Energy-Delay Investigation of Remote Inter-Process Communication Technologies

Stefanos Georgiou*a*, Diomidis Spinellis*a*

aDepartment of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athina 10434

ARTICLE INFO

ABSTRACT

Most modern information technology devices use the Internet for creating, reading, updating, and deleting shared data through remote inter-process communication (IPC). To evaluate the energy consumption of IPC technologies and the corresponding run-time performance implications, we performed an empirical study on popular IPC systems implemented in Go, Java, JavaScript, Python, PHP, Ruby, and C#. We performed our experiments on computer platforms equipped with Intel and ARM processors. We observed that JavaScript and Go implementations of gRPC offer the lowest energy consumption and execution time. Furthermore, by analysing their system call traces, we found that inefficient use of system calls can contribute to increased energy consumption and poor execution time.

1. Introduction

The energy consumption,[1] for the IT-related products, is an evergrowing matter that has caught the attention of academic researchers and industry. This is primarily due to the increasing costs, as IT-related energy consumption is estimated to reach 15% of the world's total by 2020 [42]. Environmental impact is another major concern, as IT's total greenhouse gas emissions are expected to reach 2.3% by the same year [11]. Energy consumption of IT systems is particularly important in two areas. First, the data centres, one of the vital contributors of IT sector's global energy consumption and greenhouse gas emissions. These are housing large number of server nodes communicating with clients through energy-intensive remote inter-process communication (IPC) technologies. Second, the blossoming field of IoT, where low energy performance is critical, has multiple embedded devices connected with hyper-physical systems to exchange, share, and transmit data. To this end, providing sustainable solutions, by reducing energy consumption, to ensure data centres' and IoT infrastructures environmental sustainability and business growth is of paramount importance.

Researchers have carried out studies on different aspects and granularity of software artifacts to investigate the energy consumption of data structures [32, 12, 27, 28, 44], different programming languages [26, 3, 35, 18], multi-threaded applications [31, 29, 30], and coding practices [41, 19, 37, 39, 33]. In terms of remote IPC technologies, prior work [13, 8, 7, 22, 23] focused on investigating the energy consumption and run-time performance of smart phones and embedded systems on Java implementations for remote IPC such as RPC, REST, SOAP, and WebSockets. However, IPC technologies have not been investigated in terms of energy consumption and run-time performance for different programming language implementations.

In this work, we research computer platforms equipped with Intel and ARM processors using three different IPC technologies available in Java, JavaScript, Go, Python, PHP, Ruby, and C#. We try to identify which programming language and IPC technology implementations offer the best energy and run-time performance when invoking remote procedures. Furthermore, we focus on pointing out the reasons behind our results to help software developers, specifically those concerned with IPC library development, build more energy and run-time performance-efficient implementations.

To accomplish this, we perform an empirical study on the selected computer systems on seven popular programming languages that offer implementations of three well known remote IPC technologies and investigate their energy and run-time performance cost. Our results highlight the efficiency of different implementations and libraries. We also examine whether the energy consumption of IPC technologies is proportional to the run-time performance or the systems' resource usage.

Results reveal that JavaScript and Go implementations of gRPC offer the most energy-efficient and best run-time performance implementations among the considered IPC technologies, while Ruby, PHP, Python, Java, and C# perform most inefficiently, for the most cases. We also found that the energy consumption and run-time performance is not pro-

[1]Although in the physical sense energy cannot be consumed, we will use the terms energy "consumption", "requirement", and "usage" to refer to the conversion of electrical energy by ICT equipment into thermal energy dissipation to the environment. Correspondingly, we will use the term energy "savings", "reduction", "efficiency", and "optimisation" to refer to reduced consumption.

portional for all the examined IPC technology implementations. Besides, from the extracted system call traces, we were able to name certain misuse cases of computer resources that can contribute to higher energy demands and lower run-time performance.

This work is organised as follows. Section 3 presents our research methods where we discuss the research questions, the selected subject systems, our experimental approach, and the threats to validity. Section 4 presents the results based on the programming language and the IPC technologies that achieved the best performance with respect to the energy consumption, execution time, system calls, and resource usage. Section 2 discusses prior work and compares it to this work. Finally, we conclude in Section 5 and discuss future research directions.

2. Related Work

To the best of our knowledge, this is the first work investigating the energy consumption, run–time performance, and resource usage implications of different IPC technologies implemented in various programming languages. Moreover, prior work focuses on Java implementations and libraries of the IPC technologies, while we also investigate energy implications on Go, Python, and JavaScript, C#, PHP, and Ruby. Furthermore, most of the existing studies focus on assessing the energy consumption of IPC technologies of smart phones and embedded systems, while we also explore platforms equipped with Intel processors. We briefly overview related work associated with energy efficiency on: i) various fields of software engineering, ii) smart phones and embedded systems IPC, and iii) programming languages.

2.1. Various Studies of Software Engineering

In general, researchers investigated the energy efficiency on various practices of software engineering. For instance, Sahin et al. [37, 39] examined how code obfuscation and refactoring can affect energy and run-time performance. Pereira et al. [27], Pinto et al. [32], and Pinto et al. [30] performed empirical studies to identify which data structures can offer the most energy savings for Java applications, while Lima et al. [18] performed a similar study using Haskell. Oliveira et al. [44] and Pereira et al. [28] suggested refactoring tools that can identify energy inefficient data structures in Java applications and suggest changes. Aggarwal et al. [17] examined the energy implications of application system calls. Specifically, they showed while the number of system calls between two versions of an application is changing, this will affect its energy consumption.

2.2. Smart Phones and Embedded Systems IPC

In the context of smart phones, Herwig et al. [13], Chamas et al. [8], Boven and Hennebert [7], and Mizouni et al. [22] performed experimental studies to identify the energy consumption of popular IPC technologies (see Table 1). More specifically, Chamas et al. performed an experiment where they investigated the energy consumption of three sorting algorithms (bubble, insertion, heap sort) using

three different input sizes (1.000, 10.000, and 100.000), performed locally (on an Android phone) and remotely (server offloading) by using REST, SOAP, WebSocket, and gRPC IPC technologies. They showed that i) the size of data indeed affects energy consumption, ii) the complexity of the sorting algorithms significantly affects energy consumption, iii) local execution can save more energy than remote for small input sizes, and iv) REST and SOAP are the most energy efficient architecture styles.

Mizouni et al. investigated the energy consumption and run-time performance of SOAP and RESTful web services. They showed that a RESTful web service not only has 10% lower energy consumption against SOAP but has also 30% better run-time performance. Bovet and Hennebert compared the energy consumption of RESTful and WebSocket web service in the context of smart phones and also showed that RESTful web services are far more energy efficient [7]. Similarly, Herwig et al. investigated the energy consumption of REST and WebSockets by sending and receiving data packets on three different network types *i.e.*, WLAN, 3G, and Edge. They showed that REST consumes more energy viz-a-viz the WebSockets; however, this contradicts Chamas et al., Mizouni et al., and Boven and Hennebert who proved REST as the most energy efficient IPC.

To evaluate the run-time performance and energy consumption of RESTful web services built on two well known frameworks (*i.e.,* Axis2 and CXF), Nunes et al. performed an experimental study in the context of a Raspberry Pi platform [23]. In their experiment, they compared the marshaling and unmarshaling of different message sizes and also different CPU clock frequencies. Their results illustrate that the Axis2 framework can offer efficiency in energy consumption and better run-time performance. Also, they found that CPU overclocking contributes to reduced energy consumption and faster execution time.

In our research we observed that gRPC and RPC implementations are the most energy-efficient for all of the alternatives we examined. These results have been observed because we used different programming languages and computer platforms to perform our experiment. Moreover, gRPC utilises Protocol Buffers 3 and HTTP/2 uses a persistent connection between clients and servers, multiplexing, and headers compression.

2.3. Programming Languages

Few research studies examined energy and run-time performance implications that different programming languages have for various tasks (see Table 2). Pereira et al. conducted an empirical study on 27 programming languages from *The Computer Language Benchmarks Game* and compared them in terms of energy, time, and memory performance [26]. In their study, Pereira et al. showed that compiled, semi-compiled, and interpreted programming languages such as C, Java, and Hack are the most energy-efficient. Georgiou et al. performed an analysis over 14 different programming languages to compare the Energy-Delay-Product of 25 diverse programming tasks from the

Table 1
Related Work on Smart Phones

Source	IPC	Applications	Platforms	Results
Mizouni et al. 2011	SOAP and REST	Various message sizes	Android	REST more energy efficient
Boven and Hennebert 2012	REST and WebSockets	Various message sizes	Android	REST more energy efficient
Nunes et al. 2014	REST	Axis2 and CXF	RPis	Axis more energy efficient
Herwig et al. 2015	REST and WebSockets	Various message sizes	Android	WebSockets are more energy efficient
Chamas et al. 2017	REST, SOAP, WebSockets, gRPC	Bubble, Insertion, and Heap sorting	Android	REST and SOAP are energy efficient

Table 2
Related Work on Programming Languages

Source	PROGRAMMING LANGUAGES	Data-Set	Platforms	Most Energy Efficient
Pereira et al. 2017	C, Pascal, Go, Rust, C++, Fortran, Ada, Java, Chapel, Lisp, Ocaml, Haskell, C#, Swift, PHP, F#, Racke, Hack, Python, JS, Ruby, Dart, TS, Erlang, JRuby, Perl, Lua	The Computer Language Benchmarks Game	Laptop	Compiled: C Semi-Compiled: Java Interpreted: Hack
Georgiou et al. 2018	C, C++, C#, Go, Java, JS, Perl, PHP, Python, R, Ruby, Rust, Swift, VB.NET	Rosetta Code	Server Laptop RPi	Compiled: C Semi-Compiled: C# Interpreted: JS

Rosetta Code Repository and executed them on a RPi, laptop, and server [10]. Their findings show that different programming language implementations are far more Energy-Delay-Product efficient for specific cases. They showed that C, Go, and JavaScript are, on average, the most efficient implementations among the compiled and interpreted ones, respectively.

