

Tools Supporting Green Computing in Erlang

Gergely Nagy
Áron Attila Mészáros
nagygeri97@caesar.elte.hu
archy@caesar.elte.hu

ELTE, Eötvös Lóránd University, Budapest
Faculty of Informatics, 3in Research Group, Martonvásár
Hungary

István Bozó
Melinda Tóth

bozoistvan@caesar.elte.hu
tothmelinda@caesar.elte.hu
ELTE, Eötvös Lóránd University, Budapest
Faculty of Informatics, 3in Research Group, Martonvásár
Hungary

Abstract

Energy efficiency is one of the key aspects of modern software. Therefore, we present a tool for measuring the energy consumption of Erlang programs. Using this tool, we measured the energy consumption of different basic language elements, such as data structures, higher-order functions and parallel language constructs. Based on the results of our measurements, we present refactorings that may help to decrease the energy consumption of Erlang software. The refactorings are part of the well-known RefactorErl static analyser and transformer framework for Erlang.

CCS Concepts • **Software and its engineering** → *Software performance; Software maintenance tools.*

Keywords green computing, refactoring, energy consumption, energy efficiency

ACM Reference Format:

Gergely Nagy, Áron Attila Mészáros, István Bozó, and Melinda Tóth. 2019. Tools Supporting Green Computing in Erlang. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang (Erlang '19)*, August 18, 2019, Berlin, Germany. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3331542.3342570>

1 Introduction

Energy efficiency and energy consciousness are becoming more and more widespread. Therefore, we need to consider energy efficiency even when creating software. Our aim is to create different refactorings that may help developers transform existing Erlang code bases to be more energy efficient.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Erlang '19, August 18, 2019, Berlin, Germany

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6810-0/19/08...\$15.00

<https://doi.org/10.1145/3331542.3342570>

In our previous work [7], we created a tool for measuring the energy consumption of Erlang programs. The tool is based on Running Average Power Limit (RAPL) [8], a tool created by Intel for extracting energy consumption information from the processor. The first prototype of our tool had a command-line interface for measuring energy consumption.

With the help of this tool, we investigated the energy consumption of a few complex algorithms, such as the N-queens problem and sparse matrix multiplication in order to argue about the energy behaviour of Erlang applications. The results of these measurements showed that using higher-order functions and different data structures can affect the energy consumption of Erlang programs. However, it also turned out that measuring complex algorithms have drawbacks as well.

In this paper, we present the enhanced version of the former tool, called GreenErl (Section 2), and evaluate the results of measuring more fundamental language constructs (Section 3). We also investigate the energy consumption differences of programs compiled with the BEAM and the HiPE compilers (Section 4). Finally, we propose and define some refactoring steps that may help to improve the energy efficiency of the code (Section 5). These refactorings are integrated into the RefactorErl tool. The details of our analysis can be found in [6].

2 Tool for Measuring the Energy Consumption

Based on our first experiments, we created a tool, GreenErl, for measuring the energy consumption of Erlang functions. GreenErl consists of three main modules: a Python-based GUI, an Erlang module for evaluating the measured functions and a C program for reading the RAPL registers [15]. Figure 1 shows how these components communicate and work together to provide the desired functionality.

The Python-based graphical user interface provides a way to organize the measurements to queues. Each measurement consists of three main parts. These are:

- the Erlang source file of the measured module,
- input values for the measured function(s) (optional),
- a directory for the results.

