Energyware Analysis

RUI PEREIRA, MARCO COUTO, FRANCISCO RIBEIRO, RUI RUA and JOÃO SARAIVA,
HASLab/INESC TEC & University of Minho

This documents introduces "Energyware" as a software engineering discipline aiming at defining, analyzing and optimizing the energy consumption by software systems. In this paper we present energyware analysis in the context of programming languages, software data structures and program's source code.

For each of these areas we describe the research work done in the context of the Green Software Laboratory at Minho University: we describe energyaware techniques, tools, libraries, and repositories.

1. INTRODUCTION

While in the previous century, language designer's and software engineer's main goals were to develop fast software systems, the current widespread use of non-wired computing devices is making energy consumption a key aspect not only for hardware manufacturers, but also for software developers. Software languages and their compilers provide programmers with powerful mechanisms to increase their productivity: for example, by providing advanced static type systems that reduce runtime software errors while increasing software reuse, and by offering tools that help programmers find errors (debuggers), bad smells (refactoring tools), detecting memory leaks and runtime issues (profilers), etc.

All these mechanisms and tools were developed with the goal of making programming "faster" and programs run "faster". In this document we discuss energyware as an engineering discipline to reason about energy consumption in software systems. We discuss techniques and tools developed in our Green Software Laboratory, namely, techniques to analyze the energy efficiency of 27 programming languages, to detect inefficient energy use of data structures, and to analyze software's source code and locate abnormal energy consumption.

An interesting question that frequently arises in the software energy efficiency area is whether *a faster program is also an energy efficient program*, or not. If the answer is yes, then optimizing a program for speed also means optimizing it for energy, and this is exactly what the compiler construction community has been hardly doing since the very beginning of software languages. However, energy consumption does not depends only on execution time, as shown in the equation $E_{nergy} = T_{ime} \times P_{ower}$.

A program provides a possible implementation for a given computer problem. Such a program is written in a specific programming language, it possibly uses languages data structures available in the

Author's address: Campus de Gualtar, Departamento de Informtica, 4710-057 Braga, Portugal.
Author's emails: ruipereira, mcouto, fribeiro, rrua, saraiva@di.uminho.pt

This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency, FCT - Fundao para a Cilncia e a Tecnologia within project POCI-01-0145-FEDER-016718 and UID/EEA/50014/2013. The first author is also sponsored by FCT grant SFRH/BD/112733/2015.

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

language libraries, and its source code implements a particular algorithm. Because there are many different ways of expressing the same algorithm within a programming language, the source code reflects the programmer's personal coding practices (for example, by preferring loops instead of recursion, for loops instead of while loops, etc). When developing an energy efficient software system, a programmer needs to be energy-aware, where by choosing a particular programming language, data structure and algorithm the resulting program may have very different energy consumptions.

In this document we present the results obtained in defining a green ranking for programming languages 2, by analyzing the energy consumption of the Java Collection Framework (JCF) library and defining an energy-aware refactoring for Java 3, and, finally, by adapting a software's fault localization algorithm to locate abnormal energy consumption (*"energyware faults"*) in the source code of a program 4.

2. ENERGYWARE ANALYSIS IN PROGRAMMING LANGUAGES

The massive use of mobile devices made energy consumption a key bottleneck not only for hardware manufacturers, but also for software developers. When writing efficient software, programmers often ask this question: *is a faster program also an energy efficient program?*.

A similar question arises when comparing software languages: *is a faster language, a greener one?* Comparing software languages, however, is an extremely complex task, since the performance of a language is influenced by the quality of its compiler, virtual machine, garbage collector, available libraries, etc. Indeed, a software program may become faster by improving its source code, but also by "just" optimizing its libraries and/or its compiler.

In the Green Software Language Laboratory we studied the energy efficiency of the most widely used software languages [Couto et al. 2017; Pereira et al. 2017]. We considered an open source repository where the same computer problem is expressed in each of the languages: the Computer Language Benchmark GameW (CLBG)[1]. CLBG includes a repository of programs written in twenty seven languages which implement solutions to a set of ten predefined computing problems. This language benchmark project was developed to provide a runtime ranking of programming languages.

We reused the CLBG infrastructure to define an energy-aware ranking of programming languages: We consider ten different programming problems that are expressed in each of the languages, following the exact same algorithm, as specified by CLBG. We compile/execute such programs using the state-of-the-art compilers, virtual machines, interpreters, and libraries for each of the 27 languages. We developed a framework using an energy measurement tool provided by Intel (RAPL)[2], in order to measure the energy consumption when executing such programs. We used this monitoring framework to measure the energy of all executable programs included in CLBG. Afterwards, we analyze the performance of the different implementation considering three variables: execution time, memory consumption and energy consumption.