Likewise, we performed our experiment on various IPC technologies and we found that JavaScript and Go are the programming languages that offer the best results in terms of energy consumption and run-time performance in IPC tasks, that aligns with the study on Rosetta Code [10]. In addition, we collected system call traces and tried to correlate them with the energy consumption to find out what makes particular IPC technologies more energy efficient or inefficient.

3. Methods

In this section, we describe our research work's objectives and we formulate the research questions. Also, we illustrate the experimental approach we followed to address our research questions. In the end, we discuss our research limitations and threats to validity.

3.1. Research Questions

Works carried out in the context of the energy consumption for IPC technologies are limited to Java

implementations executed on smart phones [8, 13] and embedded systems [23]. These have shown that REST implementations offer the best results for these particular devices. However, IPC technologies are often being used heavily in other IT-related contexts such as data-centers and IoT. To this end, we investigate whether the same pattern exists for computer systems equipped with Intel and ARM processors by taking into account seven different programming languages and the platforms' system calls and resource usage. Hereby, we define our research questions as follows:

RQ1. *Which IPC technology implementation offers the most energy and run-time performance efficient results?*—Our objective here is to identify the implications that each IPC technology has on the energy consumption and run-time performance for the selected programming languages. This can help practitioners select among the IPC technologies implementations that offer the most energy- and run-time performance-efficient solutions.

RQ2. *What are the reasons that make certain IPC technologies more energy and run-time performance efficient?*—Here, we investigate under the hood how each of the selected IPC technologies works, by examining the implementations' system calls. This can show us which are the system calls that are mostly used by certain appli-

Table 3
Experimental Setup Parameters

Programming	Compilers & Interpreters			IPC Technology Packages and their Versions				
Languages	Servers	RPis	REST	Version	RPC	Version	gRPC	Version
Go	1.9.4	1.9.4	NET/HTTP	1.9.4	NET/RPC	1.9.4	grpc-go	1.17.0
Java	1.8.0	1.8.0	JAX-RS	2.1.0	JAX-WS	2.1.0	grpc-java	1.17.0
JavaScript	10.4.0	10.4.0	Express	4.16.3	Express–RPC	0.0.4	grpc-node	1.17.0
Python	2.7.14	2.7.14	Flask	1.0.2	Flask–RPC	1.0.2	grpc-python	1.17.0
PHP	7.2.12	7.2.12	Laravel	5.7.15	JSON-RPC	2.0	grpc-php	1.17.0
Ruby	2.5.3p	2.5.3p	Rails	5.1.6	JSON-RPC	2.0	grpc-ruby	1.17.0
C#	4.8.0	4.8.0	ASP.NET	2.1.5	ASP.NET	2.1.5	grpc-csharp	1.17.0

cations and for most of their execution time, thus possibly contributing to increased energy consumption and lower run-time performance.

RQ3. *Is the energy consumption of the* IPC *technologies proportional to their run-time performance or resource usage?*—For this research question, we investigate a conflicting view among researchers: that energy consumption is proportional to run-time performance. We address this issue only in the context of IPC technologies. Moreover, we investigate if energy consumption is in proportion with resource usage such as maximum memory usage, number of page faults, and context switches. This can act as an indication to warn developers regarding their applications and libraries energy consumption.

We answer the above research questions by using a number of metrics. For **RQ1**, we use energy consumption, that is the product of the total power consumed and time for an execute task. In addition, we collect run-time performance measurements, that is the execution time of a task. For **RQ2**, we collect system calls, which are API calls of an application that requests services from the user to kernel space. Finally, we use the measurements of the **RQ1**, the maximum memory set size, context-switches, and page faults to answer the **RQ3**. The maximum memory set size indicates the total memory reserved for task, the number of context-switches shows the amount of times a system requested services from the kernel and the associated overhead, while page faults indicate memory pressure during execution time.

3.2. Subject Systems

Experimental Platform: We performed two experiments to consider different context and environments. For the former, we used two Lenovo ThinkCentre M910t platforms [40], where one was acting as a server and the other as a client. In a similar way to the above, we utilized two Raspberry Pis 3B model (RPi), to simulate an IoT test case. In the context of this study, we will refer to the Lenovo and RPis platforms as Intel and ARM platforms, respectively. To retrieve energy consumption, we utilised an external device, the Watts Up? Pro (WUP) [43]. Also, we used an additional RPi to fetch the energy measurements, from the WUP's internal memory, in real-time with the help of a Linux-based open source utility interface [5]. We followed this approach

to avoid further overhead on the server and client instances that could impact their energy consumption. To collect the run-time performance, we used the Linux `time` command to yield the *wall-time* of our implementations. We used the wall-time because WUP offers coarse-grained measurements for the whole computer platform and not only when a process is utilising the CPU. Figures 1a and 1b depict the platform connectivity between the subject systems.

Programming Languages: For selecting our programming languages, we employed the GitHut.info [16] and PyPL [34] web-pages (December 2018). GitHut.info offers information concerning the popularity of various programming languages by taking into account the GitHub active repositories, total number of pushes, and so on. PyPL administers its ranking based on the frequency by which a programming language tutorial has been searched on Google every month. To this end, we picked the first six most popular and active programming languages according to the GitHut.info and PyPL statistics, which were JavaScript, Java, Python, PHP, Ruby, and C#. In addition, we covered in our selection Go, because it is one of the programming languages that earned the highest popularity the recent years, according to a Tiobe study [2].

IPC Technologies: The selected IPC technologies varied among REST, RPC, and gRPC. REST is a stateless architecture style for distributed systems, widely adopted for offering world-accessible APIs. RPC is a publicly known way of causing a procedure to execute remotely and many organizations have developed APIs for it. Therefore, we selected and used the RPC and REST libraries as shown in the next paragraph. We also studied gRPC, an RPC technology developed by Google that uses Protocol buffers 3 and HTTP/2 to boost its speed and interoperability between services. We selected gRPC since many companies that are using microservices (*e.g.,* Netflix, Cisco, CoreOS) are adopting it in their production. Also, gRPC offers library implementations in diverse programming languages making it a suitable candidate for our empirical study.

Web Frameworks: We used web frameworks to build our end-points for the server and client function; however, for this research, we do not examine their impact. To select them, we employed the HotFrameworks [9], that provides a monthly ranking on web frameworks' popularity based on the numbers of GitHub stars and STACKOVERFLOW tagged questions. For JavaScript, we selected *Express* since the
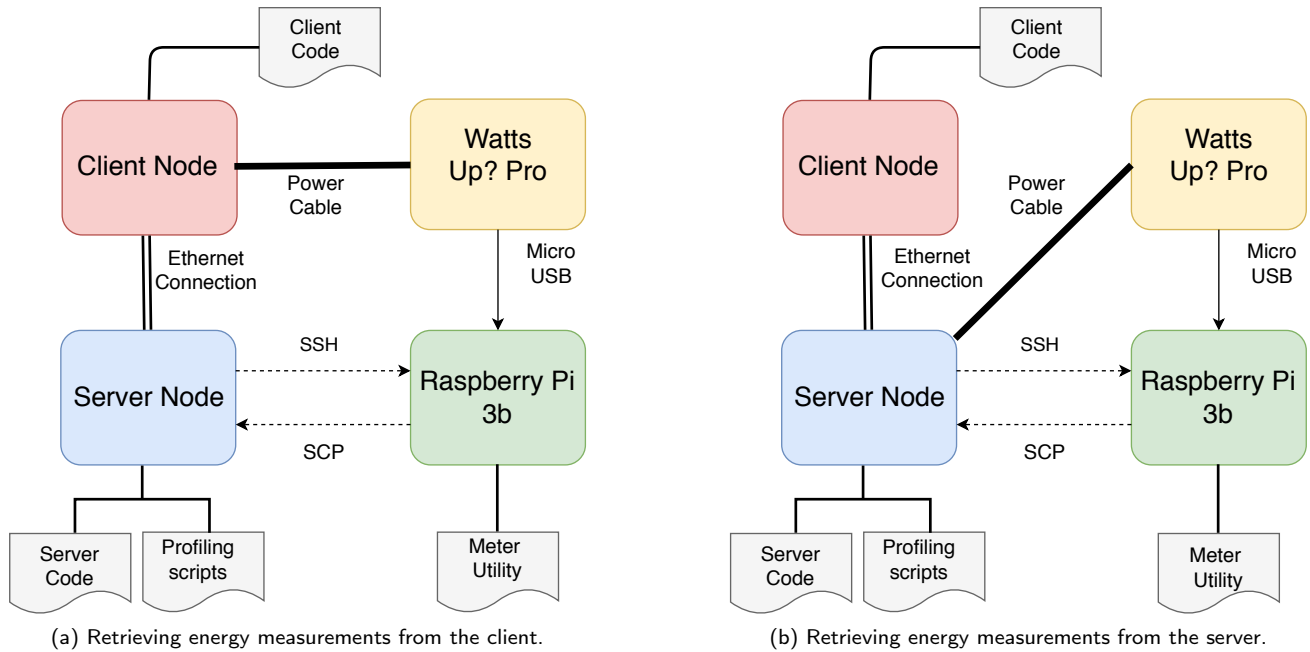
(a) Retrieving energy measurements from the client.

