Evaluation of the impact on energy consumption of lazy versus strict evaluation of Haskell data-structures

Gilberto Melfe
CISUC
Department of Informatics Engineering
Universidade de Coimbra
Coimbra, Portugal
gilbertomelfe@gmail.com

Alcides Fonseca
LASIGE
Department of Informatics
Faculdade de Ciências da Universidade de Lisboa
Lisboa, Portugal
amfonseca@ciencias.ulisboa.pt

João Paulo Fernandes
CISUC
Department of Informatics Engineering
Universidade de Coimbra
Coimbra, Portugal
jpf@dei.uc.pt

ABSTRACT

Data processing is one of the most energy-consuming tasks of computing systems. For both environmental and economical reasons, it is necessary to optimize software programs to be more energy efficient. There has been a lazy versus strict evaluation debate in the community, with a special focus on functional programming, with regards to program speed. Until now there has not been any insight about the impact of the evaluation strategy in energy consumption. We have used micro-benchmarks on lazy and strict implementations of the Map data structure abstraction to understand how execution time and package and RAM energy consumptions are affected by using lazy or strict implementations. This study proposes recommendations for developers to use strict evaluation for most tasks as the default approach, specially when iterating over elements in Maps.

CCS CONCEPTS

• **Software and its engineering** → *Software design tradeoffs*;

KEYWORDS

Functional Programming, Energy Consumption, Haskell, Data Structures

ACM Reference Format:
Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Evaluation of the impact on energy consumption of lazy versus strict evaluation of Haskell data-structures. In *XXII Brazilian Symposium on Programming Languages (SBLP 2018), September 20–21, 2018, SAO CARLOS, Brazil.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3264637.3264648

1 INTRODUCTION

Energy consumption has become a societal concern over the last decades. On a global scale, it is important to reduce energy consumption, preventing the depletion of non-renewable energy sources. It has been reported [5] that in 2012, computers in the information and communications technologies (ICT) area were responsible for almost 5% of the world's energy consumption and that that ratio was growing. These values do not account for personal computing, which in modern countries (e.g. France) can be responsible for more energy than servers [1]. Reducing energy consumption has a larger impact than just environmental: companies want to reduce their electricity spendings, and consumers want their battery-powered devices to last longer. This is specially the case with laptops, tablets and smartphones.

Energy consumption has been a concern in computing for a long time, but it has been mostly the focus of hardware designers, that have to balance the speed vs. the energy consumption of devices [14].

Recently, while there has been concrete evidence that software developers have limited knowledge on how to reduce the energy footprint of the software they develop [16], it has also been made clear that energy-efficiency is a concern for them [20], with software design decisions taking a toll on energy consumption. Examples of such decisions are the choice of the programming language [17] and of operations on data-structures [8, 10, 13, 18].

This work expands the line of work that focuses on operations on data-structures, which are present in any software. Furthermore, data processing is very representative in server workloads, occurring frequently within databases, and are essential for any machine-learning algorithm.

Existing work has shown that different data structures can have a different energy and time impact on the execution of the program [8, 10, 13, 18], resulting in a set of recommendations for developers to choose the implementations that are more energy-efficient. Two of these studies [10, 13] have focused on `Haskell`, a lazy functional programming language that is substantially different from other languages, like Java. In those studies it was not clear what was the impact of laziness on time- and energy-performance. The impact of laziness on execution-time performance has been studied in the past [4, 11] concluding that strictness is generally preferable, but not always better for obtaining faster programs.

In previous energy-aware analysis of `Haskell` data-structures [13], it has been identified that using an aggressive compiler optimization can slow down programs. One of the main optimizations performed by the Glasgow Haskell Compiler (GHC) is inlining functions, which can help the optimizer avoid lazy evaluation when it can provably, do so without altering the semantics of the program. This has further motivated this work to understand the impact of laziness.

