Helping developers write energy efficient Haskell through a data-structure evaluation

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

How a program is written has implications in the energy consumption of the running system, with economical and environmental consequences.

In this context, understanding the energy consumption of operations on data-structures is crucial when optimizing software to execute under power constricted environments. Existing studies have not focused on the different components of energy consumption that processors expose, rather considering the global consumption.

To understand the relationship between CPU and memory energy consumptions with execution time, we instrument a microbenchmark suite to collect such values, and we analyze the results.

Our benchmark suite is comprised of 16 implementations of functional sequences, collections and associative collections while measuring detailed energy and time metrics. We further investigate the energy consumption impact of using different compilation optimizations.

We have concluded that energy consumption is directly proportional to execution time. Additionally, DRAM and Package energy consumptions are directly proportional, with the DRAM representing between 15 and 31% of the total energy consumption. Finally, we also conclude that optimizations can have both a positive or a negative impact on energy consumption.

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. Helping developers write energy efficient Haskell through a data-structure evaluation.

GREENS'18, May 27–26, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5732-6/18/05…$15.00
https://doi.org/10.1145/3194078.3194080

In *GREENS'18: IEEE/ACM 6th International Workshop on Green And Sustainable Software , May 27–26, 2018, Gothenburg, Sweden*, Randy Bilof (Ed.). ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194078.3194080

1 INTRODUCTION

Energy consumption of electronic devices has become a critical factor over the last decade. There is now a higher concern for sustainability and reducing our energetic footprint. More specifically, large datacenters and supercomputers are choosing the most energy-efficient devices instead of the faster and more powerful. The reasoning is economical, as higher power consumptions result in higher electricity bills over the years. On the other end of the spectrum, embedded and mobile devices (eg. phones, tablets, laptops) also need to be energy-efficient as they run on batteries most of the time.

While the focus of energy efficiency in the past was on hardware, nowadays the impact of software on power consumption has gained awareness [21]. It has been expected for software optimizations alone to provide savings from 30% to 90% [22]. As such, there has been a push for energy-aware software development processes and lifecycle.

In order to efficiently produce "green software", developers need to be informed of the energetic consequences of choices early in the development process.

This study focuses on providing developers with energetic information about data structure implementations during the development process, so the end result is an energy efficient program.

Our focus on data-structures is based on the "Algorithms + Data Structures = Programs" premise, in which there are often trade-offs between time and memory complexity when choosing data structures. Our goal is to study that same tradeoff when also considering energy consumption.

Given the heterogeneous performances of different programming languages [20], we focus on `Haskell`, a lazy functional programming language. Haskell is in the Redmonk top 20 programming language ranking [17], and it is used by several large software companies, of which Facebook, Google and Microsoft are some examples [13].

In previous work [16] we have addressed the relationship between execution time and Package (processor socket) energy consumption. In this work we focus on both the Package and DRAM energy consumption, a shortcoming of our previous work. This expanded study is relevant because of the previously mentioned trade-off between execution time and memory usage.

This work presents an empirical evaluation of different Haskell data structure implementations on a set of serial micro-benchmarks to assess the relationship among a) Execution time, b) Package, c) DRAM, and d) Total energy consumption. Furthermore, we also evaluate the impact of different optimization levels on time and energy consumption.

This work is organized as follows:

- Section 2 presents the related work concerning data structures, time, memory and energy consumption analysis.
- Section 3 revisits preliminary existing work, in order to lay the groundwork to further expand the understanding of energy implications of data structure implementations.
- Section 4 describes the novel evaluation performed to understand how energy consumption varies with different implementations of data structures and compiler optimizations.
- Section 5 discusses possible shortcomings of this work.
- Section 6 draws conclusions.

2 RELATED WORK

In [19] the authors studied different implementations for Java Collection Framework data structures to identify the most energy efficient implementations for each operation. Based on the results of that study, and taking into account the methods used within the source code of a specific program, it is possible to select the most energy efficient implementation for use in that program. In [10] the authors created energy profiles for operations on the Java List, Map and Set abstractions. They report significant differences in the energy consumption of implementations of those abstractions for specific operations. They also conclude that computation is more energy demanding than memory usage. Our work shares a similar methodology with both of the previous works, but focuses on a sufficiently different programming language (Haskell), targeting the Edison library data structure implementations.