(b) Retrieving energy measurements from the server.

Figure 1: Experiment setup for Intel and ARM systems

AngularJS started losing popularity after 2017 and *React* is mainly used to create user interfaces. Similarly, we excluded *Django* and kept *Flask* for Python. For C#, we used ASP.NET because it is the most influential web framework. Likewise, we selected Ruby on Rails and Laravel for Ruby and PHP to develop the RESTful tasks, respectively. For Ruby and PHP RPC tasks, we could not find any official implementation of Laravel and Ruby on Rails; therefore, we wrote the RPC tasks using the JSON-RPC 2 library. For Go, we selected its built-in packages since the HotFrameworks does not offer any in its ranking. We selected JAX-RS and JAX-WS for Java since the *Spring* is mostly used for RESTful applications. For gRPC, we utilized the latest available versions for each of the selected programming languages which are publicly available on GitHub.[2]

Test Case: To perform our experiment, we either used existing or developed missing `HelloWorld` examples that are making use of the three IPC technologies discussed in this section. For gRPC, we could not test a PHP-written server program because currently there is none available.[3] Therefore, we performed the experiment using a JavaScript server code as suggested in the official documentation. Likewise, it was not possible to compile the C#'s gRPC source code for the ARM platform; therefore, we did not execute the C#'s gRPC for the corresponding platform. The reason we selected a straightforward scenario is that we were mainly concerned with examining only the cost of different remote IPC technologies when they are invoking remote procedures. More specifically, the client makes some remote procedure invocations towards the server's `HelloWorld` function, and the

server replies with a *"Hello World"* message. To this end, Table 3 illustrates the selected programming languages, their compiler and interpreter versions, and the used IPC technology packages and versions for each programming language implementation.

Execution Scripts: To control our experiment's workflow, we wrote around 1,800 lines of Unix shell scripts, to automate the execution, data collection, and results plotting process. All scripts are publicly available on GitHub.[4] For executing the tasks, we included the basic function, that makes the remote procedure call, in a loop of 20,000 and 5,000 iterations for the Intel and ARM computer systems, respectively. We took such an action to force the execution time of a task to take over a second. We did this because the WUP performs `power sampling` and reports the collected energy measurements, on a per second basis.

3.3. Research Approach

To perform our experiment, we followed the experimental approach described below.

- We started our computer systems and stopped unnecessary background processes according to suggestions by Hindle [15] and waited for our system to reach a *stable condition i.e.,* where the energy consumption was idle (23 and 1.5 Joules for Intel and ARM processor, respectively).
- Then, we started our execution script that initiates, through SSH, the i) server instance to receive requests, ii) the RPi to retrieve energy consumption measurements from WUP's internal memory, and iii) the client to perform the tests and collect execution time.

---

[2]https://github.com/grpc
[3]https://grpc.io/docs/tutorials/basic/php.html

[4]https://github.com/stefanos1316/Rest_and_RPC_research/scripts

- When any IPC implementation finished with its execution, we left a small window of a minute, using the Linux `sleep` command, to avoid *tail power states* [6] and to allow our device to reach the stable condition before executing the next implementation.
- Once the whole experiment was done, all the data from the nodes (*i.e.,* the client and RPi) were transferred to the server, using the SCP utility, to sanitise the energy measurements from the idle time and plot graphs.

Because we had only a single WUP at our disposal, we executed the above experiment twice, once for measuring the Intel's server instance energy consumption and once for measuring the client's consumption. Afterwards, we performed the same experiment for the ARM platforms. To minimise measurement noise, we performed each experiment 50 times and obtained statistic results such as the standard deviation, mean, and median values of energy consumption and run-time performance. By plotting histograms for each of the programming language IPC implementations, we observed minor variations between their values. To this end, we decided to retrieve and depict as results the median values (shown in Section 4).

3.4. Threats to Validity

Internal validity. Internal validity refers to possible issues of our techniques that can lead to false results and imprecision. Here, we reveal potential sources of such problems.

Having full control over our operating systems' workload and background operations is hard, because, at any time, different daemons may operate. Also, when a task is executing and enters in a waiting state (*e.g.,* due to an I/O operation) the WUP will still record the energy consumption of our computer platform. This could affect our calculations, too.

For the Java's gRPC tasks, we were not able to compile its native extensions on the embedded systems. However, because of the JVM we were able to execute the task on the embedded systems without the need to compile it. Therefore, we are not aware to which extent these fact can affect our results.

External validity. External validity refers to the extent to which the results of our study can be generalised to other programming implementations. Here, we present the limitations of our study.

According to Sahin et al. empirical studies that use real applications show different energy consumption results from studies that use micro-benchmarks (*i.e.,* traditional desktop software) [38]. Admittedly, since our study's results are based on micro-benchmarks, our findings could be different for real software.

Finally, we evaluate the energy consumption and run-time performance of three IPC technology tasks written in seven programming languages, and running on server platforms and embedded systems. Thus, it is currently difficult for us to generalise our arguments for other programming languages or platforms.

4. Results and Discussion

Here we discuss the collected results of the energy consumption and run-time performance that the selected computer platforms achieved for each IPC technology implementation. We also answer our research questions and discuss the outcomes. Finally, we discuss the significance of our measurements.

4.1. RQ1. Which IPC technology implementation offers the most energy and run-time performance efficient results?

We first present the obtained results for the Intel and ARM computer platforms. Specifically, we show which type of IPC technologies and programming language implementations are the most energy- and run-time performance-efficient among our scenarios. Tables 4 and 5 illustrate the median values of the energy consumption (energy in joules) and run-time performance (time in seconds) for a particular programming language. Moreover, we compare the corresponding implementations viz-a-viz the best implementation's results (most efficient case) in the form of the ratio. Also, Figures 2a and 2b present box plots regarding the programming languages results for each IPC technology.

4.1.1. Intel platforms

Comparison among IPC technologies. In the context of Intel platforms, we can see that, for both server and client, the gRPC offers the lowest energy consumption and execution time for all the IPC technology implementations apart from those of Go and PHP (see Table 4). For Go's implementation, gRPC has the highest energy consumption and lowest run-time performance compared to REST and RPC which are making use of the built-in `net rpc` and `http` libraries. The results also present that RPC is the IPC technology that has the next best results, regarding energy consumption and run-time performance, for all the implementations except from Java, while it offers the best results for Go. Also, we can see that REST implementations contribute to the highest energy consumption and lowest run-time performance among the implementations.

Comparison among programming languages. The results of Table 4 show that JavaScript is the programming language that exhibits the best results for all cases. Plus, we can see that Go outputs the second most energy- and run-time performance-efficient results, while C# is following, and last we observe that Java, Python, Ruby, and PHP offer the lowest performance.

4.1.2. ARM platforms

Comparison among IPC technologies. Likewise to Intel platforms results, the gRPC again contributes to the lowest energy consumption and execution time, for the most cases, among the selected IPC technologies in the context of ARM platforms for the server and client instances. In contrast, Go, Java, and PHP implementations are having the most inefficient results while executing the gRPC task (see Table 5). For RPC scenarios, we observe that our implementations have the

Table 4
Energy and Run-time Performance Results for Intel Platforms

		Nodes' Collected Measurements				Ratio comparison against the most efficient case		
IPC Names	Programming Languages	Energy (in joules) Client	Server	Time (in seconds) Both Nodes	Total Energy Consumption	Energy (in joules) Client	Server	Time (in seconds) Both Nodes
gRPC	Go	99	105.8	28	204.8	3.5	3.9	9.3
	Java	451.5	364	35	815.5	16	13.4	11.6
	JavaScript	**28.2**	**27**	**3**	**55.2**	–	–	–
	Python	606.7	588.7	16	1195.4	21.5	21.8	5.3
	PHP	11 543.9	1291.3	479	12 835.1	409.3	47.8	159.6
	Ruby	139.4	61.6	39	201	4.9	2.2	13
	C#	65.6	105.2	55	170.8	2.3	3.8	18.3
RPC	Go	84.8	70.4	16	45.2	2.3	2.1	1.6
	Java	5224.5	2715.5	855	7940	144.3	84	85.5
	JavaScript	**36.2**	**32.3**	**10**	**26.5**	–	–	–
	Python	319.5	347	73	666.5	8.8	10.7	7.3
	PHP	930.5	1214.6	67	25.7	10.2	37.6	6.7
	Ruby	458.2	513.5	43	971.7	12.6	15.8	4.3
	C#	399.9	364	27	763.9	11	11.2	2.7
REST	Go	94.1	79	19	260.3	1	2.5	1.9
	Java	687	617.8	42	1304.8	7.7	19.6	4.2
	JavaScript	**88.1**	**31.4**	**10**	**109.5**	–	–	–
	Python	637	370	78	1007	7.2	11.7	7.8
	PHP	7003	20 574.5	628	79.4	90	65.2	62.8
	Ruby	1988.2	6191.2	304	8179.4	22.5	197.1	30.4
	C#	1206.6	789.4	44	1996	13.6	25.1	4.4