SBLP 2018, September 20–21, 2018, SAO CARLOS, Brazil

ACM ISBN 978-1-4503-6480-5/18/09…$15.00
https://doi.org/10.1145/3264637.3264648

In this work, we address the question of how laziness and strictness impact the execution time and energy of data-structure operations, through an experimental evaluation based on existing micro-benchmarks. These micro-benchmarks are executed on a lazy and a strict version of each data-structure, allowing to compare results. More concretely, this work makes the following contributions:

- An empirical evaluation of strictness *vs.* laziness in data-structure operations over micro-benchmarks.
- A set of recommendations for developers of `Haskell` and other languages where laziness/strictness is optional, in order to minimize energy consumption.

The remaining of the article is organized as follows: Section 2 presents related work that provides background on both the motivation and techniques used in our evaluation. Section 3 presents the details of our approach to compare the two evaluation strategies. Section 4 presents and analyses the results obtained from executing our micro-benchmarks. Section 5 presents and addresses the major threats to the validity of this study. Finally, Section 6 draws conclusions and presents future work.

2 RELATED WORK

The interest in being able to compare the (runtime and memory) efficiency of equivalent data structures has received tremendous attention by (not only) the research community. This interest has recently been complemented with efforts to understand the energy efficiency of comparable data structures.

In [18] the authors studied the Java Collection Framework data structures, trying to determine, for each operation over them, the most energy-efficient implementation. Their study makes it possible to choose the most efficient implementation according to a piece of software's utilization of the available operations. In [7] the authors studied the Java List, Map and Set abstractions. They encountered very different energy consumption levels for specific operations in the various implementations. One of their conclusions is that computation is a greater energy consumer than memory usage. In our work we also compare different implementations for specific data structure abstractions focusing on a sufficiently different programming language, `Haskell`. Indeed `Haskell` is a programming language that has features such as laziness, partial function application, tail recursion, software transactional memory, and a kind system [19]. Features that are not standard in most mainstream programming languages. Also, the use of recursion is standard in `Haskell` programs, and the type system of the language restricts side effects, in a unique manner.

In [10] the authors have studied the energy behavior of programs written in `Haskell`, comparing different data structure implementations, provided by the `Edison` `Haskell` library and used in sequential programs/benchmarks, and the use of different concurrency primitives. They concluded that for sequential `Haskell` programs the execution time can be a strong predictor for energy consumption. When it comes to concurrent programs they concluded that: i) simple refactorings (e.g. switch between thread management constructs or data sharing primitives) may have a significant impact on energy consumption, and ii) executing faster does not mean it will always consume less energy. In [13] the authors have studied the relation between execution time and CPU package and DRAM memory energy consumption, for `Haskell` programs. They also investigated the effect that different GHC (Glasgow Haskell Compiler) optimization options packages have on the time/energy performance. They concluded that: i) time performance and package/dram energy performances are directly proportional, ii) the impact of memory energy consumption is lower than the package energy consumption and iii) compiler optimization options can decrease or increase execution time, but the energy consumption remains directly proportional to time.

While our work also considers data structures, `Haskell` programs and measures execution time, package and DRAM energy consumptions, we focus on comparing the data structure implementations, aiming to distinguish, from an energy consumption perspective, the non-strict and strict evaluation strategies.

To the best of our knowledge this is the first attempt at such a distinction.

Actually, while programmers and researchers have historically disputed the (efficiency) benefits and disadvantages of laziness vs. strictness, and despite our best attempts, we actually could not find published research work that focuses on empirically comparing both alternatives applied to comparable data structures, even if limited to comparing their runtime and memory efficiency.

3 APPROACH

In this section we describe the methodological approach that we have followed in order to compare the use of lazy approaches to programming against the use of strict equivalents, in the context of `Haskell`.

The programs that we compare are based on data structure libraries that make fundamental use of laziness or strictness. These libraries are described in detail in Section 3.1.

In order to build programs that make extensive use of such data structure libraries and their operations we rely on an independent benchmark, that we describe in detail in Section 3.2.

Finally, in Section 3.3 we describe the experimental setup that we have built in order to measure the runtime and the energy consumption of the programs whose efficiency we seek to compare.

