

Although exceptions are needed to detect and manage unexpected errors, they involve significant time penalties which also involve energy consumption. They are produced by successive calls to the exception handler passing a parameter as an argument. When its type and the type of parameter specified by the handler match, the handler takes control of the execution flow.

In some cases when these exceptions are within loops, it is possible to replace them with other statements, such as *continue* and *break*, which allow changing the flow of execution and control possible errors in the same way that exceptions do, but with lower cost (see Listing 3).

As expected, the percentage of energy saving depends on the number of exceptions thrown and the structure of the data, so there are infinite possible variations that can be applied and analyzed in this experiment. Therefore, the evaluation in greater depth of the conditions that influence these improvements has been omitted, along with the comparative graph, especially given the differences obtained between the different RPi models were not significant (as shown in Figure 3).

The objective is to demonstrate that significant energy efficiency improvements can be achieved when this technique is applied. In the standard test, 100 simple exceptions were thrown and, although they did not have any content apart from its declaration (*class myexception : public exception {} myex;*), energy saving reached 99%. Thus, it is recommended to apply this technique to control unexpected errors within loops whenever possible under the above conditions.

**Listing 3.** Exceptions. Standard and energy-efficient code.

```
int standardCode(){
  int variable = 100;
  for (int i=0; i<1; i++){
    try{
      if (variable == 100) {
        throw myex;
      }
    }catch (exception& e){
    }
  }
return 0;
}
```

```
int energyEfficientCode() {
    int variable = 100;
    for (int i=0; i<1; i++){
      if (variable != 100) {
        continue;
      }
    }
  return 0;
}
```

### *6.4. Initialization versus Assignment (T14)*

Whatever the type of data, it is highly advisable to initialize variables directly when they are declared, since a subsequent instantiation implies an unnecessary assignment that can be avoided. The tests (see Listing 4) analyze the energy savings achieved by using complex double numbers and compare direct initialization (energy-efficient code) with the alternative use of a subsequent assignment (standard code). Energy savings of up to 52.97% are achieved using this technique (as shown in Figure 3).

With regard to this technique, the percentage of energy saving depends on the data types, so there are many possible variations that can be applied and analyzed in this experiment (considering primitive, derived and abstract or used-defined data types, along with datatype modifiers, e.g., unsigned or long). Thus, in this case, the evaluation in greater depth of the conditions that influence

these percentages of improvement has also been omitted, along with the comparative graph, since the differences obtained between the different RPi models were not significant.

**Listing 4.** Initialization versus assignment. Standard and energy-efficient code.

```
void standardCode(){
  std::complex<double> mycomplex;
  mycomplex = (3.14);
}
```

```
void energyEfficientCode() {
  std::complex<double> mycomplex(3.14);
}
```

### *6.5. Loop Count Down (T18)*

Those loops in which the order of its counter is not determinative can be traversed in the opposite direction (instead of traversing it forward). The reason lies in the fact that it is faster to process "*i- -*" (energy-efficient code) than $i = 0$; $i < 100$ (standard code) as a loop condition, since traversing it forward implies more steps (see Listing 5). These include subtraction, evaluation of the result, iterator increment (if the result is not zero) and continue. It is worth noting that the loop termination clause also has an important impact on the improvement, since they have to be evaluated an extensive number of times.

As shown in Figure 6, energy savings from 20% to 30% can be achieved when applying this technique using model 3B+ and arrays with more than 20 elements, whereas in model 4 the improvement is almost nil (between 1.29% and 2.86%).

To understand this, it is important to highlight that runtime results are significantly lower in the RPi 4, so that even the execution time of the standard code on the model 4 is up to 50 % faster than the execution time of the efficient code on the RPi 3B+ (as can be consulted in the Supplementary Materials). Thus, this significantly reduces the chances of further improving the results by applying energy-efficient code. This is due not only to the out of order execution and the larger cache of the model 4 (which leads to a reduction in the miss penalty when applying the standard code, reducing the difference with the energy-efficient one), but because of the specific improvements introduced in the Cortex-A72 of the model 4 (that add power and performance optimizations to the previous A53 design of the model 3B+ including an improved branch prediction algorithm, increased dispatch bandwidth, lower-latency execution units, and higher bandwidth L2 cache) there is a performance improvement from 10 to 50% according to Arm [47], especially regarding floating-point workloads.