In [7] the authors presented a framework to abstract different sources of energy consumption data, through a set of APIs. That framework allows developers to conduct profiling of their software taking energy consumption into account, in a portable manner. In our work, we also intend to make energy consumption data available to software developers, but we have focused only on using the Intel RAPL interface.

In [12] the authors evaluated the relationship between performance and energy consumption of various lock-free data structures, showing that, in the considered workloads, these often perform better and consume less energy than common locking implementations. In [8], different parallel algorithms are evaluated, with the results showing a high correlation between execution time and energy consumption. In this work, we also study the performance-energy consumption relationship of different data structures, but we do not consider concurrency or parallelism.

In [3] the authors proposed a theoretical framework for determining the bounds of the energy-consumption of algorithms, similar to how time and space complexities are analyzed. This work focuses on practical experiments, reporting actual execution time and energy consumption metrics, presenting evidence on the connection between the two.

In [14] the authors show that low-level code optimizations for low-power are different than code optimizations for high speed. In [23] the authors demonstrated that the optimizations used to reduce execution time increase the memory-related energy consumption. In [1] the authors study the effect of compilation optimizations on the energy consumption of embedded systems. They conclude that architecture-driven optimizations have a higher impact on the power/energy consumption of embedded systems than more general low-level code optimizations. This work also addresses whether Haskell compiler optimizations have energy-consumption penalties, when considering both CPU and DRAM energy consumptions.

In [18] the authors conducted a study to analyze the energy consumption impact of a number of optimizations available in GCC (GNU Compiler Collection). They found that in many platforms there is a frequent direct correlation between execution time and energy consumption. In our work we also confirm this direct correlation, for our experimental setup (using the GHC, Glasgow Haskell Compiler).

In [11] the authors compared the resource consumption of programs with and without compiler optimizations. They focused on the several languages backed by the GCC (GNU Compiler Collection). Their main findings are that the optimizations which produce faster running code also result in less energy consumption by the code produced. In our work we have also found this conclusion to be true. We have, however, discovered that performing optimizations can also lead to the generation of less performant code, in respect to, both time and energy consumption. This is confirmed in Section 4.3.

3 "HASKELL IN GREEN LAND", OUR FIRST APPROACH…

In previous work [16] we investigated the relationship between the execution time of an Haskell program and the energy consumed by the hardware while running the program. The focus of that work was two-fold: the usage of readily available data structure implementations (from the Edison Haskell library) and concurrent programming. This work substantially expands the first aspect.

Our research methodology is based on using micro-benchmarks to evaluate operations individually. We instrument benchmark execution to read and record processor registers related to energy consumption. We then analyze and compare the results of different implementations. These steps are detailed further in this section.

3.1 Experimental Setup

The Edison library. Edison is a Haskell library of purely functional data structures, providing three higher level abstractions (Sequences, Collections and Associative Collections), along with a number of concrete implementations for those abstractions. These are listed in Table 1, and a more comprehensive description can be found in the EdisonAPI [5] and EdisonCore [6] Haskell packages documentation.

We omited some of the implementations in Edison (EnumSet, MinHeap, TernaryTrie, SizedSeq, MyersStack, PatriciaLoMap and RevSeq) for different reasons: some were wrappers around other implementations; others deviated from the "common" API prescribed by the EdisonAPI package; one was limited in the number of elements it could contain; and another was simply too costly to evaluate such

Table 1: Abstractions and Implementations available in `Edison`.

Collections	Associative Collections	Sequences
		BankersQueue
EnumSet		SimpleQueue
StandardSet		BinaryRandList
UnbalancedSet	AssocList	JoinList
LazyPairingHeap	PatriciaLoMap	RandList
LeftistHeap	StandardMap	BraunSeq
MinHeap	TernaryTrie	FingerSeq
SkewHeap		ListSeq
SplayHeap		RevSeq
		SizedSeq
		MyersStack

that it would not perform the work required in a timely manner. We have also split the Collections analysis into separate analysis of `Heaps` and `Sets` because they do not strictly abide by the same API.

RAPL. Running Average Power Limit (RAPL) [2] is an interface provided by modern Intel processors which, among other capabilities, allows for the estimation of the system's energy consumption — accessed through a number of Model Specific Registers (MSR) — in four different domains:

- *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 [16] considered only the *PKG* domain. This work aims to overcome such shortcoming by analyzing the DRAM impact within the total package consumption.