3.1 Data structure libraries

In this paper, we study three different instances of the Map abstraction, which essentially holds $key \mapsto value$ associations. Those instances are:

`Data.Map` a general-purpose implementation of ordered maps (dictionaries), based on balanced binary trees

`Data.IntMap` an efficient implementation of maps, with integer keys, based on big-endian patricia trees

`Data.HashMap` an implementation of unordered maps from hashable keys to values, optimized for performance, based on hash array mapped tries

These different instances have similar APIs. They include functions to: i) construct basic data structures, e.g. `singleton` (taking a key and a value and returning a map with that association), ii) insert data into existing data structures, e.g. `insert` (taking a key, a value and an existing map, and returning a new map with all the existing, plus the new, key/value association), iii) query the data structures, e.g. `lookup` (taking a key and a map, and returning a possible value,

if the key/value association exists in the map), and iv) deleting key/value associations from existing maps, e.g. `delete` (taking a key and map, and returning a new map with that key/value association removed). There are also functions to: v) traverse maps, e.g. the `map` function (taking a function and a map, and applying that function to every value present in the map), vi) combine maps, e.g. `union` (taking two maps and returning the union of those maps, preferring data from the first when duplicate key are encountered) and `intersection` (taking two maps and returning data from the first for the keys existing in both), vii) filter maps, e.g. `filter` (taking a predicate, function returning bool, and returning a map with all key/value associations for which the value satisfies the predicate), and viii) convert maps, e.g. `toList` (taking a map and returning a list with all the key/value pairs present in the map).

The most significant difference between the three instances APIs is that: for the `Data.Map` instance, functions are available that are based on an underlying order of the key/value associations, e.g. `elemAt` which retrieves a key/value pair by its zero-based index in the sequence sorted by keys.

For each API instance, two equivalent implementations were evaluated: one lazy and another that is strict. Despite performing the same operations, they have different strictness semantics. As expected, the strict variant forces the evaluation of both keys and values down to Weak Head Normal Form (WHNF, meaning that values are evaluated until the outermost data constructor). The lazy version has the same guarantee only for keys, not evaluating values until they are necessary. The `Data.Map` and `Data.IntMap`, lazy and strict implementations, can be found in the *containers* `Haskell` package[1]. The `Data.HashMap` implementations, are found in the *unordered-containers* `Haskell` package[2]. Both packages are installed by default with the standard Haskell Platform[3] installation.

3.2 The benchmark

In order to build a coherent and comparable effort that makes extensive use of the functions available on the different Map libraries, we relied on an independent micro-benchmark suite.

This benchmark was proposed in [9], has been used in the past in different contexts and by different authors [10, 12, 13], and is composed of the operations listed in Table 1. For each operation, various suggested input sizes and number of repetitions are listed. Each line of the table abstracts one micro-benchmark with the following format:

$$iters * operation(base, elems)$$

This format should be read as: repeat *iters* times, an operation working with a base data structure of size *base*, and with (if required by the operation) an additional data structure of size *elems*.

As an example, the `removeAll` operation should remove 1000 elements from a base of 100000 elements, and this is repeated 10 times.

The operations in the benchmark were then implemented by us resorting to the API functions presented by the respective `Haskell` modules. The goal was to have a uniform definition for each operation, regardless of the concrete data structure to be used, so that the impact of each data structure choice can be evidenced. A few example implementations follow. The `containsAll` operation was implemented thus:

Table 1: Benchmark Operations.

iters	*operation*	*base*	*elems*
1	add	100000	100000
1000	addAll	100000	1000
1	clear	100000	n.a.
1000	contains	100000	1
5000	containsAll	100000	1000
1	iterator	100000	n.a.
10000	remove	100000	1
10	removeAll	100000	1000
10	retainAll	100000	1000
5000	toArray	100000	n.a.

```
containsAll :: Map Key Datum -> Map Key Datum -> Bool
containsAll a b =
    let
        bKeys = keys b
    in
        and . map (\ bk ->
                    ( member bk a ) &&
                    ( ( a ! bk ) == ( b ! bk ) ) ) $ bKeys
```

The type of that function tells us it expects two `Map` values and returns a value of type `Bool`. The only difference in the same operation realization for the different map implementations is the function type. If we replace `Map` in the presented function by `HashMap` we get the function used with that implementation. If we replace `Map Key` in the presented function by `IntMap` we get the specialized type/function with `Int` type keys (used with the `Data.IntMap` implementation).