**Listing 5.** Loop count down. Standard and energy-efficient code.

```
void standardCode(){
  for (int i=0; i<N; i++) {
    a[i]=i;
  }
}
```

```
void energyEfficientCode() {
  int i = N+1;
  while (--i) {
    a[i] = i;
  }
}
```

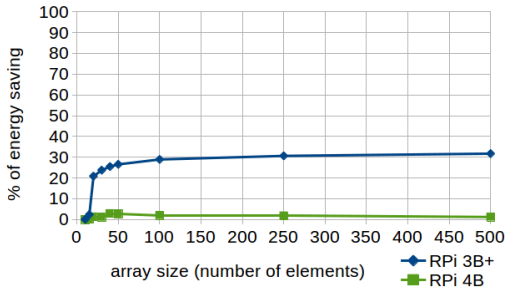


**Figure 6.** Loop count down. Percentages of energy saving according to the array size (number of elements) in RPi 3B+ and 4B.

### *6.6. Passing Structures by Reference (T20)*

Regarding function calls, structures passed by value imply significant overhead produced by the need for a complete copy of each structure (including constructor and destructor classes). Instead, this overhead can be avoided if the structures are passed by reference (see Listing 6). However, it is important to take into account the fact that, unlike value arguments, reference arguments can alter the original instance. Therefore, it is advisable that pointers to structures be declared as constants if the intention is not to modify their pointed values.

The percentage of energy saving produced by this technique depends on the data structures used, therefore there are infinite number of possible variations that can be applied and analyzed in this experiment. Thus, the evaluation in greater depth of the conditions that influence these improvements has also been omitted, along with the comparative graph, since, in addition, the differences obtained between the different RPi models were not significant.

As can be observed, the structure used to demonstrate the effectiveness of this technique consists of a class with two string members and a data substructure formed by an array of 10 elements and an integer variable that serves as an index. The class constructor consists of three assignments (corresponding to the initialization of the two string attributes and the index of the substructure). In addition, the *getIndex* function simply returns the value of the index.

In the standard code, the class is passed by value (implying a complete copy of its structure) and the index is directly accessed ($value.getIndex$), whereas in the energy-efficient code the class is passed by reference so that the index is accessed through a pointer ($reference\text{–>}getIndex()$). This simple modification resulted in an improvement of around 90.54% in the four RPi models.

**Listing 6.** Passing structures by reference. Standard and energy-efficient code.

```
typedef struct {int array[10]; int index;} Structure;

class Class {
  private:
    string    attribute_a ;
    string    attribute_b;
    Structure structure;
  public:
    Class(string attribute1, string attribute2, int i);
    int getIndex();
};

int standardCode(Class value){
  return value.getIndex();
}
```

```
int energyEfficientCode(Class *reference){
    return reference->getIndex();
  }
```

### *6.7. Linear Search (T24)*

In general, two comparisons are required to conduct a common linear search for finding an element within a list: the first comparison to control the loop iterations, and a second one to check if the current element matches the targeted value. In this way, the algorithm sequentially checks all the elements until founding a match or reaching the end of the list with an unsuccessful outcome. This energy-efficient technique consists of replacing the *for* loop with a *while* statement (see Listing 7). In this way, only one comparison ($list[i] != search$) is required within the loop. In addition, it should also be noted that the array size should be increased by one element to insert the target value at the end to avoid potential problems arising if the search value is not contained within the list. In this way, when the found index corresponds to the last position it means that the target value is not contained in the array. The arrays of both types of code (standard and energy-efficient) were initialized by placing the target element in the last position.

As shown in Figure 7, improvements from 20% to 33% with array sizes between 20 and 30 elements are achieved when applying this technique using the RPi 3B+, whereas with model 4 the percentage of improvement is significantly reduced and only reaches 7.63%. It is also important to highlight that although there are faster algorithms, linear search is practical when using short lists or performing single searches in un-ordered lists. In addition, it is a well-known and widely used algorithm which is especially recommended for beginners in programming, since it is simple to implement.