Table 5
Energy and Run-time Performance Results for ARM Platforms

		Nodes' Collected Measurements				Ratio comparison against the most efficient case		
IPC Names	Programming Languages	Energy (in joules) Client	Server	Time (in seconds) Both Nodes	Total Energy Consumption	Energy (in joules) Client	Server	Time (in seconds) Both Nodes
gRPC	Go	10.8	6.9	10	17.7	4.9	5.3	3.3
	Java	83.5	75.6	71	159.1	45.2	58.1	23.6
	JavaScript	**2.2**	**1.3**	**3**	**3.5**	–	–	–
	Python	11.7	16.5	18	28.2	5.3	12.6	6
	PHP	348.7	88.7	331	437.4	201.2	68.2	110.3
	Ruby	8.3	8.3	12	16.6	3.7	6.3	4
	C#	–	–	–	–	–	–	–
RPC	**Go**	**5.2**	**6.6**	**7**	**11.8**	–	–	–
	Java	73.3	40.6	258	113.9	14	6.1	36.8
	JavaScript	20.9	20.5	22	41.4	4	3.1	3.1
	Python	31	36.5	52	67.5	5.9	5.5	7.4
	PHP	7.7	8.8	28	16.9	1.4	1.3	4
	Ruby	33	35	42	68	6.1	5.3	6
	C#	219.3	290.7	53	510	42.1	44	7.5
REST	**Go**	**4.1**	**3.2**	**8**	**7.3**	–	–	–
	Java	33.9	43.5	22	77.4	8.2	13.5	2.7
	JavaScript	18.3	18.3	22	36.6	4.4	5.7	2.7
	Python	60.4	35	75	95.4	14.7	10.9	9.3
	PHP	175.8	360.1	153	535.9	42.8	112.5	19.1
	Ruby	196	1094	679	1290	47.8	341.8	84.8
	C#	45.3	96.5	68	108.5	11	30.1	8.5

next best results after those of Java and C#. Among the IPC technologies, REST resulted in the least energy- and run-time performance-efficient results for the ARM platforms.

Comparison among programming languages. JavaScript also offers the best results for the ARM platforms when it comes to gRPC task. However, in contrast to the Intel platforms, Go implementations resulted to the most efficient results in terms of energy consumption and run-time performance, while JavaScript is the runner-up for RPC and REST tasks. Compared to the Intel platforms, the presented results for the ARM systems do not depict a clear winner among the remaining language implementations.

4.1.3. Range of results

Figures 2a and 2b illustrate the box plots of the obtained median energy consumption of each implementations with confidence interval of 95%. The points located in our box plots,that are above the maximum values measurements are highly energy inefficient implementations.

In the case of Intel platforms (see Figure 2a), we notice that only the REST implementations diverge significantly among the results, for both client and server instances. For the RPC's implementations, we can observe small differences in the implementations energy consumption, which is even smaller for gRPC results. Also, we observe that some highly inefficient implementations exist for gRPC and REST which are those of PHP's.

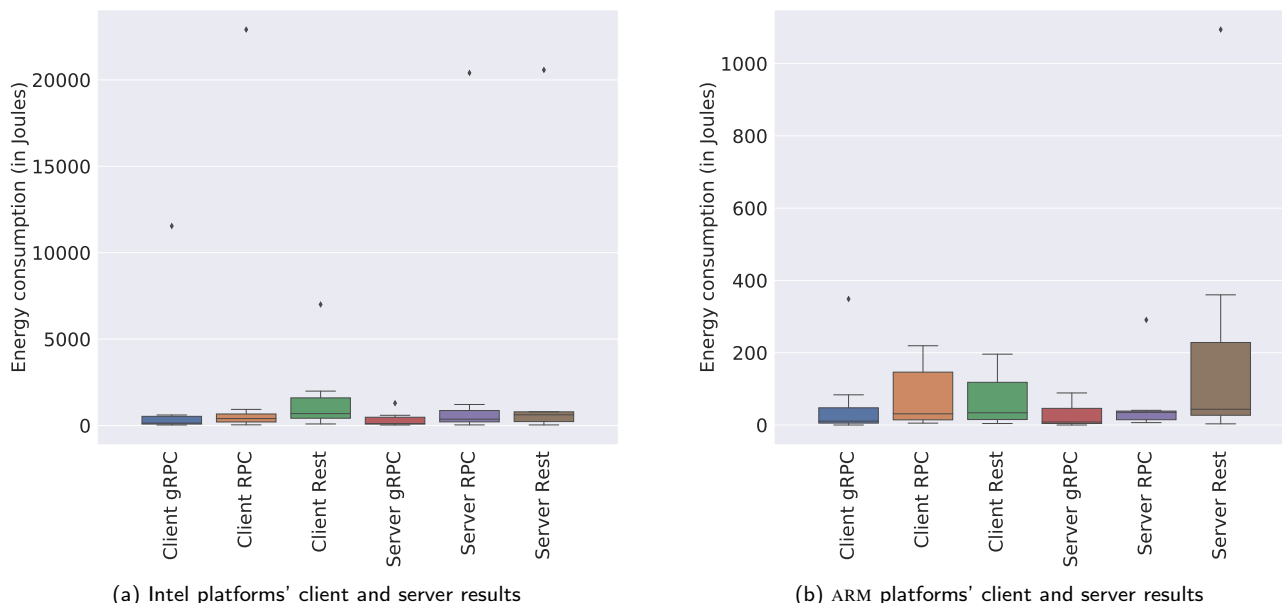In contrast to Intel platforms, for the ARM systems, we

(a) Intel platforms' client and server results

(b) ARM platforms' client and server results

Figure 2: Intel and ARM computer platforms results

see higher divergences in the implementations' energy consumption (see Figure 2b). Such a fact highlights the necessity of proper selection, of IPC protocol and language implementation in the context of embedded systems and battery-restricted devices. Likewise to Intel platforms, PHP implementation for gRPC has the lowest performance. However, for the REST and RPC scenarios, Ruby and C#, respectively, had the most inefficient implementations.

> *JavaScript and Go are the programming language offering the most energy and run-time performance efficient library implementations for the Intel and ARM platforms. In addition, for almost all programming language implementations, we found that gRPC is the IPC technology having the most efficient results.*

4.1.4. Interpreting the Findings

From the collected results, JavaScript emerges as the programming language with the lowest energy consumption and best run-time performance implementation for gRPC. All the gRPC libraries, for all the selected programming languages, are using shared C as their core-library to build their own implementations on top of it. However, JavaScript's gRPC library implementation is using C++ native addons[5] to achieve better performance, a pattern that is not applied for the other implementations. Additionally, Oliveira et al. showed that combining Java or JavaScript applications with native programming languages such as C/C++, can offer up to 100× times less energy consumption and ten times better run-time performance for devices with ARM microprocessor [24]. Also, the studies of Georgiou et al. and Pereira et al. have shown that C++ is among the most energy

efficient programming languages for servers and laptops. Therefore, using C++ with native addons helps JavaScript to reduce energy consumption.

4.1.5. Accuracy of obtained Results

According to Saborido et al. [36], a low sampling rate might miss energy consumption measurements appearing for a short period (energy spikes). Nevertheless, using a device such as the WUP could output imprecise measurements.

To evaluate if the above statement is true and obtain more samples, we performed experiments on the RPis with some JavaScript and Go tasks, that exhibited low energy and run-time performance. To perform these experiments, we had in our disposal: 1) an oscilloscope[6] connected with a current probe[7] and 2) a multimeter.[8] We first connected the multimeter and the current probe on an extension cable live, where a computer system was plugged in. Next, we compared the oscilloscope's current measurements (obtained as input from the current probe) against the multimeter's. We utilized a multimeter (as ground truth for current measurements) to ensure that its measurements aligned with the current probe's. We then made the appropriate settings to the oscilloscope (input voltage, scaling, and true root mean square measurements) according to the current probe's specifications. Also, we measured the average Direct Current (DC) for the tasks execution time by using the oscilloscope's `between rulers` option (see the dashed lines of Figure 3), which allows to obtain measurements for a specified period of time.

Figure 3 depicts the server and client waveforms while executing the gRPC task with a sampling rate of 200,000 units per second. The X-axis illustrates the sampling du-

---

[5]https://nodejs.org/api/addons.html

[6]https://www.picotech.com/oscilloscope/3000
[7]http://www.all-sun.com/EN/d.aspx?pht=1066
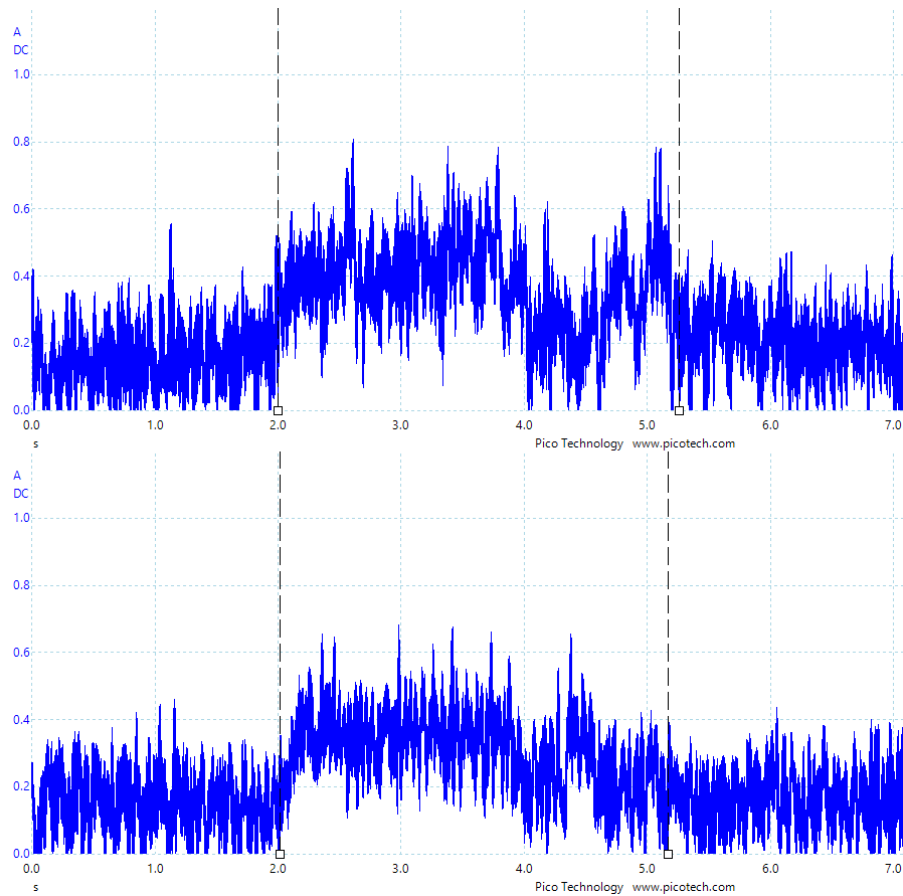[8]https://www.uni-t.cz/en/p/multimeter-uni-t-ut139c