Table I contains the normalized results for energy, time and memory obtained by benchmarking the ten CLBG problems. The first column states the name of the programming languages, preceded by either a (c), (i), or (v) classifying them as either a compiled, interpreted, or virtual-machine language, respectively.

By looking at the overall results, shown in Table I, we can see that the top 5 most energy efficient languages keep their rank when they are sorted by execution time and with very small differences in both energy and time values. This does not come as a surprise, since those languages are known to be heavily optimized and efficient for execution performance, as our data also shows. Thus, as time influences

[1] http://benchmarksgame.alioth.debian.org/

[2] https://software.intel.com/en-us/articles/intel-power-governor

Table I. : Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

energy, we had hypothesized that these languages would also produce efficient energy consumptions as they have a large advantage in one of the variables influencing energy, even if they consumed more power on average. If we look at the remaining languages in Table I, we can see that only 4 languages maintain the same energy and time rank (OCaml, Haskel, Racket, and Python), while the remainder are completely shuffled. Additionally, looking at individual benchmarks we see many cases where there is a different order for energy and time.

Additionally, the individual benchmarks included in [Pereira et al. 2017] show that, although the most energy efficient language in each benchmark is almost always the fastest one, the fact is that there is no language which is consistently better than the others: there many cases where there is a different order for energy and time. This allows us to conclude that the situation on which a language is going to be used is a core aspect to determine if that language is the most energy efficient option.

There are many situations where a programmer has to choose a particular programming language to implement his algorithm according to functional or non functional requirements. For instance, if one is developing software for wearables, it is important to choose a language and apply energy-aware techniques to help save battery. Another example is the implementation of tasks that run in background. In this case, execution time may not be a main concern, and they may take longer than the ones related to the user interaction. To this end, we present in Table II a comparison of three language characteristics: energy consumption, execution time, and peak memory usage. In order to compare the languages using more than one characteristic at a time we use a multi-objective optimization algorithm to sort these languages, known as Pareto optimization.

For each ranking, each line represents a Pareto optimal set, that is, a set containing the languages that are equivalent to each other for the underlying objectives. In other words, each line is a single rank or position. The most common performance characteristics of software languages used to evaluate and choose them are execution time and memory usage. If we consider these two characteristics in our

Table II. : Pareto optimal sets for different combination of objectives.

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C Pascal Go	C	C Pascal	C Pascal Go
Rust C++ Fortran	Rust	Rust C++ Fortran Go	Rust C++ Fortran
Ada	C++	Ada	Ada
Java Chapel Lisp Ocaml	Ada	Java Chapel Lisp	Java Chapel Lisp Ocaml
Haskell C#	Java	OCaml Swift Haskell	Swift Haskell C#
Swift PHP	Pascal Chapel	C# PHP	Dart F# Racket Hack PHP
F# Racket Hack Python	Lisp Ocaml Go	Dart F# Racket Hack Python	JavaScript Ruby Python
JavaScript Ruby	Fortran Haskell C#	JavaScript Ruby	TypeScript Erlang
Dart TypeScript Erlang	Swift	TypeScript	Lua JRuby Perl
JRuby Perl	Dart F#	Erlang Lua Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript Hack		
	PHP		
	Erlang		
	Lua JRuby		
	Ruby		

evaluation, `C`, `Pascal`, and `Go` are equivalent. However, if we consider energy and time, `C` is the best solution since it is dominant in both single objectives. If we prefer energy and memory, `C` and `Pascal` constitute the Pareto optimal set. Finally, analyzing all three characteristics, this scenario is very similar as for time and memory. It is interesting to see that, when considering energy and time, the sets are usually reduced to one element. This means, that it is possible to actually decide which is the best language.

More recently we extended the study of energyware programming languages to consider a different program repository: *Rosetta Code*[3]. While CLBG was developed with the single goal of comparing language execution time, *Rosetta Code* was developed with more program comprehension purposes. Our new results show interesting findings, such as [Pereira et al. 2018].

3. ENERGYWARE DATA STRUCTURES

One obvious way to make a programming language more energy, memory and runtime efficient is by defining powerful compiler optimizations. Language libraries, however, also play an important role on the performance of the executable programs: modern programming languages offer a large set of libraries, which are themselves part of language, and are the building blocks when writing their programs. An important set of libraries, offered by most languages and widely reused by their programs, provides advanced mechanisms to manipulate data structures.