**Listing 7.** Linear search. Standard and energy-efficient code.

```
int standardCode(int *list,int N,int search){
  int i;
  for (i = 0; i < N; i++)
    if (list[i] == search)
      return i;
  return -1;
}
```

```
int energyEfficientCode(int *list,int N,
                        int search){
  int i;
  list[N] = search;
  i = 0;
  while (list[i] != search)
    i++;
  if (i == N)
    return -1;
}
```

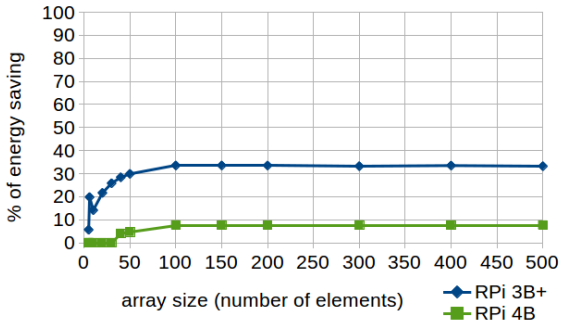


**Figure 7.** Linear search. Percentages of energy saving according to the array size (number of elements) in RPi 3B+ and 4B.

In this case, runtime is also significantly lower in the RPi 4 so that even the execution time of the standard code on the model 4 is up to 50% faster than the execution time of the efficient code on the RPi 3B+ (as can be consulted in the Supplementary Materials). Thus, this significantly reduces the chances of further improving the results by applying energy-efficient code (as discussed in Section 6.5).

In general, the application of these twenty-five techniques achieves significant energy savings and their improvements are similar in the four models of Raspberry Pi used. However, it should also be noted that in some experiments (discussed in greater depth in Sections 6.2, 6.5 and 6.7), model 3B+ achieves better energy savings than model 4, due to the differences between the efficiency-focused Cortex-A53 of the model 3B+ and the performance-oriented Cortex-A72 of the model 4.

## 7. Limitations and Future Work

An extensive number of different tests could have been designed to demonstrate the effectiveness of each energy-efficient technique, since there are infinite possible variations which could have been applied and analyzed in each of the experiments. These variations, together with the wide range of possible workloads, may significantly influence the improvement percentages discussed above. In addition, although the discussed results can be extrapolated to other IoT devices, their percentages of improvement will be also conditioned by the programming language and hardware architecture used.

Consequently, future work is expected to expand the current research to other IoT devices, regardless of architecture, with the objective of analysing the range of potential applications available

for these techniques. In addition, the use of other programming languages and different compilers, apart from GCC, will also be addressed.

With regard to the techniques that have achieved the best results, we hope to evaluate in greater depth various aspects that may improve energy efficiency, such as data types or workloads that need to be considered when writing energy-efficient code. In the same way, conducting a detailed analysis of the improvements that could be achieved by combining some of the proposed techniques will also be considered.

It is also hoped that in the future the current proposal can be extended and applied to energy-efficient networked computing architectures [48], and thus help resolve the challenges that the unpredictable large volume of data generated by Internet of Everything currently faces.

## 8. Conclusions

A set of twenty-five techniques have been analyzed to demonstrate their effectiveness for writing energy-efficient code for Raspberry Pi boards. They were previously selected from the literature with the objective of being used in an extensive range of programming routines and tasks. All of them are focused on transformations aimed at reducing running time and energy consumption while preserving the semantics of the program without altering the algorithm.

The application of these techniques achieves energy savings of up to 99.56% and their improvements are similar in the four models of Raspberry Pi used, although in some experiments, model 3B+ achieves better energy savings than model 4, due to the differences between the efficiency-focused Cortex-A53 of the model 3B+ and the performance-oriented Cortex-A72 of the model 4. In addition, this proposal analyzes the energy consumed through the manual application of these techniques compared to the result of directly applying the automatic optimizations offered by the GCC compiler. In this way, it has been demonstrated that the role of the programmer is essential when it comes to achieving the best energy savings, since the compiler does not obtain the same improvements that a programmer can achieve when applying theses energy-efficient techniques manually.

Therefore, programmers should be aware of the important impact that even small and simple portions of code can have on energy consumption. Applying energy-efficient techniques analyzed here is recommended, especially given that these improvements become important when they are applied to code in programs that run non-stop, very common in IoT devices.

Seven of the twenty-five techniques were analyzed in greater depth, with the objective of evaluating the conditions that influence energy efficiency improvements. In addition, the results discussed in the current research can be extrapolated to other devices with similar Arm architectures, especially those related to the IoT.

**Supplementary Materials:** The following are available online at http://www.mdpi.com/2079-9292/8/10/1192/s1, Spreadsheet S1: complete experimental results.