Figure 3: Client (top) and server (bottom) waveforms of gRPC task implemented in JavaScript running on ARM platforms

ration in seconds, while the Y-axis shows the current measurements in Amperes. We present the results obtained from the waveforms in Table 6. Specifically, we show the related task, the language it was implemented with, the devices used to measure its energy consumption and their accuracy, and the average power consumption for the client and server implementations. For the oscilloscope, we obtained our measurements in current and multiplied it with 5V (RPi's power supply) to obtain power consumption. The results indicate that the WUP's measurements fall slightly shorter than the oscilloscope's (*i.e.,* 2–7.1%). This happens because the oscilloscope exhibits measurements with the precision of millisecond for the tasks execution (*e.g.,* 4.432 seconds), while the WUP captures per second measurements (*e.g,* 4 seconds). Therefore, if a task elapses some milliseconds after the WUP reports energy consumption then the energy consumption until the next second will also be added in the average DC, that is partially idle energy consumption of the RPi.

4.2. RQ2. What are the reasons that make certain IPC technologies more energy and run-time performance efficient?

To answer **RQ2**, we execute the whole experiment one more time to retrieve and analyse system calls from the Intel and ARM platforms. Likewise, Aggarwal et al. [4, 17] investigated how the energy consumption of applications

is changing according to the number of their system calls. They showed when the number of system calls between two applications diverges significantly; it is more likely that the application's energy consumption will differ too. However, what we do here is that we examine the system call traces produced by our IPC technology implementations qualitatively and we try to delineate the reasons behind our results. Therefore, we first analyse the obtained system call traces (for the client and server) and then we try to interpret and discuss our findings.

To collect system call traces, we utilise the `strace` command-line tool and we collect data using the flags `-c` (provides a summary-like output) and `-f` (retrieves child process traces). Due to the large volume of results, we did not include the collected system call traces in this paper; however, they are publicly available in our GITHUB repository.[9]

4.2.1. Platforms System Calls

On the ARM and Intel platforms system calls we identify a large number of wait-like system calls such as `futex`, `waitid`, and so on. By investigating the results, we observe that Go is using the `futex`, `waitid`, and `epoll_wait` system calls extensively; this is not happening for the JavaScript implementations. By reading the official documentation of

---

[9]https://github.com/stefanos1316/Rest_and_RPC_research/arm/syscalls

Table 6
Comparison between Watts Up? Pro and Oscilloscopes Measurements

IPC	Language	Device	Accuracy	Average Server Power (in Watts)	Average Client Power (in Watts)
gRPC	JavaScript	WUP	± 5%	1.94	2.04
		Oscilloscope	± 6%	1.90	2.09
RPC	Go	WUP	± 5%	1.77	2.00
		Oscilloscope	± 6%	1.81	2.13
REST	Go	WUP	± 5%	1.68	1.88
		Oscilloscope	± 6%	1.71	2.02

the IPC technology implementations and we found out that Go, for all the IPC technologies, is using *channels*;[10] a synchronous method to serialize main memory access and increase thread-safety [25]. Therefore, it forces the client to wait for an answer from the server before invoking the next remote procedure. This increases execution time and thereby adds to the energy footprint through the system's fixed energy consumption cost.

Likewise, Python implementation system calls are spread among socket, connect, close, sendto, recvfrom, fcntl, and stat for the REST and RPC technologies. Although Python is not using broadly wait-based system calls, it still has the implementations among with the lowest performance for both energy usage and execution time. In the case of gRPC, Python supports both synchronous and asynchronous methods to interact with the client's and server's stubs. However, in the example, a synchronous method is used and, thus, the futex system call takes up most of the implementation's execution time.

We observe similar behavior, with the above, for C#, Java, Ruby, and PHP implementations. JavaScript, on the other hand, due to its asynchronous nature spends most of its execution time on system calls such as writev, mmap, munmap, read, brk, socket, connect, and less than 20% of its execution time on epoll_ctl.

4.2.2. Identifying the Facts

We execute our experiment again by using the -e flag to sanitise our traces from the wait-like system calls (*e.g.,* futex, wait4). We do that since these system calls indicate that an implementation is not using any computing resource—since it is in a sleeping state—and to diagnose which system calls might impact its energy consumption and execution time. Additionally, we remove traces that are related to the compilation since they do not offer an actual execution of the tasks. Figures 4a and 4b, illustrate the time that each of the implementations spend in kernel space (sys time) against the real time for the associated computer platform and IPC protocol. The *Y*-axis supplies information regarding the total median time (of 50 executions) that IPC implementations spent on system calls during their whole execution, while the *x*-axis shows the relevant programming language implementations.

After obtaining our traces, we compare of the most and

least efficient IPC implementation system calls across each of the programming languages according to the results of subsection 4.1. Also, we performed an intra-language (instead of inter-language) comparison of the IPC implementations to be just with languages supporting only asynchronous or synchronous for the investigated tasks. We did that separately for the programming languages affected heavily from the system calls such as C#, Go, and PHP. Moreover, we analyzed JavaScript's system calls because it offers the most energy- and run-time performance-efficient implementations.

For C#, the results for both platforms suggest that the server instances are way more affected by the system calls against the clients. For the RPC implementations, the sched_yield system calls occupies a major portion of time causing a large number of context switches which degrades the implementations performance. This might also be the reason that places C#'s RPC among the implementations with the poorest energy and run-time performance, especially for the ARM platforms (see Table 5).
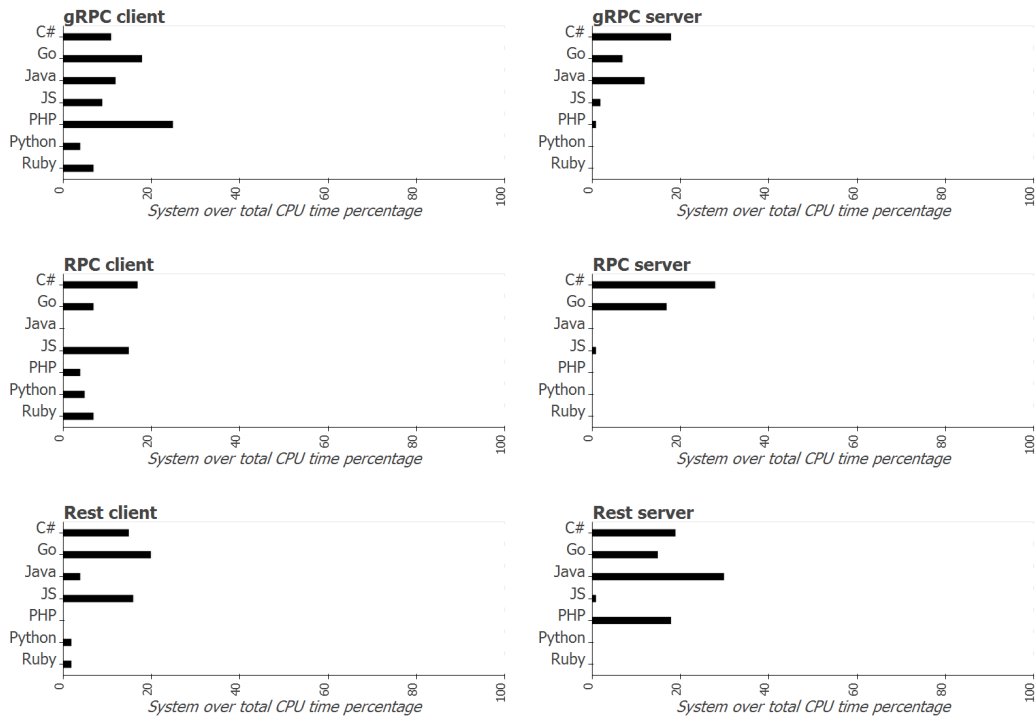
In contrast to C#, Go's client implementations are getting affected more from the system calls, close to 20% of their total execution time. Also, Go's RPC is the most energy- and run-time performance-efficient implementation, while gRPC gives the weakest results. To this line, the system call traces are showing that both of them are using mostly the same system calls *e.g.,* write, read, and sched_yield. However, for the Intel platforms, what makes them different is that RPC makes, in total, at least two times fewer system calls against gRPC. Therefore, by taking into account the work of Aggarwal et al. [4] this can explain the reasons why RPC's implementation is more energy-efficient than gRPC's. For the ARM platforms, the same is not happening since the number of their system calls are similar.