The remaining code, the body of the function, works as follows: we obtain a list of the keys in map `b` with the `keys` function; we then traverse that list with the `map` function, mapping the keys to `Bool` values, indicating their presence in map `a` (with the `member` function) and the equivalence of the values associated with those keys in each map (accessed with the `!` function); finally we take the conjuntion of all `Bool` values with the `and` function. This tells us if the totality of map `b` is contained in map `a`.

The `iterator` operation has the following implementation:

```
iterator :: Map Key Datum -> Map Key Datum
iterator = map id
```

This operation receives one `Map`, to be traversed, and returns that same map as a result (the `map` function traverses the map, applying the `id` function to every value; the `id` function simply returns back its single argument).

The `removeAll` operation has the following implementation:

```
removeAll :: Map Key Datum -> Map Key Datum -> Map Key Datum
removeAll = difference
```

This operation receives two `Map` values. It returns a new `Map` containing the elements of the first argument map which are not

[1] Available at https://hackage.haskell.org/package/containers
[2] Available at https://hackage.haskell.org/package/unordered-containers
[3] The Haskell Platform is a set of compiler and Haskell libraries available at https://www.haskell.org/platform/

present in the second argument map, effectively removing from the first all elements present in the second.

Throughout the executions of the benchmark the Key and Datum type synonyms were fixed to be the `Int` type.

3.3 Experimental Setup

In our work, we have used Running Average Power Limit (RAPL) [2] in order to obtain energy consumption data for the program variants we want to compare.

RAPL is an interface provided by modern Intel processors which allows for power consumption adjustment for processor packages, and, of more interest to this study, for the estimation of energy consumption related metrics. The estimations are accessed through Model Specific Registers (MSRs) present in the processors, via a specific Linux kernel module.

Although RAPL provides consumption estimations, their accuracy has been extensively confirmed [3, 6, 21]. For desktop computers, which is the context studied in this paper, such studies show that the values estimated by RAPL are consistent, even if they suffer from an offset. This means that if we look at two concrete estimation values provided by RAPL, they can be safely compared since although they may not correspond to the real consumption values, they are (equally) impacted by the same offset. So, if the first is greater than the second, offsets included, than it would also be greater than the second, offsets excluded.

RAPL can report energy consumption values for four (physical) domains, in and around of a processor socket. These are:

- *PKG*: total energy consumed by an entire processor socket
- *PP0*: energy consumed by all cores and caches
- *PP1*: energy consumed by uncore devices (such as integrated GPUs), usually unavailable
- *DRAM*: energy consumed by all DIMMs

Previous work [10, 13] has considered the energy consumption of the *PKG* and *DRAM* domains. In this work we consider those same two domains.

In order to automate the execution of the program variants we implemented and the analysis of their efficiency indicators (runtime and energy consumption), we have used `Criterion` [15].

`Criterion` is a powerful `Haskell` library and tool that provides a way to define, execute and report the results of a generic set of benchmarks. By default, each benchmark runs for at least 5 seconds, but it can run for a longer period if the library needs more data to perform the analysis of the results. After gathering enough data it performs an Ordinary Least-Squares Regression on the values obtained, to estimate the time needed for a single execution of the activity being benchmarked.

In previous work [10], `Criterion` has been extended so that it would report on not only execution time but also on the *PKG* and *DRAM* [13] energy consumptions. Our experiments also considered this modified version of `Criterion`.

In this work we have not experimented with different compiler optimization options. Since we have used the `Haskell` Cabal build tool, the benchmarks were compiled with Cabal's default (GHC) optimization option, O1.

The experiments were executed on a 4-core Intel Core i7-4790 (Haswell) with 16 GB of DDR 1600 running openSUSE 42.1 and GHC 7.10.2.