The Criterion tool. `Criterion` is a library and tool used to define and execute benchmarks. Furthermore, this tool also facilitates the analysis of the collected data. We have extended `Criterion` to collected energy consumption metrics [16]. Energy-related metrics are obtained similarly to how execution time is measured: using external C function calls. In this case, C functions read values from the RAPL MSRs. Since there is a natural overflow in these registers, the values are post-processed. This was carried out by comparing consecutive values obtained from those MSRs, and discarding the energy consumed in the short window in which an overflow occurs. The high sampling frequency minimizes the imprecision of overflow verification.

The benchmark. In order to understand the impact of the different data-structures, we have used a micro-benchmark suite, initially developed for the Java programming language [15]. The operations used in this dataset are depicted in Table 2.

These operations can be abstracted by the format:

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

This format should be interpreted as: iterate *operation* a given number of times (*iters*) over a data structure with a *base* number of elements. If *operation* requires an additional data structure, the number of elements in it is given by *elems*. As an example, the `containsAll` operation checks if all elements of a secondary data structure with 1000 elements (*elems*) are present in a 100000 elements (*base*) data structure, and this operation is repeated 5000 times (*iters*).

Table 2: 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.

The implementation of these benchmarks on `Edison` tried to minimize the adaptation overhead, using as much of the `Edison` functions as possible. As an example, the `containsAll` operation for the `Associative Collections` abstraction was defined to use the already available `submap` `Edison` API method, thus:

```
containsAll :: A.FM Key Datum -> A.FM Key Datum -> Bool
containsAll = flip A.submap
```

Besides being a functional programming language, Haskell has another major difference when compared with Java: it is lazy by default. Laziness affects benchmarks: if the result of a computation will not be used later in the program, that computation will never occur. In order to guarantee that all iterations of the operation execute, regardless of whether its result is used or not, strictness is forced using `seq` primitives. In this case, we applied the `deepseq` primitive, which forces the strict evaluation of the first argument, and returns the second. As an example, we present the `addAllNTimes` function (used with the `Sequences` abstraction), which performs the `addAll` operation strictly `n` times.

```
addAllNTimes :: S.Seq Int -> S.Seq Int -> Int -> S.Seq Int
addAllNTimes s _ 0 = s
addAllNTimes s t n = deepseq (addAll s t)
                             (addAllNTimes s t (n - 1))
```

To ensure the practicality of the benchmark, a 3 hour limit was imposed for each operation. Whenever a benchmark would time out, the workload was reduced, either by reducing the number of iterations or of elements [16]. The detailed description of the operations is available in Table 3, with the modified workload parameters underlined.

3.2 Results

Preliminary experiments concluded that, for the same operation and implementation, execution time and energy consumption are directly proportional. Figure 1 shows the execution time and energy

Table 3: Modified Benchmark Operations.

abstraction	*iters*	*operation*	*base*	*elems*
Associative Collections	1	clear	50000	n.a.
	2500	remove	3125	1
	10	retainAll	25000	1000
	2500	toArray	3125	n.a.
Sequences	1	add	3125	50000
	625	containsAll	3125	1000

consumption relative to their respective maximums for the same operation on all the implementations of `Sequences`. `BraunSeq` was the slowest and most energy inefficient while `ListSeq` was the most performant implementation, only taking around 5% of `BraunSeq`'s time and energy consumption. The direct correlation between time and energy consumption is shown in Figure 1, with both bars for each implementation being the same height.

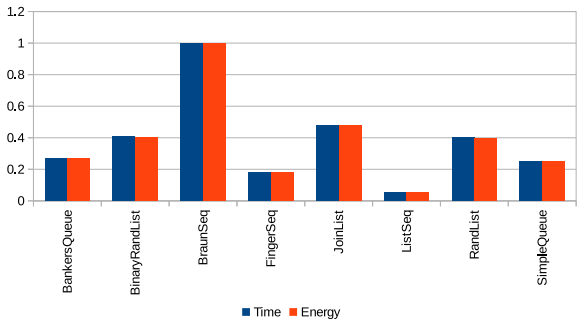

Figure 1: Relative Time and Energy consumption values for the `toArray` operation, for `Sequences`.