**Author Contributions:** Conceptualization, J.C.-G.; formal analysis, J.C.-G. and F.-L.P.; funding acquisition, J.-L.G.-S.; investigation, J.C.-G. and F.-L.P.; methodology, J.C.-G.; project administration, J.-L.G.-S. and M.-Á.P.-T.; software, J.C.-G.; supervision, J.-L.G.-S. and M.-Á.P.-T.; validation, J.C.-G. and F.-L.P.; visualization, J.C.-G. and F.-L.P.; writing—original draft, J.C.-G.; writing—review and editing, J.C.-G., F.-L.P., J.-L.G.-S. and M.-Á.P.-T.

**Funding:** This research is partly financed by the European Union through the European Regional Development Fund (ERDF) Programme: Extremadura Operational Programme 2014–2020, Thematic objective 1: Research and Innovation, and Thematic Objective 2: Information and communication technologies. Ref. 2018.14.02.332A.444.00, CultivData Project. This work was also supported in part by the European Regional Development Fund, and the Regional Ministry of Economy, Science and Digital Agenda of the Junta of Extremadura under project GR18195.

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

## References

1. Raspberry Pi Foundation. Available online: https://www.raspberrypi.org (accessed on 29 September 2019).

2. Khan, T. A Deep Learning Model for Snoring Detection and Vibration Notification Using a Smart Wearable Gadget. *Electronics* **2019**, *8*, 987. doi:10.3390/electronics8090987.
3. Alarcón-Paredes, A.; Francisco-García, V.; Guzmán-Guzmán, I.P.; Cantillo-Negrete, J.; Cuevas-Valencia, R.E.; Alonso-Silverio, G.A. An IoT-Based Non-Invasive Glucose Level Monitoring System Using Raspberry Pi. *Appl. Sci.* **2019**, *9*, 46. doi:10.3390/app9153046.
4. Aghenta, L.O.; Iqbal, M.T. Low-Cost, Open Source IoT-Based SCADA System Design Using Thinger.IO and ESP32 Thing. *Electronics* **2019**, *8*, 822. doi:10.3390/electronics8080822.
5. Ambrož, M.; Hudomalj, U.; Marinšek, A.; Kamnik, R. Raspberry Pi-Based Low-Cost Connected Device for Assessing Road Surface Friction. *Electronics* **2019**, *8*, 341. doi:10.3390/electronics8030341.
6. Ali, A.S.; Coté, C.; Heidarinejad, M.; Stephens, B. Elemental: An Open-Source Wireless Hardware and Software Platform for Building Energy and Indoor Environmental Monitoring and Control. *Sensors* **2019**, *19*, 17. doi:10.3390/s19184017.
7. Thomas, A.; Hedley, J. FumeBot: A Deep Convolutional Neural Network Controlled Robot. *Robotics* **2019**, *8*, 62.
8. Sobota, J.; PiŜl, R.; Balda, P.; Schlegel, M. Raspberry Pi and Arduino boards in control education. *IFAC Proc. Vol.* **2013**, *46*, 7–12, doi:10.3182/20130828-3-UK-2039.00003.
9. Urban, P.L. Prototyping Instruments for the Chemical Laboratory Using Inexpensive Electronic Modules. *Angew. Chem. Int. Ed.* **2018**, *57*, 11074–11077, doi:10.1002/anie.201803878.
10. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandes, J.P.; Saraiva, J. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017), Vancouver, BC, Canada, 23–27 October 2017; ACM: New York, NY, USA, 2017; pp. 256–267, doi:10.1145/3136014.3136031.
11. Corral-García, J.; González-Sánchez, J.L.; Pérez-Toledano, M.A. Evaluation of Strategies for the Development of Efficient Code for Raspberry Pi Devices. *Sensors* **2018**, *18*, 4066. doi:10.3390/s18114066.
12. GCC, the GNU Compiler Collection. Available online: https://gcc.gnu.org (accessed on 29 September 2019).
13. Dawson-Haggerty, S.; Krioukov, A.; Culler, D.E. *Power Optimization—A Reality Check*; Tech. Rep. UCB/EECS-2009-140; EECS Department, University of California: Berkeley, CA, USA, 2009.
14. Abdulsalam, S.; Lakomski, D.; Gu, Q.; Jin, T.; Zong, Z. Program energy efficiency: The impact of language, compiler and implementation choices. In Proceedings of the International Green Computing Conference, Dallas, TX, USA, 3–5 November 2014; pp. 1–6, doi:10.1109/IGCC.2014.7039169.
15. Shore, C. Efficient C Code for ARM Devices. In Proceedings of the ARM Technology Conference, Santa Clara, CA, USA, 9–11 November, 2010; pp. 1–14.
16. Mantovani, F.; Calore, E. Performance and Power Analysis of HPC Workloads on Heterogeneous Multi-Node Clusters. *J. Low Power Electron. Appl.* **2018**, *8*, 13. doi:10.3390/jlpea8020013.
17. Calore, E.; Mantovani, F.; Ruiz, D. Advanced Performance Analysis of HPC Workloads on Cavium ThunderX. In Proceedings of the 2018 International Conference on High Performance Computing Simulation (HPCS), Orleans, France, 16–20 July 2018; pp. 375–382, doi:10.1109/HPCS.2018.00068.
18. Fog, A. *Optimizing Software in C++—An Optimization Guide for Windows, Linux and Mac Platforms*; Copenhagen University College of Engineering: Ballerup, Denmark, 2018.
19. Sloss, A.; Symes, D.; Wright, C. *ARM System Developer's Guide: Designing and Optimizing System Software*; Elsevier: Amsterdam, The Netherlands, 2004.
20. Goldthwaite, L. *Technical Report on C++ Performance*; ISO/IEC PDTR 18015; ISO: Geneva, Switzerland, 2006.
21. Guntheroth, K. *Optimized C++: Proven Techniques for Heightened Performance*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016.
22. Malviya, N.; Khunteta, A. Code Optimization using Code Purifier. *Int. J. Comput. Sci. Inf. Technol.* **2015**, *6*, 4753–4757.
23. Gupta, N.; Seth, N.; Verma, P. Optimal Code Compiling in C. *Int. J. Comput. Sci. Inf. Technol.* **2015**, *6*, 2050–2057.
24. Cooper, K.D.; Mckinley, K.S.; Torczon, L. Compiler-Based Code-Improvement Techniques. 1998. Available online: http://www.cs.tufts.edu/~nr/cs257/archive/keith-cooper/survey.pdf (accessed on 29 September 2019).
25. Lee, M.E. Optimization of Computer Programs in C. Available online: http://leto.net/docs/C-optimization.php (accessed on 29 September 2019).