4 ANALYSIS AND RESULTS

In this section we present and analyze the results obtained from evaluating the benchmark operations over different data structures both in their lazy and strict versions.

The micro-benchmark suite consists of 10 micro-benchmarks (or operations, detailed in Table 1) that were executed for each lazy and strict variant of each of the 3 instances of the map abstraction (`Map`, `IntMap` and `HashMap`), resulting in a total of 60 executions. From each execution, we measured the three metrics of interest: execution time, *PKG* energy consumption and *DRAM* energy consumption. This setting results in 90 head-to-head comparisons between lazy values and strict values, i.e, in a total of 180 values.

Results shown in vertical bar plots have the 95% confidence interval depicted as error bars, to evidence the statistical significance of the comparisons. Error bars that overlap between strict and lazy are not considered as significant.

In the majority (58 out of 90) of strict-vs-lazy comparisons, the lazy variant has a greater execution time, *PKG* and *DRAM* energy consumption. Figure 1 shows an example of this behavior in the `iterator` operation for all three data-structures. The `iterator` operation of `Data.Map` and `Data.IntMap` is the most extreme case of advantage of strict over lazy implementations. The lazy versions take between 27.09% and 36.78% more time/energy for the `Data.IntMap`, and between 44.47% and 46.96% more time/energy for the `Data.Map` implementation.

In the remaining 32 cases, the lazy version outperforms (both in speed and energy consumption) the strict version. Figure 2 shows the most evident example, concerning the `containsAll` operation over the `IntMap` implementation. In execution time, the lazy version of the `add` operation, for the `Data.Map` implementation, was 10.93% faster than the strict version.

To gain a better understanding of the overall behavior of the different operations and implementations, Figure 3 shows the ratio of improvement of the strict variants over the lazy variants for each of the `Data.Map`, `Data.IntMap` and `Data.HashMap` implementations. A positive value indicates that the strict variant improved over the lazy variant (the strict variant took less time, or energy, than the lazy variant), and a negative value indicates the opposite relation.

Because we were using micro-benchmarks the results obtained are somewhat sensitive to, even small, measurement imprecisions (we discuss this threat to validity in detail in Section 5). We posit that differences between the lazy and strict variants, lower than a few percent, essentially mean they are equivalent. The landscape of results was basically the following: there were 55.56%, of the lazy and strict value comparisons, that fell within a margin lower than 5% variation, 28.89% that fell within the 5% to less than 10% margin and 15.55% for which the difference between lazy and strict amounted to a value greater than 10%.

We also computed, for each combination of measurements (e.g. time vs. *PKG* energy or *DRAM* energy, time vs. total energy and *PKG* vs. *DRAM* energy), the Pearson correlation coefficient, having