4 ENERGY CONSUMPTION ANALYSIS OF DATA-STRUCTURES

Our previous work [16] was limited to the analysis of the *PKG* RAPL domain. However, RAPL exposes other fine-grained information about energy consumption, such as the *DRAM* domain. This work expands on previous work to consider such domain, since different data-structure implementations imply trade-offs between time and memory complexity.

More concretely, this work focuses on the following three new research questions:

RQ1 How does execution time influence package and DRAM energy consumptions?

RQ2 What is the impact of DRAM energy consumption on the overall energy consumption?

RQ3 How do the different compilation optimization options, available in a modern `Haskell` optimizing compiler, affect the execution time and package/DRAM energy consumptions?

These questions are addressed in this section: in Section 4.1, we detail the experimental setup; in Section 4.2 we describe experiments that evaluate the relationship between package and DRAM energy consumptions and execution time, answering the first two research questions; in Section 4.3, we address the last research question.

4.1 Experimental Setup

In order to answer the proposed research questions, we have relied on the same methodology used for our previous work [16] (summarized in Section 3).

We studied `Edison`'s data structure implementations detailed in Table 1, excluding a few of them as already explained in Section 3.1. We used the same benchmark suite (described in Section 3.1) with the same operations (Table 2) and problem sizes (Table 3).

Because we now want to focus on both the *PKG* and *DRAM* RAPL domains, we adapted `Criterion` to collect DRAM energy consumption measurements in parallel to time and package energy measurements.

Experiments were executed on a 4-core Intel Core i7-4790 (Haswell) with 16 GB of DDR 1600 running openSUSE 13.2 and GHC 7.10.2.

4.2 Execution time *versus PKG* and *DRAM* energy consumptions

The modifications of `Criterion` to include *DRAM* measurements allow us to revisit the experiments of our previous work under a new perspective. The following experiment focuses on the relationship between the execution time and the package and the DRAM energy consumption domains. Another goal is to understand the impact that each of these two domains have on overall energy consumption.

We ran the adapted benchmark (Section 3.1) to collect the measurements for the three relevant metrics for each operation and implementation: execution time, package, and DRAM, energy consumption measurements. The absolute values are dependent on the hardware and the benchmark configurations, which is not the focus of this experiment. As such, we present the relative values as percentages of the maximum value observed for a concrete operation across all implementations. Using relative values evidences the relationship between different metrics.

In this section we present the more meaningful and representative results. The complete results are available in the companion site http://green-haskell.github.io.

Figure 2 presents the three metrics for the `iterator` operation, for the `Sequences` abstraction. The highest execution time, package or DRAM energy consumptions are represented as 1, and other values are represented as ratios of those values. For this particular operation, the `FingerSeq` implementation took the longest to finish, and consumed the most *PKG*/*DRAM* energy (100% in all three metrics). We can also see that the `JoinList` implementation took approximately 60% of the execution time and about 65% of the energy consumption for *PKG* and *DRAM*, compared with the `FingerSeq` implementation. The best performant implementation was `ListSeq`, consuming the least in all metrics (below 10% of the energy consumption of `FingerSeq`).

Overall, we observed that, for all data structure implementations provided by `Edison`, a higher execution time implies higher energy consumption in both the package and DRAM domains and, consequently, a lower execution time leads to a lower energy consumption in both domains. From this analysis, it is possible to answer **RQ1**.

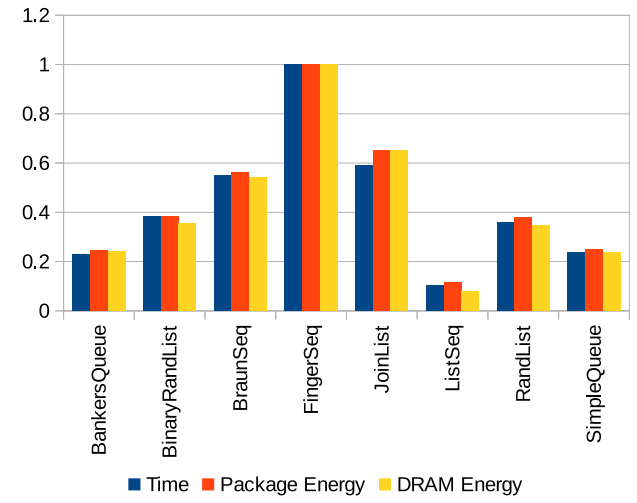

Figure 2: Relative Time, *PKG* and *DRAM* energy consumption results, for the `iterator` operation, for `Sequences`.