In the Green Software laboratory we conducted a detailed study in terms of energy consumption of the widely used *Java Collections Framework* (JCF) library [4]. We considered three different groups of data structures, namely Sets, Lists, and Maps, and for each of these groups, we studied the energy consumption of each of its different implementations and methods. We exercised and monitored the energy consumed by each of the API methods when handling low (25000 objects), medium (250000 objects) and big (1 million objects) data sets [Pereira et al. 2016].

A first result of our study is a quantification of the energy spent by each method of each implementation, for each of the data structures we consider. Figure 1 presents the List results for population of 25k. Each row in the tables represents the measured methods, and for each analyzed implementation, we have two columns representing the consumption in Joules(J) and execution time in milliseconds(ms). Each row has a color highlight (under the J columns) varying between a Red to Yellow

[3]http://www.rosetta.org

[4]docs.oracle.com/javase/7/docs/technotes/guides/collection\s/index.html

to Green. The most energetically inefficient implementation for that row's method (the one with the highest consumed Joules) is highlighted Red. The implementation with the lowest consumed Joules (most energetically efficient) is highlighted Green.

	ArrayList		AttributeList		CopyOn Write ArrayList		LinkedList		RoleList		Role Unresolved List		Stack		Vector	
Methods	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms	J	ms
add	0.9773	71	1.1510	67	1.7839	117	1.8016	86	1.4801	76	1.1865	74	1.5659	76	1.5177	69
addAll	1.3353	76	1.0492	88	1.3586	82	1.1043	88	1.6661	76	1.8672	88	1.1015	88	1.7903	73
addAlli	1.7855	86	1.6035	68	1.1789	86	1.7272	99	1.5980	81	1.2497	85	1.2962	72	1.6268	90
addI	1.7125	93	1.3849	87	1.6558	119	1.6404	96	1.2718	85	1.3124	86	1.5287	83	1.4554	86
clear	1.1284	76	1.2409	75	1.7155	68	1.6497	74	1.6705	76	1.4304	80	1.6199	73	1.0574	71
contains	2.7568	166	2.4228	165	3.1768	167	3.1552	193	2.1751	162	2.4688	164	2.0128	166	2.1558	168
containsAll	1.5993	87	1.8053	92	2.1889	92	2.2887	118	1.3244	100	1.3930	96	1.2054	89	1.5091	87
get	2.0029	83	1.1171	78	1.4918	77	2.0168	109	2.2110	81	1.6613	71	1.8956	86	1.4978	73
indexOf	1.4447	76	2.0325	84	1.5682	70	2.6289	101	1.5674	79	1.1944	81	1.8090	81	2.0788	75
iterateAll	2.0701	79	1.0473	77	1.0103	73	2.6401	107	1.3605	85	1.7822	71	1.6036	81	1.1336	87
iterator	1.4893	84	1.1589	84	1.3922	72	1.7666	108	1.9760	73	1.3300	79	2.1895	84	1.6505	83
lastIndexOf	1.7750	99	1.7666	98	2.0383	94	2.5019	127	1.8914	92	1.4211	95	1.2260	84	1.2296	96
listIterator	1.4457	76	1.6190	84	1.3737	71	2.5003	106	1.3380	80	1.5176	85	1.6354	69	1.2746	81
listIteratori	1.7356	78	1.1552	81	1.5160	77	2.1996	105	1.7588	79	1.0334	80	1.8799	85	1.7545	78
remove	1.1308	96	1.4480	85	2.1946	162	1.6924	98	1.4560	84	1.1368	85	1.2663	96	1.4973	82
removeAll	8.0905	671	7.8108	697	7.3237	666	8.3150	752	7.6148	692	7.9911	664	7.3824	654	7.1281	665
removei	1.9135	85	1.3534	92	2.2858	118	1.7174	100	1.6308	85	1.6369	89	1.5850	81	1.5486	90
retainAll	2.7037	193	2.7845	200	2.6052	198	2.5982	205	3.0973	197	2.4172	200	2.7635	242	3.4019	245
set	0.9476	64	1.5943	70	1.9669	110	2.0474	112	1.5249	76	1.2312	73	1.4938	75	1.4957	72
sublist	1.3108	76	1.6021	80	1.4792	80	1.8457	98	1.4910	85	1.5117	71	1.7082	75	0.9414	75
toArray	1.6418	84	1.5024	84	2.0934	73	1.6739	106	1.5418	79	1.7455	83	1.5694	69	2.0213	80