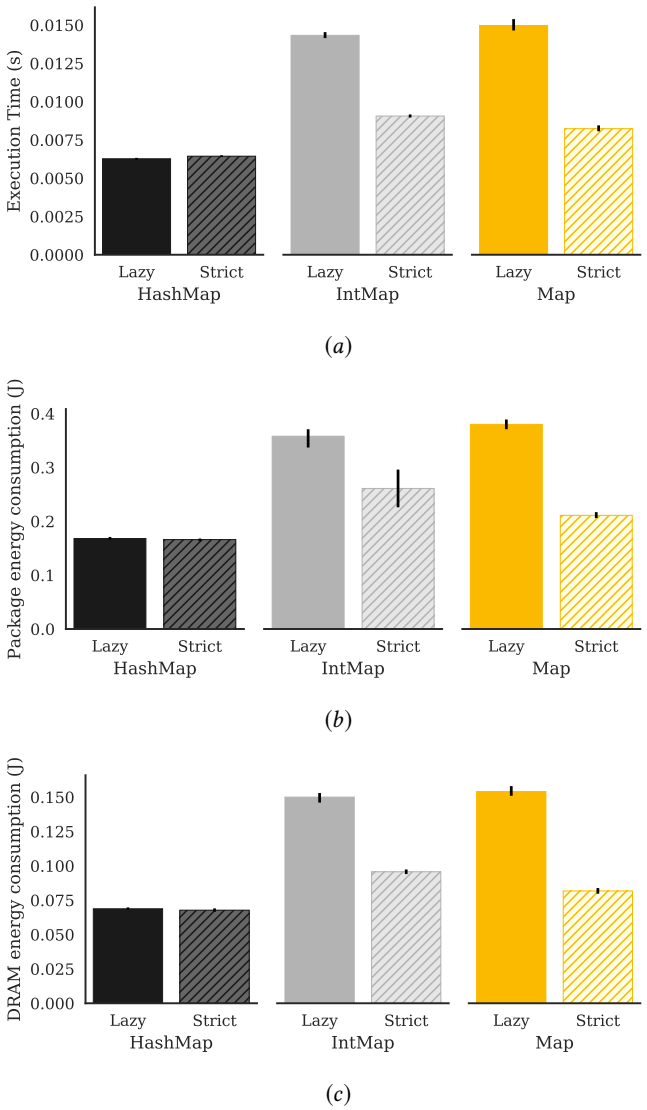

Figure 1: (a) Time, (b) Package energy, (c) DRAM energy measurements, for the `iterator` operation.

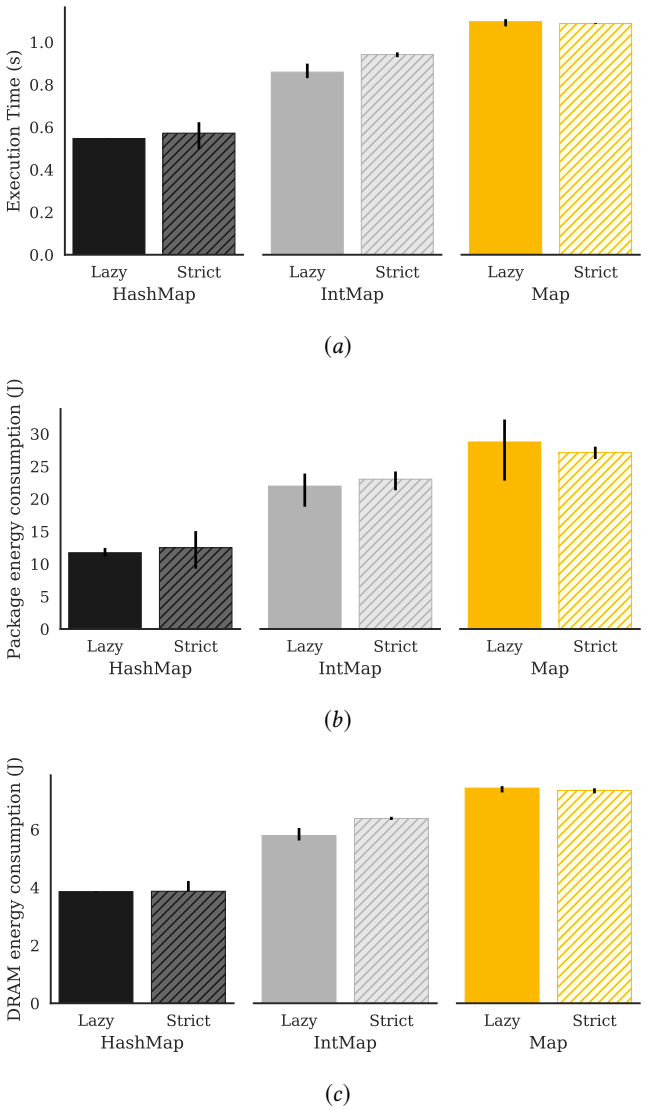

Figure 2: (a) Time, (b) Package energy, (c) DRAM energy measurements, for the `containsAll` operation.

obtained a value above 0.99 for each such combination, which indicates a strong positive correlation between the measurements.

For the conditions of our experiments, this means that PKG energy consumption, DRAM energy consumption and total energy consumption are all directly proportional to execution time: when the execution time grows so does the energy consumption, when the execution time decreases so does the energy consumption.