RQ1 the execution time and energy consumptions of the package/DRAM are directly proportional. Indeed, we observed that lower execution times lead to lower energy consumption in both domains.

Another goal of this first experiment is to assess the relative impact of both domains, *PKG* and *DRAM*, on the overall energy consumption. We seek to verify if there is a trade-off between the package and DRAM energy consumptions.

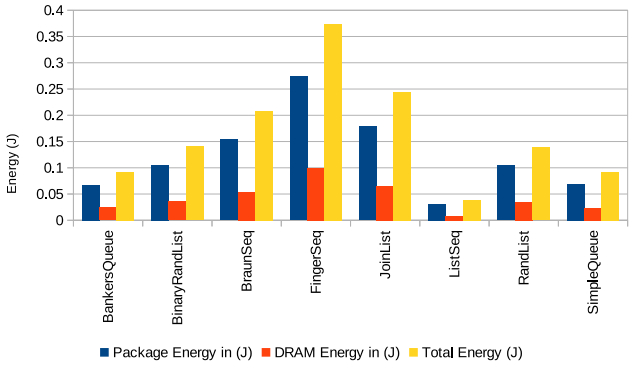

Figure 3: *PKG*, *DRAM* and Total energy consumption results, for the `iterator` operation, for the `Sequences` abstraction.

Figure 3 shows the package, DRAM and total energy consumption absolute values for the same example operation.

From the recorded data, energy consumption is highly correlated with execution time (a Pearson correlation coefficient above 0.9) across all Edison's abstraction implementations.

This observation evidences that a trade-off between the *PKG* and *DRAM* domains does not occur (in the context of the `Edison` `Haskell` library) and the most energy-efficient operations are the fastest ones.

Analyzing the values for all operations and implementations, we can conclude that:

- for `Sequences`, the impact of the DRAM energy consumption represents between 18.3% and 29.4% of the total energy consumption (which means the *PKG* energy weighs from 70.6% to 81.7%);
- for `Sets`, the DRAM impact varies from 14.8% to 30.4% (which means that the *PKG* varies from 69.6% to 85.2%);
- for `Associative Collections`, the variation of the DRAM energy consumption impact is between 19.3% and 30.9%;
- and for `Heaps`, that same variation is situated between 19.2% and 30.9%.

Therefore, the answer to the **RQ2** is:

RQ2 the impact of DRAM energy consumption, across all implementations of the three `Edison`'s abstractions, is between 14.8% and 30.9% of the total energy consumption.

4.3 Time and Energy impact of compilation optimizations

A second experiment was devised to evaluate whether the relationship between execution time and energy consumption, for the package and DRAM domains, would be impacted by the different code optimization levels provided by a modern `Haskell` optimizing compiler.

We considered GHC (Glasgow Haskell Compiler) as it is the "de facto" standard compiler for `Haskell`, and the focus of most compiler optimization research targeting lazy functional programming languages. GHC provides (among other) a set of -O* options to specify packages of optimization flags:

- O0, meaning no optimizations will be performed;
- O1, the default used by the Cabal build tool (which we use), in which short meaningful optimizations are performed.
- O2, which performs all optimizations, without compilation time restrictions.

It is important to note that in the documentation for this GHC version, the authors warn that the -O2 option would probably not be able to produce better code than -O1.

For this experiment we executed the benchmark suite using the three optimization levels (using the default O1 corresponds to the same results as in the previous experiments).

Figures 4 and 5 are examples of the results obtained, showing absolute values for execution time, package and DRAM energy consumptions for each optimization level.

The overall results mirror previous findings: across all abstractions and implementations a lower execution time results in a lower energy consumption in both domains (*PKG* and *DRAM*) (and higher execution times leads to higher energy consumption).

Despite reaching the same conclusion, there were some surprising results in the relationship between the values obtained for the different optimization levels.