PHP's RPC—which has the best energy and run-time performance among PHP's implementations for the client instances—is mainly using system calls such as connect, close, send, recv, and socket, while the client-side gRPC is extensively using the openat and mmap2 to map data on virtual memory. For the server instances for both platforms, REST implementation for PHP suffers from a great number of system calls.
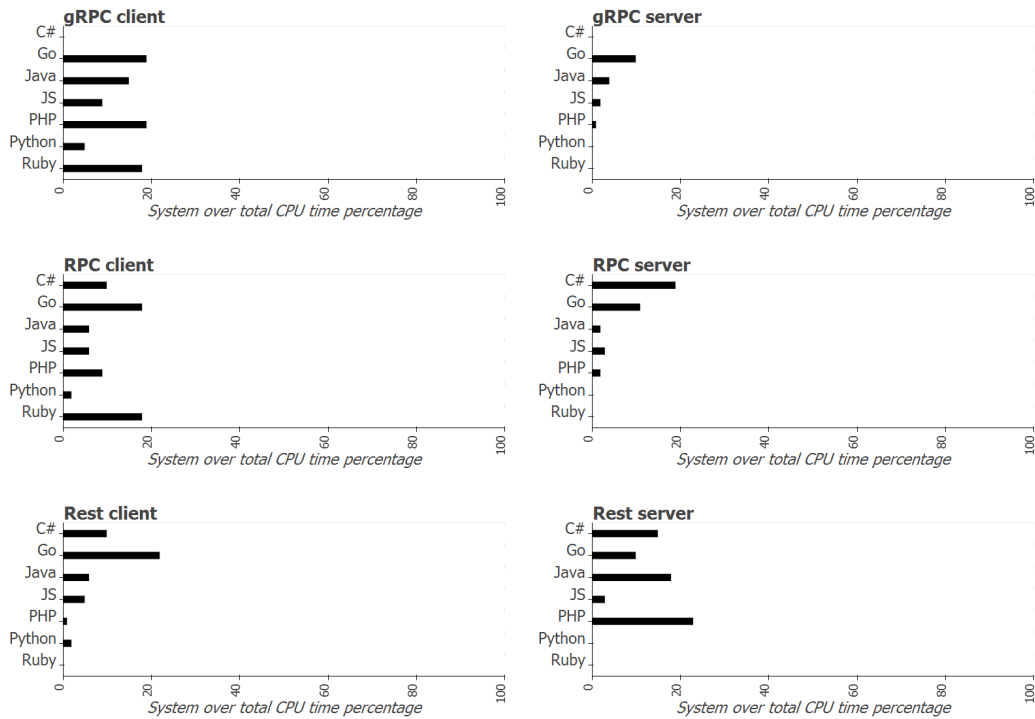
JavaScript that has gRPC as the most energy- and performance-efficient implementations makes broad use of writev system calls that write data into multiple buffers. Also, JavaScript's gRPC is using mostly the read, write,

---

[10]https://gobyexample.com/channels

(a) Intel platforms' system calls impact

(b) ARM platforms' system calls impact

Figure 4: Platforms System Calls

writev, and clock_gettime system calls, while its REST implementation (the most efficient one) utilises considerably the socket, connect, and close system calls for the client and accept and shutdown for the server.

Our analysis shows the frugal opening, connecting, closing, accepting, and shutting down connections can impact the energy consumption and run-time performance of the IPC technologies. Moreover, an extensive number of context switches can severe implementations' performance. Besides, the usage of writev system call appears in the most efficient implementations.

4.2.3. Lessons learned

Each of the selected implementations is making different use of the system calls in their lower level of abstraction. Employing tools such as `strace` to investigate them can provide hints to identify reasons behind the increased energy consumption and the poor run-time performance. We have also observed that JavaScript had the least wait-like systems calls because Node.js—JavaScript's server-side run-time environment—uses an event-driven, asynchronous model. Therefore, JavaScript's server program is not waiting a function's or an API's return data to start serving another request. However, when a function or API call returns the request's data, Node.js uses a notification mechanism to send immediately back data to the client. Therefore, compared to all the other implementations, Node.js fully utilizes its execution time to serve client requests through its asynchronous nature and having less wait-like system calls during execution time.

By examining only the system calls of IPC implementations, we can not always have a clear picture of their energy and run-time performance implications. For instance, Go implementations spend an important amount of time in kernel space; nevertheless, they are among the most energy and run-time performance efficient implementations. This suggests that the type of system calls can affect the energy and run-time performance of IPC implementations.

4.3. RQ3. Is the energy consumption of the IPC technologies proportional to their run-time performance or resource usage?

In **RQ3**, we aim to identify if the energy consumption of different programming language implementations, in the context of IPC technologies, have proportional i) run-time performance and ii) resource usage. To this line, a similar research question to ours was answered by Pereira et al. [26] where they compared the energy consumption, run-time performance, and memory usage of 27 different programming languages and showed that energy consumption is not proportional to the memory usage, while in some cases it is proportional to the execution time. Therefore, we perform a similar experiment in the context of IPC technologies and we examine resource usage as illustrated below. To answer **RQ3**, we break it down into two sub-questions and we answer separately as follows.

4.3.1. Energy consumption and run-time performance

Initially, we collected the median values of the energy consumption and run-time performance of each implementation from our experiment as shown in Tables 4 and 5 and we rendered scatter plots. This offers a complete picture regarding the proportionality of measurements for each of our platforms. In Figures 5a and 5b the X-axis presents the IPC language implementations, while the Y-axis depicts the median energy consumption, for the server or client instance, and their run-time performance.

From the outcome, we observe cases where lower execution time is not associated with reduced energy consumption.

For instance, for the Intel platform, the C#'s REST client and server implementations have execution time 1.2 times lower than the C#'s gRPC; however, the REST implementations energy consumption is 7.5 times higher than the gRPC's. Similarly, Java's RPC implementations are 1.3 times slower than PHP's REST; nevertheless, PHP's client and server consume 1.3 to 7.5 times more energy, respectively. Also, Ruby's REST server uses more energy than PHP's gRPC server but still Ruby's implementation is 1.5 times faster. Go's server and client RPC instances have the same execution time as Python's server and client; however, Python's implementations consume almost three times more energy (see Figure 5a).

For the ARM platforms a similar behavior is observed with the Intel's platforms. Some cases depict that energy consumption is proportional to execution time, while other not (see Figure 5b). For example, C#'s client and server implementations for RPC results to 1.2 times lower execution time than C#'s REST implementations. However, C#'s REST server and client instances are 4.8 and 3 times, respectively, more energy-efficient than C#'s RPC. Additionally, C#'s RPC server and client implementations are 1.3 times faster than Java's gRPC server and client; nevertheless, Java's implementations are 2.6–3.8 times more energy-efficient than C#'s RPC implementations. Also, PHP's REST implementations are 1.6 times faster than Java's RPC; but, Java's implementation uses 2.3 and 8.8 times less energy than PHP REST client and server, respectively.

> *In the context of IPC technologies, almost all the selected programming language implementations have proportional median values for the energy consumption and run-time performance. However, we found many cases where fast execution time did not resulted to energy savings.*

4.3.2. Energy consumption and resource usage

In this sub-research question, we are trying to see whenever the resource usage of the different implementations is proportional to the energy consumption. To this line, we used the Linux `/usr/bin/time -v` command (version 1.9), which offers information regarding the implementations' resource use, such as context switching, page faults, and main memory usage. In this way, we can have a deeper understanding of the way each implementation is allocating memory and different operations that are causing peculiar system calls.

Tables 7 and 8 illustrate the Maximum Resident Set Size (MRSS), the Minor Page Faults (MPF), and the Voluntary Context Switches (VCS) for the client and server, respectively. We show the peak main memory usage for our implementations' processes by using the MRSS that is showing the portion of memory occupied by a process in the main memory. Additionally, we select the MPF since it can show the number of cases when our implementations are trying to access particular memory pages that are not currently mapped in the virtual address space. Also, we ignored the Major