Fig. 1: List results for population of 25k

Both *RoleUnresolvedList* and *AttributeList* contain the most efficient methods. Interesting to point out that both of these extend *ArrayList*, which contains less efficient methods, and very different consumption values in comparison with these two. We can also clearly see that *LinkedList* is by far the most inefficient List implementation. We can also see examples where a faster collection is less energy efficient. For example, looking at the AttributeList, the *addAll* method takes 88 milliseconds and consumes 1.0492 joules, while the *set* method is faster, taking 70 milliseconds, yet it consumes more energy, 1.5943 joules.

This JCF energy-awareness can not only be used to steer software developers in writing greener Java software, but also in optimizing legacy Java code. We have developed a Java data structure refactoring tool, named *jStanley*, which refactors Java source code when a greener collection is available [Pereira et al. 2018]. *jStanley* is a static analysis tool which suggests a more energy efficient (and/or performance efficient) Java collection, by statically detecting collections used in a Java project, and which methods are used for each collection. Using this information, it not only suggests a better alternative, but can automatically refactor the code with the new collections if the programmer chooses so. To see the full data on the remainder Java collections, please see [Pereira et al. 2016].

We have also executed an initial evaluation with 7 publicly available Java projects used in other research works. Figure III contains the information on the projects analyzed, including the number of tests cases and their coverage. The *Analysis* column displays the time spent by *jStanley* to detect and change the energy inefficient collections, and how many were changed. Finally, the *Improvement* column shows the percentage of improvement on the project's energy consumption (total and CPU energy) and performance. By using *jStanley*, we were able to improve the energy consumption between 2% and 17%. The execution time has also decreased between 2% and 13%

4. ENERGYWARE SOURCE CODE ANALYSIS

While programming language Integrated Development Environments (IDE) have traditionally incorporated powerful advanced type and modular systems; refactoring, testing and debugging frameworks;

Table III. : Evaluation data for the projects

	Test Suite		Analysis		Improvement		
Project	**#Tests**	**%Coverage**	**Analysis (ms)**	**#Changes**	**%PKG (J)**	**%CPU (J)**	**%ms**
Barbecue	152	62	2735	14	5.10	5.81	1.70
Battlecry	1*	69.4	514	4	16.79	11.49	12.76
Jodatime	4221	88.5	10490	5	7.21	7.29	7.75
Lagoon	18	4	1513	7	4.25	4.38	3.18
Templateit	3	14	1019	14	1.55	1.77	2.05
Twfbplayer	57	91	3437	51	6.07	6.05	3.14
Xisemele	167	20	588	1	6.04	6.30	4.36

* Instead of unit tests, this project has a simulated execution example

and other tools to improve software developers productivity and effectiveness, there is no concrete evidence that this trend has included techniques to optimize or even analyze source code energy consumption.

In GSL we developed a methodology to statically detect abnormal energy consumption in the source code of a software system [Pereira et al. 2017; Pereira 2017; 2018; Pereira et al. 2018]. We defined SPELL - *SPectrum-based Energy Leak Localization* to determine red (energy inefficient) areas in software. We consider an *energy leak* synonymous to an energy inefficiency. In this context, a parallel is made between the detection of anomalies in the energy consumption of software during program execution, and the detection of faults in the execution of a program. Having this parallelism established, we adapted established fault localization techniques, often used to detect software bugs in program executions, to locate energetic faults in programs.