For some operations in some implementations, the non-optimized code (O0) took less time and energy to execute than the same code

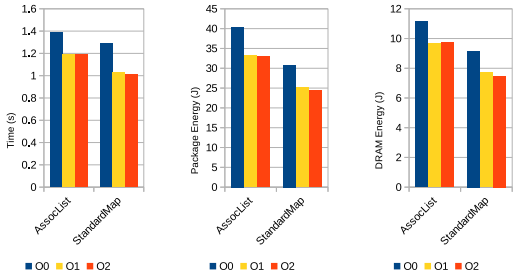

Figure 4: Time, *PKG* and *DRAM* energy consumption results, for the `addAll` operation, for the `Associative Collections` abstraction, using -O* compilation options.

compiled with optimizations enabled (options O1 and O2). An example of this behavior can be observed in Figure 5, for the `Sets` abstraction, for the `toArray` operation, for the `StandardSet` implementation. These are examples of the warning documented by the GHC authors regarding the lack of consistency of optimization levels regarding O2, but also of the same concerning O1.

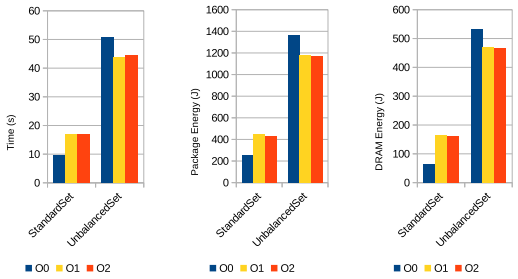

Figure 5: Time, *PKG* and *DRAM* energy consumption results, for the `toArray` operation, for the `Sets` abstraction, using -O* compilation options.

Using these results, we can answer **RQ3**:

RQ3 whenever the optimization options of the compiler (GHC) reduce the execution time, the energy consumption of the package and DRAM domains will also decrease. However, in some cases, enabling optimizations can increase execution time and consequently energy consumption.

4.4 Guidelines for developers

From the conclusions of the above experiments (and our previous work[16]) we can make the following recommendations to developers:

- To minimize energy consumption, developers should choose data structure implementations that run as fast as possible for the most common operations over those data structures.
- Optimizations that are able to reduce the execution time of applications should also be used for reducing the energy consumption.

These recommendations are based on the `Edison` data structures library, in which a fast execution for a specific operation over those data structures is a sound indicator for lower energy consumption.

But developers must be aware that higher optimization levels (including the default) are not guaranteed to produce faster executing code. As we have seen in the Figure 5 not optimizing can be faster and more energy efficient.

5 THREATS TO VALIDITY

In this work, as in our previous work [16], we focused on the `Haskell` programming language. `Haskell` is very different from most programming languages, as it is a combination of a purely functional language and a lazy language. We do not claim that our results apply to any other programming language. However, our conclusions are shared with another studies [10, 19] that focus on the Java language.

Another limitation of this work is that we considered only one hardware platform. While different processor speedups, DRAM configurations and even different architectures have a large influence on these results, the chosen hardware is representative of hardware commonly used.

Our experiments are based on a micro-benchmark suite, comprised of different operations over specific data structures abstractions, instead of on real world applications, for which the results could be different. The reason for choosing micro-benchmarks is that real-world applications have other components that impact energy-consumption, which were not interesting for this study. Additionally, real-world applications have very different operation patterns, and the conclusions drawn from one application would not apply to a different application. Using micro-benchmarks, it is possible to choose the best implementation, based on the profile of operations used by a target real-world application.

Our study focused on `Edison`, a `Haskell` library that provides several implementations for different data structures abstractions. There are other data structure implementations available for `Haskell`, for which results could vary, but not drastically.

From different methods of collecting energy measurements, we selected RAPL, which provides estimations for different power-related characteristics. This choice was because of its wide availability and according to the asserted accuracy of RAPL [4, 9].

Another decision we made was to use GHC and its three levels of optimization. Another `Haskell` compiler or different optimization configurations could have been tested. GHC is the most popular and developed compiler. Given this, evaluating every simple compilation option is unfeasible. As such, we considered the aggregated optimization options available (O0, O1 and O2).

6 CONCLUSIONS

Energy efficiency has become a pressing concern for modern day hardware/software developers. In various domains (e.g. data centers, mobile battery powered devices) energy saving is a key factor.

Energy efficiency comes not only from the hardware, but also from the software running on the hardware. As such, in order to write energy-efficient software, developers need information on the energy consumption profiles of different artifacts in computing.

This study focuses on one kind of artifact: different data structure implementations.

We revisit our previous work, in which we evaluate the energy consumption behavior of `Haskell` data structures (exposed by the `Edison` library). This time, we focus on the role of DRAM energy consumption within the total energy consumption and its correlation with the execution time. We use micro-benchmarks and collect values from different Intel RAPL domains, specifically *PKG* and *DRAM*.