26. Ghosh, K. Writing Efficient C and C Code Optimization. Available online: https://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization (accessed on 29 September 2019).
27. Isensee, P. C++ Optimization Strategies and Techniques. Available online: http://www.tantalon.com/pete/cppopt/main.htm (accessed on 29 September 2019).
28. Kim, D.; Hong, J.E.; Yoon, I.; Lee, S.H. Code refactoring techniques for reducing energy consumption in embedded computing environment. *Clust. Comput.* **2018**, *21*, 1079–1095. doi:10.1007/s10586-016-0691-5.
29. Park, J.J.; Hong, J.E.; Lee, S.H. Investigation for Software Power Consumption of Code Refactoring Techniques. In Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering, Vancouver, BC, Canada, 1–3 July 2014; pp. 717–722.
30. Gottschalk, M.; Jelschen, J.; Winter, A. Energy-efficient code by refactoring. *Softwaretechnik-Trends* **2013**, *33*, doi:10.1007/s40568-013-0030-4.
31. Mont Blanc Project. Available online: https://www.montblanc-project.eu (accessed on 29 September 2019).
32. Yokoyama, D.; Schulze, B.; Borges, F.; Mc Evoy, G. The survey on ARM processors for HPC. *J. Supercomput.* **2019**, *75*, 7003–7036. doi:10.1007/s11227-019-02911-9.
33. Calore, E.; Schifano, S.F.; Tripiccione, R. Energy-Performance Tradeoffs for HPC Applications on Low Power Processors. In *Euro-Par 2015: Parallel Processing Workshops*; Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 737–748.
34. Jubertie, S.; Melin, E.; Raliravaka, N.; Bodèle, E.; Bocanegra, P.E. Impact of Vectorization and Multithreading on Performance and Energy Consumption on Jetson Boards. In Proceedings of the 2018 International Conference on High Performance Computing Simulation (HPCS), Orleans, France, 16–20 July 2018; pp. 276–283, doi:10.1109/HPCS.2018.00055.
35. Ardito, L.; Torchiano, M. Creating and evaluating a software power model for linux single board computers. In Proceedings of the 6th International Workshop on Green and Sustainable Software, Gothenburg, Sweden, 27 May 2018; pp. 1–8.
36. Kaup, F.; Gottschling, P.; Hausheer, D. PowerPi: Measuring and modeling the power consumption of the Raspberry Pi. In Proceedings of the 2014 IEEE 39th Conference on Local Computer Networks (LCN), Edmonton, AB, Canada, 8–11 September 2014; pp. 236–243.
37. Bekaroo, G.; Santokhee, A. Power consumption of the Raspberry Pi: A comparative analysis. In Proceedings of the IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech), Balaclava, Mauritius, 3–6 August 2016; pp. 361–366.
38. Qureshi, B.; Koubaa, A. On Energy Efficiency and Performance Evaluation of Single Board Computer Based Clusters: A Hadoop Case Study. *Electronics* **2019**, *8*, 182. doi:10.3390/electronics8020182.
39. He, Q.; Weaver, V.; Segee, B. Comparing Power and Energy Usage for Scientific Calculation with and without GPU Acceleration on a Raspberry Pi Model B+ and 3B. In Proceedings on the International Conference on Internet Computing (ICOMP), Las Vegas, NV, USA, 30 July–2 August 2018; The Steering Committee of The World Congress in Computer Science, Computer: San Diego, CA, USA; pp. 3–9.
40. Nakhkash, M.R.; Gia, T.N.; Azimi, I.; Anzanpour, A.; Rahmani, A.M.; Liljeberg, P. Analysis of Performance and Energy Consumption of Wearable Devices and Mobile Gateways in IoT Applications. In Proceedings of the International Conference on Omni-Layer Intelligent Systems, Crete, Greece, 5–7 May 2019; pp. 68–73.
41. Chávez, F.; de Vega, F.F.; Díaz, J.; García, J.; Rodriguez, F.; Castillo, P.A. Energy-consumption prediction of genetic programming algorithms using a fuzzy rule-based system. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, Kyoto, Japan, 15–19 July 2018; pp. 9–10.
42. Toldinas, J.; Lozinskis, B.; Baranauskas, E.; Dobrovolskis, A. MQTT Quality of Service versus Energy Consumption. In Proceedings of the 2019 23rd International Conference Electronics, Palanga, Lithuania, 17–19 June 2019; pp. 1–4.
43. Oma, R.; Nakamura, S.; Enokido, T.; Takizawa, M. An Energy-Efficient Model of Fog and Device Nodes in IoT. In Proceedings of the 2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), Krakow, Poland, 16–18 May 2018; pp. 301–306, doi:10.1109/WAINA.2018.00102.
44. Yosuf, B.; Musa, M.; Elgorashi, T.; Lawey, A.Q.; Elmirghani, J.M. Energy Efficient Service Distribution in Internet of Things. In Proceedings of the 2018 20th International Conference on Transparent Optical Networks (ICTON), Bucharest, Romania, 1–5 July 2018; pp. 1–4.

45. Kaup, F.; Hacker, S.; Mentzendorff, E.; Meurisch, C.; Hausheer, D. The Progress of the Energy-Efficiency of Single-Board Computers. Tech. Rep. NetSys-TR-2018-01. Otto-von-Guericke-University, Institute for Intelligent Cooperative Systems. 2018. Available online:https://www.netsys.ovgu.de/netsys_media/publications/NetSys_TR_2018_01.pdf (accessed on 29 September 2019).
46. Source Code of the Tests Developed for the Manuscript Evaluation of Strategies for the Development of Efficient Code for Raspberry Pi Devices. Available online: http://www.mdpi.com/1424-8220/18/11/4066/s1 (accessed on 29 September 2019).
47. Arm Cortex-A72 CPU. Available online: https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a72 (accessed on 29 September 2019).
48. Baccarelli, E.; Naranjo, P.G.V.; Scarpiniti, M.; Shojafar, M.; Abawajy, J.H. Fog of Everything: Energy-Efficient Networked Computing Architectures, Research Challenges, and a Case Study. *IEEE Access* **2017**, *5*, 9882–9910. doi:10.1109/ACCESS.2017.2702013.