(a) Intel platforms' Energy and Run-Time Performance Comparison

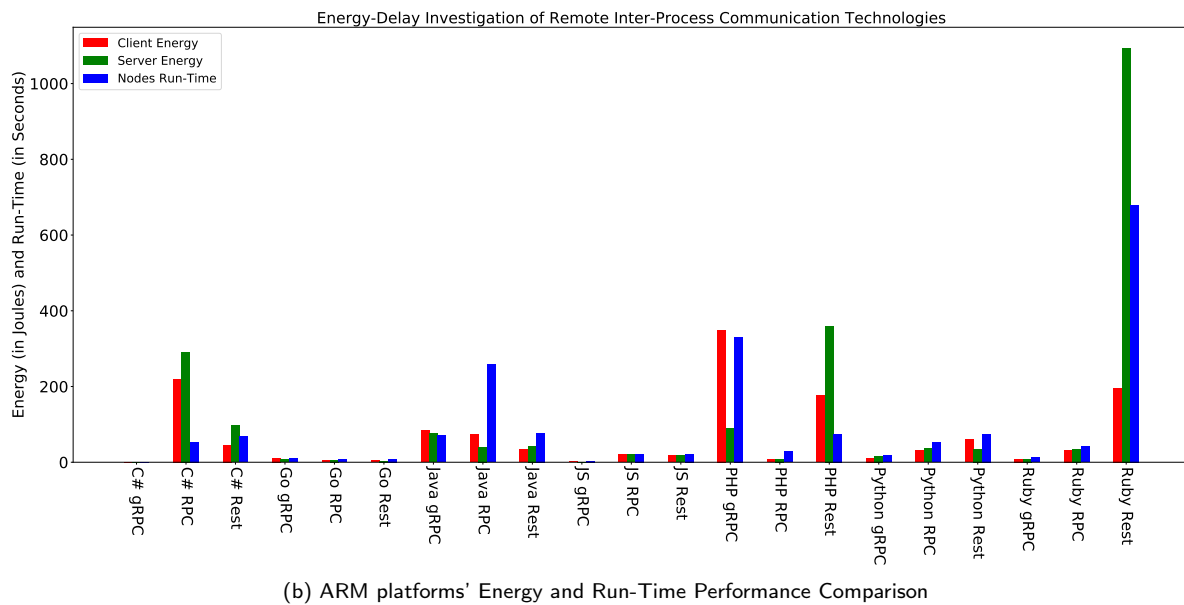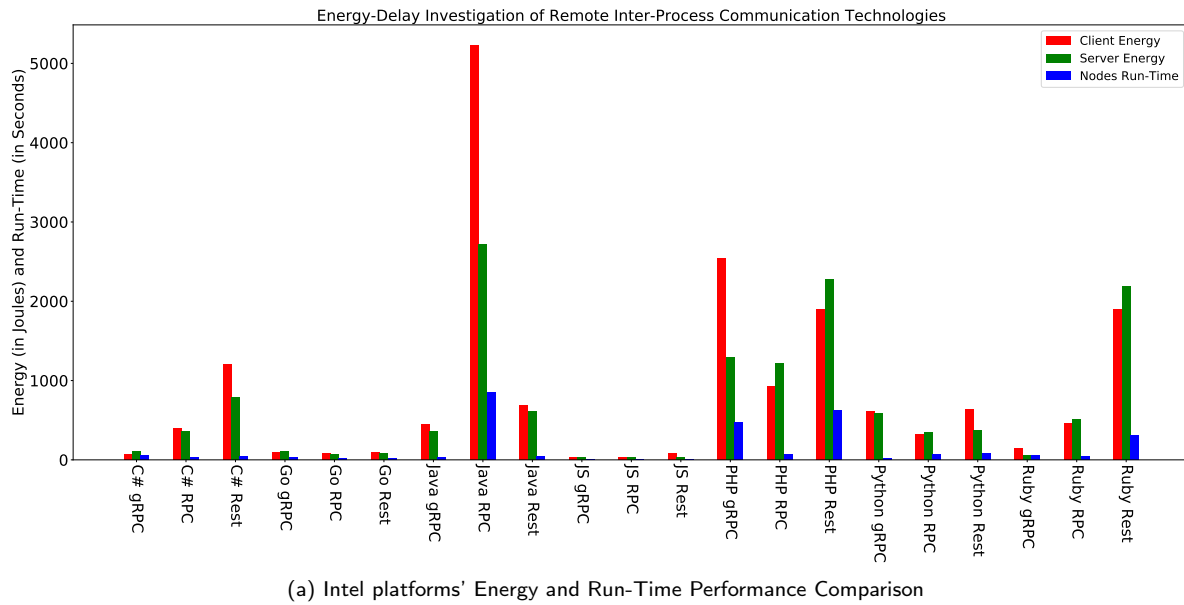(b) ARM platforms' Energy and Run-Time Performance Comparison

Figure 5: Platforms Energy and Run-Time Performance Comparison

Page Faults because only a few instances occurred in our results. Finally, we use the VCS to observe the number of times an implementation's processes were context-switched while waiting for resources that were unavailable at that time. For the sake of simplicity, results reported in Tables 7 and 8 have been divided by 1,000 in order to improve readability.

For the Intel platforms we observed, for the gRPC and RPC, that the most energy- and performance-efficient implementations which are C#, Go, Java, and JavaScript tend to have high MRSS, while PHP, Python, and Ruby have the lowest MRSS and overall performance among the implementations (see Table 7). We observe similar behavior for the REST implementations apart from Ruby that has the highest MRSS but still the poorest performance. Regarding the MPF metrics we found a significant divergence within the programming language implementations. Therefore, finding an association between our results, it is not possible. Likewise to MPF, the number of VCSs do not affect the energy efficiency of the selected IPC implementations. We identified some of the most energy-efficient implementations having a very low VCS, while the same holds for inefficient implementations.

For the ARM platforms we observed for the MRSS there is not a clear indication regarding the proportionality of memory usage for the most energy-efficient implementations against the least efficient ones. However, in Table 8 we observe that the implementations with high MRSS, such as Go and JavaScript, are the most energy- and run-time performance-efficient. Regarding the MPF and VCS for the ARM platform's implementations, we did not find any association with energy consumption as the results tend to be

Table 7
Intel platforms' MRSS (in KB), MPF, and VCS

IPC	Data Name	Server							Client						
		C#	Go	Java	JS	PHP	Python	Ruby	C#	Go	Java	JS	PHP	Python	Ruby
gRPC	MRSS	92	161	328	357	47	27	23	90	117	329	407	28	26	21
	MPF	81	55	90	86	51	4	5	77	216	95	99	4192	7	5
	VCS	167	137	121	0.5	41	266	42	25	136	125	1	470	20	40
RPC	MRSS	143	124	171	98	22	22	21	102	88	168	467	26	22	26
	MPF	128	45	50	21	1	26	6	83	34	50	113	1	6	6
	VCS	151	86	85	0.6	32	366	76	85	82	65	4	40	59	84
REST	MRSS	213	100	6	98	40	22	487	51	72	142	457	26	19	20
	MPF	114	38	2	21	5208	6	128	179	30	40	110	1	7	5
	VCS	174	121	0.05	0.6	29	215	119	824	149	48	4	45	58	113

Table 8
ARM platforms' MRSS (in KB), MPF, and VCS

IPC	Data Name	Server							Client						
		C#	Go	Java	JS	PHP	Python	Ruby	C#	Go	Java	JS	PHP	Python	Ruby
gRPC	MRSS	0	93	66	79	34	17	12	0	91	66	94	20	16	10
	MPF	0	35	15	17	5	2	17	0	34	15	21	7974	2	16
	VCS	0	78	180	0.08	0.04	84	20	0	66	267	0.1	10	5	10
RPC	MRSS	84	68	32	67	15	18	12	84	69	30	94	20	17	13
	MPF	243	26	6	15	1	5	13	213	26	5	22	1	5	17
	VCS	243	33	4	0.4	8	24	13	108	27	17	3	10	14	26
REST	MRSS	82	57	2	70	31	18	126	49	57	29	96	20	20	14
	MPF	77	25	2	16	15	4	328	46	24	5	22	1	5	2
	VCS	81	37	0.03	0.4	8	5	57	146	50	11	3	18	19	36

sparse among the most and least efficient implementations.

> *Neither the Minor Page Faults nor the Voluntary Context-Switches can be used to justify the energy consumption results in terms of IPC technologies. However, the number of Maximum Resident Set Size tends to be proportional to most of the cases. Therefore, drawing clear conclusions regarding energy consumption and resource usage is challenging.*

4.4. Significance of Measurements

In this section, we demonstrate the significance of our measurements by illustrating the possibilities of energy savings while selecting a particular IPC implementation. Later on, we justify its feasibility for software practitioners to utilise our findings and gain a more energy-conscious development.

According to the work of Hindle [14], even minor optimizations on the energy consumption of smart phones for four million users per hour can result in significant worldwide energy savings equivalent to an American household's monthly power use per hour. According to WEBFX,[11] there are more than 4.5 billion of Facebook posts from various ICT products on a daily basis, where clients use IPC implementations to interact with servers. If we consider that Facebook is built on PHP then we can assume—through rough calculations—that the energy cost of 20,000 post requests through gRPC can cost up to 22,560.9 and 12,834.4 Joules for an Intel server and client, respectively (see Table 4). That is translated to 1.39 and 0.76 Mega-Watt hours for 4.5 billion Facebook daily post requests for a server and client in-

stance, respectively. However, switching to the most energy-friendly implementation of gRPC (*i.e,* JavaScript) that consumes 28.2 and 27 Joules for 20,000 post requests, leads to 0.0017 and 0.0016 Mega-Watt hours for an Intel server and client to server the daily post requests of Facebook, respectively. The PHP IPC implementation of Facebook daily amount of post requests is a bit more than the monthly power use of an American's household [1], while using JavaScript reduces power consumption significantly.

Another challenge here is convincing companies and developers to switch to more eco-friendly implementations, which requires further training and migration to different programming languages. To this end, Meyerovich and Rabkin [21] have identified a set of factors that facilitate language adoption by analyzing a data-set of 200,000 Source-Forge and 590,000 Ohloh projects and by performing multiple survey studies on 1,000–13,000 software practitioners. The results suggest that small companies and software practitioners are willing to switch to a programming language if the latter offers more libraries and better performance. Regarding energy consumption as a performance metric, Manotas et al. [20] have shown in the context of a survey study, that software practitioners are willing to adopt energy-conscious development, and they consider it essential in the context of data centers and mobile devices.

5. Conclusions

We performed an empirical study over diverse remote IPC technologies implemented in different programming languages to appraise their energy and run–time performance. Our results highlight JavaScript's and Go's implementations as the most energy- and run–time performance-efficient

---

[11]https://www.webfx.com/internet-real-time/

compared to PHP, Java, C#, Python, and Ruby. Practitioners can benefit from our study by noting the following.

- Energy consumption and run-time performance can vary significantly among different programming language implementations; therefore, making the right selection of IPC can benefit the applications' energy consumption.
- The use of writev system calls is more energy and run-time performance efficient because it makes, in total, fewer system calls.
- Neither the memory usage nor number of context-switches can indicate the energy or run-time performance of a library's efficiency.

Researchers can build on our study by comparing different web frameworks, using more test cases, and applying our findings in real-world micro-services application is to evaluate their performance.

References

[1] , . How much electricity does an American home use? - FAQ - U.S. Energy Information Administration (EIA). URL: https://www.eia.gov/tools/faqs/faq.php?id=97.