All the code that we have written for designing, implementing and running our experiment, as well as the obtained results, both in raw and processed formats, are available in the companion site:

http://green-haskell.github.io/energy_consumption-lazy_vs_strict_evaluation_of_data-structures/

5 THREATS TO VALIDITY

In this section, we analyze aspects that may threaten the validity of our study and the conclusions that we draw.

The focus of our work is the energy behavior of the `Haskell` programming language and (some of) its libraries. `Haskell` is purely functional and lazy by default. The results obtained are thus specific to a language that has very different characteristics from mainstream languages, and therefore may not be "transferable" to those other languages.

Regarding the execution environment, we have run the proposed benchmarks in only one hardware configuration. While different processors, DRAM configurations and architectures may influence

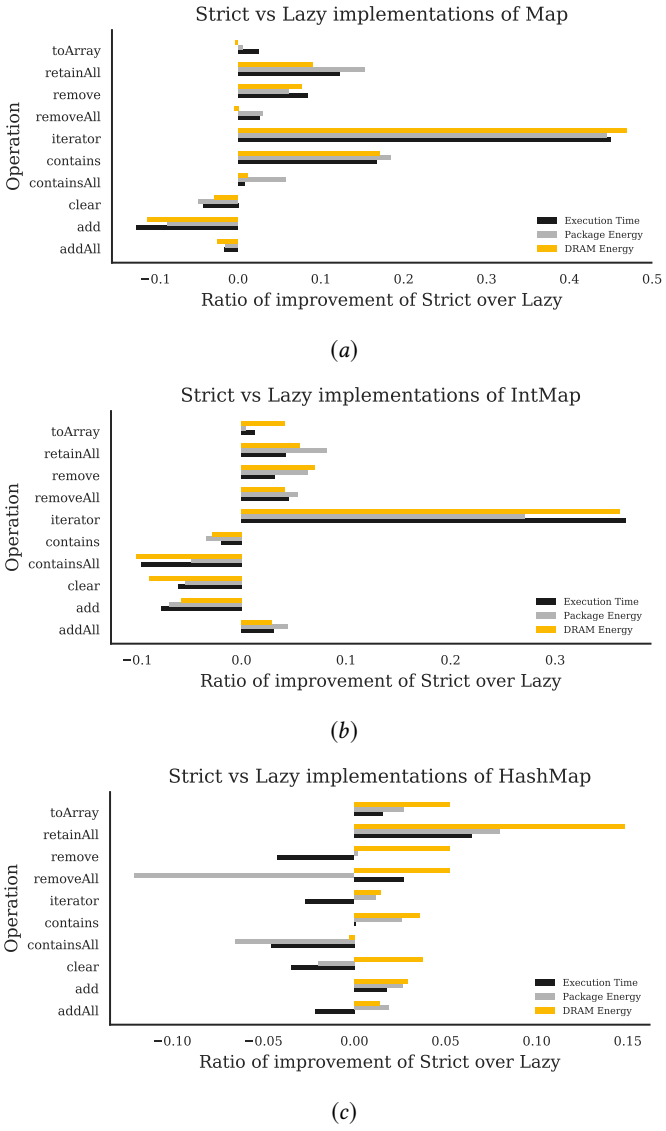

Figure 3: Time, Package and DRAM energy improvements of strict over lazy evaluation of different operations on (a) Map, (b) IntMap, and (c) HashMap

the obtained results, we have chosen hardware that is widespread and so is commonly used.

We studied a small set of data structures implementations for which lazy and strict implementations were readily available. Although other data structure implementations exist for which we could study their energy efficiency, here we are focused in comparing lazy and strict equivalent implementations, which are not frequently available. In our work we chose to use the `Int` type as the type for the keys and values inserted into the map implementations. The `Int` type was chosen in order to have the minimum overhead in key/datum evaluation as possible, in order to focus on the data structure evaluation. The results obtained could vary if we use more complex types, for which the laziness/strictness trade-off is more distinguishable.

We used Intel's RAPL interface to gather energy measurements. Although different methods (e.g., physical measurement) might have been used, RAPL's accuracy has already been asserted in different studies [3, 6].