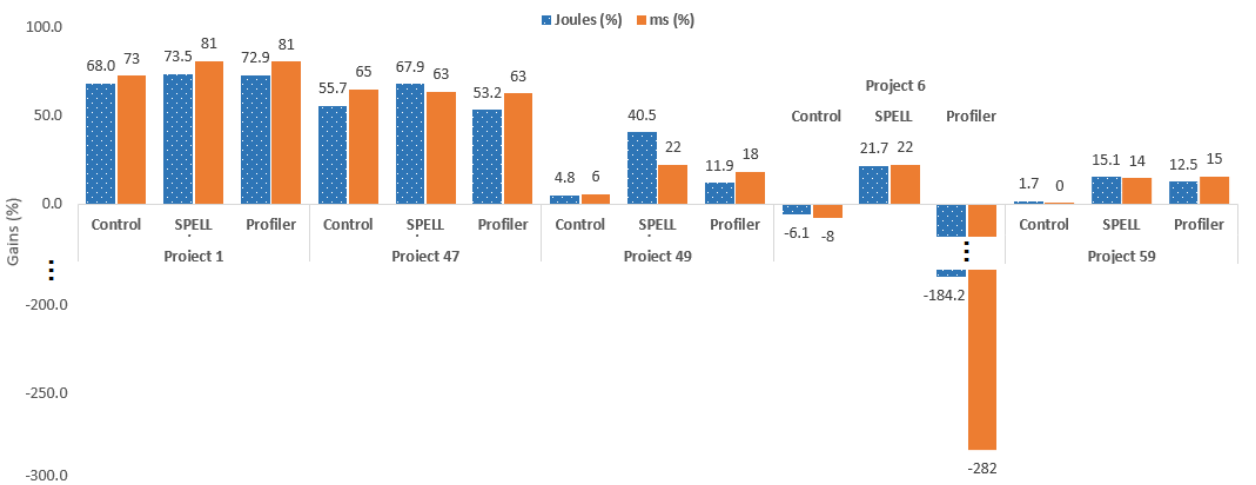

Fig. 2: Global percentage of gains for all projects

The software system to be analyzed is executed with a set of test cases, and components of such system (for example, packages, functions, loops, etc.) are instrumented to estimate/measure the energy consumption at runtime. Inefficient energy consumption, the so-called energy leaks, are interpreted in SPELL as program faults, and we adapt Spectrum-based Fault Localization (SFL) techniques to relate energy consumption to the systems source code. Our analysis associates different percentage of responsibility for the energy consumed to the different components of the underlying system. Thus, the result of our analysis is a ranking of components sorted by their likelihood of being responsible for

energy leaks, essentially pinpointing and prioritizing the developers attention on the most critical red spots in the analyzed system. Thus, giving more useful information to have better support in making decisions of what parts of the system need to be optimized, ultimately helping place a new stepping stone for energy-aware programming.

Our SPELL methodology is language independent. However, to validate it with real software systems, we built a specific front-end for the Java language. Supported by this tool, our technique was able to identify potential energy leaks in the source code of concrete Java projects. Based on this identification, a set of expert Java programmers were then asked to improve the (energy) efficiency of those projects. Figure 2 shows the results of our study. For each project, we had one programmer use SPELL, one use a profiler, and one use nothing as to have a control group. The blue bars represent the energy gains percentage, while the orange bars represent the performance gains percentage. Both compared to the original, unmodified, project.

In all 6 projects, programmers using SPELL were able to better optimize the energy consumption of the analyzed Java projects, even when compared to those using profilers. The analysis of their success in doing so provided statistical evidence that the programs they ended up altering indeed consume less energy that the ones they were originally given, with an improvement, for different projects, between 15% and 74%.

5. CONCLUSIONS

This document reported the work researched and performed at the Green Software Laboratory. Our studies show that applying energyware practices when choosing a programming language, data structures, and the applied code practices of implementing one's software do play a key role in the energy efficiency of the resulting software system. As our results show, energy optimization cannot always be solved by run-time optimization practices. The presented techniques, tools, and motivation of our GSL project is to steer programmers into developing more energy sustainable software. Thus, energyware analysis plays an important part in achieving these goals.

A more detailed look on each section's content, tools, repositories, and other related research work can be found in the Green Software Lab's Website [5].

REFERENCES

Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and Joo Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP)*. Article 7, 8 pages. (best paper award).

Rui Pereira. 2017. Locating Energy Hotspots in Source Code. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 88–90. DOI: http://dx.doi.org/10.1109/ICSE-C.2017.151 (ACM SRC silver award).

Rui Pereira. 2018. *Energyware Engineering: Techniques and Tools for Green Software Development*. Master's thesis. Departamento de Informática, Universidade do Minho.

Rui Pereira, Tiago Carçã, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2018. SPELLing Out Energy Leaks: Aiding Developers Locate Energy Inefficient Code. (2018). (submitted).

R. Pereira, T. Caro, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proc. of the 39th Int. Conf. on Soft. Eng. Companion*. ACM.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, Jo ao Paulo Fernandes, and João Saraiva. 2018. Ranking Programming Languages by Energy Efficiency. *Science of Computer Programming* (2018). Submitted.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM*

[5] Green Software Laboratory: https://greenlab.di.uminho.pt

SIGPLAN International Conference on Software Language Engineering (SLE 2017). ACM, New York, NY, USA, 256–267. DOI:http://dx.doi.org/10.1145/3136014.3136031

Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of the 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press.