[2] , 2017. TIOBE Index | TIOBE - The Software Quality Company. URL: https://www.tiobe.com/tiobe-index/.

[3] Abdulsalam, S., Lakomski, D., Gu, Q., Jin, T., Zong, Z., 2014. Program energy efficiency: The impact of language, compiler and implementation choices, in: Green Computing Conference (IGCC), 2014 International, pp. 1–6.

[4] Aggarwal, K., Hindle, A., Stroulia, E., 2015. GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 311–320.

[5] Bailey, P., 2017. Watts Up Pro power meter interface utility for Linux. URL: https://github.com/pyrovski/watts-up.

[6] Bornholt, J., Mytkowicz, T., McKinley, K.S., 2012. The model is not enough: Understanding energy consumption in mobile devices, in: 2012 IEEE Hot Chips 24 Symposium (HCS), pp. 1–3.

[7] Bovet, G., Hennebert, J., 2012. Communicating With Things - An Energy Consumption Analysis, in: 2012 IEEE Tenth International Conference on Pervasive Computing (Pervasive '2012).

[8] Chamas, C.L., Cordeiro, D., Eler, M.M., 2017. Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis, in: 2017 IEEE 9th Latin-American Conference on Communications (LATINCOM), pp. 1–6.

[9] Frameworks, H., 2018. Web framework rankings | HotFrameworks. URL: https://hotframeworks.com/.

[10] Georgiou, S., Kechagia, M., Louridas, P., Spinellis, D., 2018. What Are Your Programming Language's Energy-Delay Implications?, in: 15th International Conference on Mining Software Repositories (MSR), ACM, New York, NY, USA. p. 11.

[11] GeSI, 2018. Gesi smarter 2030. URL: http://smarter2030.gesi.org.

[12] Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., Hindle, A., 2016. Energy Profiles of Java Collections Classes, in: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 225–236.

[13] Herwig, V., Fischer, R., Braun, P., 2015. Assessment of REST and WebSocket in regards to their energy consumption for mobile applications, in: 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), pp. 342–347.

[14] Hindle, A., 2015. Green mining: a methodology of relating software change and configuration to power consumption. Empirical Software Engineering 20, 374–409. doi:10.1007/s10664-013-9276-6.

[15] Hindle, A., Wilson, A., Rasmussen, K., Barlow, E.J., Campbell, J.C., Romansky, S., 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. 12–21.

[16] Info, G., 2018. GitHut - Programming Languages and GitHub. URL: http://githut.info/.

[17] Karan Aggarwal, 2014. The Power of System Call Traces: Predicting the Software Energy Impact of Changes. URL: http://archive.org/details/Cascon2014.

[18] Lima, L.G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., Fernandes, J.P., 2019. On Haskell and energy efficiency. Journal of Systems and Software 149, 554–580. doi:10.1016/j.jss.2018.12.014.

[19] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D., 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. 2–11.

[20] Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J., 2016. An Empirical Study of Practitioners' Perspectives on Green Software Engineering, in: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 237–248. doi:10.1145/2884781.2884810.

[21] Meyerovich, L.A., Rabkin, A.S., 2013. Empirical Analysis of Programming Language Adoption, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ACM, New York, NY, USA. pp. 1–18. doi:10.1145/2509136.2509515. event-place: Indianapolis, Indiana, USA.

[22] Mizouni, R., Serhani, M.A., Dssouli, R., Benharref, A., Taleb, I., 2011. Performance Evaluation of Mobile Web Services, in: 2011 IEEE Ninth European Conference on Web Services, pp. 184–191.

[23] Nunes, L.H., Nakamura, L.H.V., Vieira, H.d.F., Libardi, R.M.d.O., Oliveira, E.M.d., Estrella, J.C., Reiff-Marganiec, S., 2014. Performance and energy evaluation of RESTful web services in Raspberry Pi, in: 2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC), pp. 1–9.

[24] Oliveira, W., Oliveira, R., Castor, F., 2017. A Study on the Energy Consumption of Android App Development Approaches, IEEE Press, Piscataway, NJ, USA. pp. 42–52.

[25] org, G., 2019. Go synchronization. URL: https://golang.org/pkg/sync/.

[26] Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J., Saraiva, J., 2017. 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, ACM. pp. 256–267. doi:10.1145/3136014.3136031.

[27] Pereira, R., Couto, M., Saraiva, J., Cunha, J., Fernandes, J.P., 2016. The Influence of the Java Collection Framework on Overall Energy Consumption, in: Proceedings of the 5th International Workshop on Green and Sustainable Software, ACM, New York, NY, USA. pp. 15–21.

[28] Pereira, R., Simão, P., Cunha, J., Saraiva, J., 2018. jStanley: Placing a Green Thumb on Java Collections, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA. pp. 856–859. doi:10.1145/3238147.3240473.

[29] Pinto, G., 2013. Refactoring Multicore Applications Towards Energy Efficiency, in: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, ACM, New York, NY, USA. pp. 61–64.

[30] Pinto, G., Canino, A., Castor, F., Xu, G., Liu, Y.D., 2017. Understanding and overcoming parallelism bottlenecks in ForkJoin applications, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 765–775. doi:10.1109/ASE.2017.8115687.

[31] Pinto, G., Castor, F., Liu, Y.D., 2014. Understanding Energy Behaviors of Thread Management Constructs, in: Proceedings of the

2014 ACM International Conference on Object Oriented Program-
ming Systems Languages & Applications, ACM, New York, NY,
USA. pp. 345–360.

[32] Pinto, G., Liu, K., Castor, F., Liu, 2016. A Comprehensive Study
on the Energy Efficiency of Java Thread-Safe Collections, in: 32nd
IEEE International Conference on Software Maintenance and Evolu-
tion, IEEE Computer Society, Raleigh, North Carolina, USA.

[33] Procaccianti, G., Fernández, H., Lago, P., 2016. Empirical evalua-
tion of two best practices for energy-efficient software development.
Journal of Systems and Software 117, 185–198.

[34] of Programming Language, P.P., 2018. PYPL PopularitY of Program-
ming Language index. URL: http://pypl.github.io/PYPL.html.

[35] Rashid, M., Ardito, L., Torchiano, M., 2015. Energy Consumption
Analysis of Algorithms Implementations, in: 2015 ACM/IEEE Inter-
national Symposium on Empirical Software Engineering and Mea-
surement (ESEM), pp. 1–4.

[36] Saborido, R., Arnaoudova, V., Beltrame, G., Khomh, F., Antoniol, G.,
2015. On the impact of sampling frequency on software energy mea-
surements. PeerJ PrePrints 3, e1219. doi:10.7287/peerj.preprints.
1219v2.

[37] Sahin, C., Pollock, L., Clause, J., 2014a. How Do Code Refactorings
Affect Energy Usage?, in: Proceedings of the 8th ACM/IEEE Inter-
national Symposium on Empirical Software Engineering and Mea-
surement, ACM, New York, NY, USA. pp. 36:1–36:10. doi:10.1145/
2652524.2652538.

[38] Sahin, C., Pollock, L., Clause, J., 2016. From benchmarks to real
apps. Journal of Systems and Software 117, 307–316.

[39] Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., Clause, J., 2014b.
How Does Code Obfuscation Impact Energy Usage?, in: 2014 IEEE
International Conference on Software Maintenance and Evolution
(ICSME), pp. 131–140. doi:10.1109/ICSME.2014.35.

[40] lenovo thinkcentre, 2018. ThinkCentre M910 Tower | Power Your
Business | Lenovo Australia. URL: https://www3.lenovo.com/au/en/
desktops-and-all-in-ones/thinkcentre/.

[41] Tonini, A.R., Fischer, L.M., Mattos, J.C.B.d., Brisolara, L.B.d., 2013.
Analysis and Evaluation of the Android Best Practices Impact on the
Efficiency of Mobile Applications, pp. 157–158.

[42] Van Heddeghem, W., Lambert, S., Lannoo, B., Colle, D., Pickavet,
M., Demeester, P., 2014. Trends in worldwide ICT electricity con-
sumption from 2007 to 2012. Computer Communications 50, 64–76.

[43] WattsUpMeter, 2017. Watts up? Products: Meters. URL: https:
//www.wattsupmeters.com/secure/products.php?pn=0.

[44] Wellington, O., Renato, O., Fernando, C., Benito, F., Gustavo, P.,
2019. Recommending energy-efficient java collections, in: Proceed-
ings of the 16th International Conference on Mining Software Reposi-
tories, MSR 2019, 26-27 May 2019, Montreal, Canada., pp. 160–170.

Stefanos Georgiou is a PhD Candidate in the De-
partment of Management Science and Technology
at the Athens University of Economics and Busi-
ness, Greece. He holds a BSc in Networks and Sys-
tems Programming from the University of Cyprus
and a MSc in PERvasive Computing and COMmu-
nications for sustainable development (PERCCOM).
In his PhD he aims to reduce applications energy
consumption by using software engineering tech-
niques and practices.

Diomidis Spinellis is a Professor in the Depart-
ment of Management Science and Technology at
the Athens University of Economics and Business,
Greece and director of the University's Business
Analytics Laboratory. He is the author of two
award-winning books, *Code Reading* and *Code
Quality: The Open Source Perspective*. His most
recent book is *Effective Debugging: 66 Specific
Ways to Debug Software and Systems*. He has con-
tributed code that ships with Apple's MACOS and
BSD Unix, and is the developer of *CScout*, *UML-
Graph*, *dgsh*, and other open-source software pack-
ages, libraries, and tools. He served as the Editor
in Chief for *IEEE Software* over the period 2015–
2018.