In this work we use micro-benchmarks to compare the execution time and energy consumption of equivalent data structure versions. By using micro-benchmarks, the results of our experiments, and the conclusions drawn by them, are subject to vary significantly among different experiment runs.

To assess these possible variations we re-executed our experience entirely and analyzed the obtained results. In our second execution, 83.89% of the measurements fell within a 5% margin of the first execution values (10.56% within a 5% to 10% margin, and 5.55% above a 10%). These results strengthen our confidence in the values obtained and the conclusions drawn. Nevertheless, the analysis of our results should be made with this (low) variance in mind.

6 CONCLUSIONS AND FUTURE WORK

In this work we explored the relation between execution time, package energy and DRAM energy, for lazy and strict evaluation strategies, used in different implementations of the Map data structure abstraction, for the `Haskell` language.

Based on the reported results, it is possible to conclude that:

i) the energy consumption is (linearly) proportional to the execution time (as confirmed by the calculated value(s), 0.99, for the Pearson correlation coefficient);
ii) the strict evaluation tends to be more efficient in terms of execution time and energy consumption than the lazy evaluation.

Developers should use strict implementations of data structures as a default, in order to improve their application performance in both time and energy consumption. However, if an application that relies on performing several *add*s on *Map*s or *IntMap*s, a Lazy implementation would be more performant.

In fact, the next steps of this line of research are to create more coarse-grained benchmarks that use these operations in a more real-world context. It is expected that the faster and more energy efficient solution is the one that performs better in the majority of the operations used in that larger benchmark. Another aspect we intend to explore is laziness vs strictness in more complex algorithms that rely on data structures, such as sorting or graph search algorithms.

In order to be able to assess whether the insights we have drawn here on the comparison between lazyness and strictness generalize to contexts that differ from the one we studied, we also plan to target more programming languages where it is possible to have lazy and strict variants of the same program.

ACKNOWLEDGMENTS

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718.

This work has also been funded by the LASIGE Research Unit (UID/CEC/00408/2013).

REFERENCES

[1] Fabrice Collard, Patrick Fève, and Franck Portier. 2005. Electricity consumption and ICT in the French service sector. *Energy Economics* 27, 3 (2005), 541–550.

[2] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '10)*. ACM, 189–194.

[3] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, 455–470.

[4] Robert Ennals and Simon Peyton Jones. 2003. Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-strict Programs. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, 287–298.

[5] Erol Gelenbe and Yves Caseau. 2015. The Impact of Information Technology on Energy Consumption and Carbon Emissions. *Ubiquity* 2015, June, Article 1 (June 2015), 15 pages.

[6] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (Jan. 2012), 13–17.

[7] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 225–236.

[8] Nicholas Hunt, Paramjit S. Sandhu, and Luis Ceze. 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*. 63–70.

[9] Leo Lewis. 2011. Java Collection Performance. http://dzone.com/articles/java-collection-performance

[10] Luís G. Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 517–528.

[11] Jan-Willem Maessen. 2002. Eager Haskell: Resource-bounded Execution Yields Efficient Iteration. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, 38–50.

[12] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 503–514.

[13] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Helping Developers Write Energy Efficient Haskell Through a Data-structure Evaluation. In *Proceedings of the 6th International Workshop on Green and Sustainable Software (GREENS '18)*. ACM, 9–15.

[14] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. 2014. A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems. *ACM Comput. Surv.* 46, 4, Article 47 (March 2014), 31 pages.

[15] Bryan O'Sullivan. 2009. criterion: Robust, reliable performance measurement and analysis. http://www.serpentine.com/criterion/

[16] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (May 2016), 83–89.

[17] 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 SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. ACM, 256–267.

[18] 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 *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

[19] Benjamin Pierce. 2002. Type Operators and Kinding. In *Types and Programming Languages*. MIT Press, Chapter 29.

[20] Gustavo Pinto and Fernando Castor. 2017. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* 60, 12 (Nov. 2017), 68–75.

[21] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (March 2012), 20–27.