To understand such role, we have formulated the following research questions:

RQ1 How does execution time influence Package energy and DRAM energy consumptions?

Answer: The execution time and energy consumptions of the Package/DRAM are directly proportional.

RQ2 What is the impact of DRAM energy consumption on the overall energy consumption?

Answer: The impact of DRAM energy consumption in our experiments ranges between 14.8% and 30.9% of the total energy consumption.

Given the direct correlation between execution time and energy consumption (regardless of the domain), we validated whether compiler optimizations also improved energy efficiency, resulting in the following research question:

RQ3 How do the different compilation optimization options, available in a modern `Haskell` optimizing compiler, affect the execution time and Package/DRAM energy consumptions?

Answer: Whenever the optimization options of the compiler reduce the execution time of the produced code, then the energy consumption of the Package and DRAM domains also decreases. However, optimizations can make a program slower, consequently increasing the energy consumption.

Based on this work, we present guidelines for developers to follow in order to reduce the energy footprint of their software applications. We are also making public the results of our per-operation microbenchmarks (at green-haskell.github.io), which can be used by developers to choose the most energy-efficient data-structures according to their operation usage. For instance, if an application spends most of its time performing the `iterator` operation on the `Sequences` abstraction, the developer should use a `ListSeq` implementation.

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).

The authors would also like to thank Vasco T. Vasconcelos for his comments on an early draft.

REFERENCES

[1] Shuhaizar Daud, R.Badlishah Ahmad, and Nukala S. Murhty. 2008. The effects of compiler optimizations on embedded system power consumption. (12 2008), 6 pages.

[2] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*. IEEE, 189–194.

[3] Erik D. Demaine, Jayson Lynch, Geronimo J. Mirano, and Nirvan Tyagi. 2016. Energy-Efficient Algorithms. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS '16)*. 321–332.

[4] 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.

[5] Robert Dockins. 2015. EdisonAPI package. (2015). http://hackage.haskell.org/package/EdisonAPI-1.3

[6] Robert Dockins. 2015. EdisonCore package. (2015). http://hackage.haskell.org/package/EdisonCore-1.3

[7] Hayden Field, Glen Anderson, and Kerstin Eder. 2014. EACOF: A Framework for Providing Energy Transparency to enable Energy-Aware Software Development. *CoRR* abs/1406.0117 (2014). arXiv:1406.0117 http://arxiv.org/abs/1406.0117

[8] Alcides Fonseca and Bruno Cabral. 2017. Understanding the impact of task granularity in the energy consumption of parallel programs. *Sustainable Computing: Informatics and Systems* (2017).

[9] 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.

[10] 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.

[11] Pedro Rangel Henriques and David Branco. 2016. Impact of GCC optimization levels in energy consumption during program execution. (2016), 20-26 pages.

[12] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. In *Proceedings of the 2011 15th Workshop on Interaction Between Compilers and Computer Architectures (INTERACT '11)*. IEEE Computer Society, 63–70.

[13] Haskell in Industry. 2018. (2018). https://wiki.haskell.org/Haskell_in_industry [Online; accessed 03-February-2018].

[14] Mahmut Kandemir, Vijaykrishnan Narayanan, and Mary Jane Irwin. 2002. Compiler Optimizations for Low Power Systems. (01 2002).

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

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

[17] Stephen O'Grady. 2014. The redmonk programming language rankings: June 2017. *RedMonk. com* (2014).

[18] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2013. Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms. *CoRR* abs/1303.6485 (2013). arXiv:1303.6485 http://arxiv.org/abs/1303.6485

[19] Rui Pereira, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. *CoRR* abs/1602.00984 (2016). arXiv:1602.00984 http://arxiv.org/abs/1602.00984

[20] 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*. ACM, 256–267.

[21] Gustavo Pinto and Fernando Castor. 2017. Energy efficiency: a new concern for application software developers. *Communications of the ACM* 60, 12 (2017), 68–75.

[22] Cluster Green Software. 2015. (2015). http://www.clustergreensoftware.nl/english/ [Online; accessed 03-October-2016].

[23] Joseph Zambreno, Mahmut Taylan Kandemir, and Alok Choudhary. 2002. *Enhancing compiler techniques for memory energy optimizations*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 2491. Springer Verlag, Germany, 364–381.