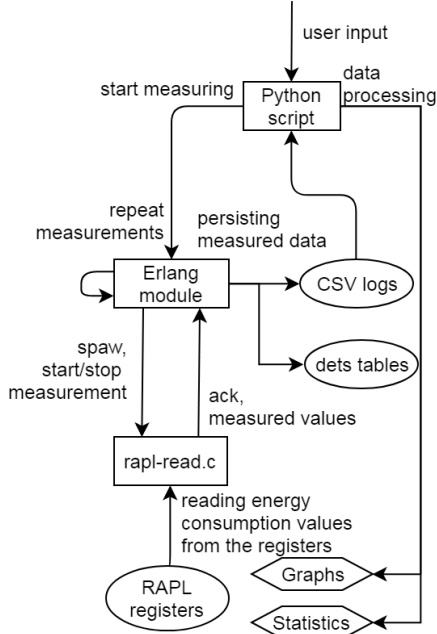


Figure 1. Flowchart of the components of the GreenErl framework

Additionally, it can also be specified how many times we would like to repeat the measurements. The list of functions to measure from the given module can also be specified.

The specified inputs are passed directly to the functions to be measured. However, the inputs can be generated by exporting and implementing the `generate_input` function in the selected module. If this function is provided, the framework uses it to transform the specified inputs. An example is shown in the following code snippet of how to use function `generate_input` to create lists of length `N`.

```
generate_input(N) -> [lists:seq(1,N)].
```

GreenErl takes all the functions exported from the module and evaluates them using the generated inputs.

After a measurement suite is evaluated, GreenErl creates log files in the specified directory that contain the average of the results calculated from the repeated measurements. These logs contain energy consumption information of both the RAM and CPU, as well as runtime values.

Finally, the GUI of GreenErl also provides a way to visualize the result of each measurement on a graph.

3 Language Constructs

Based on the previous investigations [7] we identified some key areas, that we needed to check. These are the following:

- Data structures,
- Higher-order functions,
- Parallel language constructs and
- Algorithmic skeletons.

In this section, we summarize the measured functions and give an overview of our findings regarding the key areas.

All measurements in this paper were conducted using a computer with an Intel Core™ i7 6700HQ CPU, 8GB of DDR4 RAM, running Ubuntu 18.04 LTS operating system, and we used Erlang/OTP 21.2.6.

3.1 Data Structures

We investigated data structures that can be used to store key-value pairs, such as proplists, maps and dictionaries. For each structure we measured the energy cost of creating an instance of the given structure, as well as some basic operations on the data structures, such as finding, updating or deleting an element belonging to a given key.

Creating When comparing the cost of creating these data structures, we found the creation of lists to be the cheapest in terms of the consumed energy. This is in line with the expected results. The interesting fact is that if someone needs a map or a dictionary, it is more energy efficient to create a list first and use either `maps:from_list/1` or `dict:from_list/1` to convert the proplist to the associative data structure. For dictionaries, this could mean up to 50% energy savings, while for maps we can save up to 25% of energy compared to inserting the elements one by one to the associative structure.

Even though using dictionaries consumes less energy, but creating them is far more expensive than creating a map.

Figure 2 shows the measured energy consumption values for different ways of creating the investigated containers.

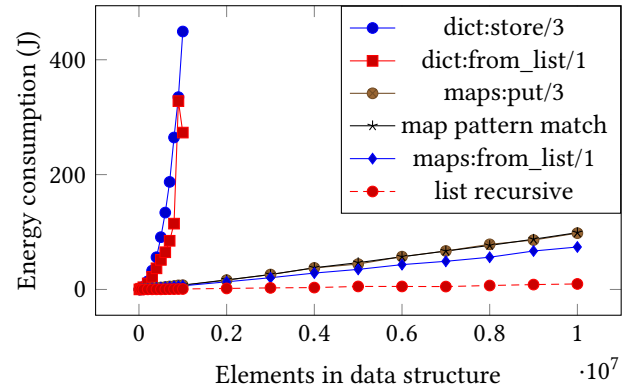


Figure 2. Creating the investigated data structures

Finding Looking up an element in a list or an associative container has different algorithmic complexity. Because of this, the expectation was that finding an element in a list consumes much more energy than using either a dictionary or a map to do so. This expectation has been confirmed by the measurements, but we also found that using `lists:keyfind/3` instead of `proplists:get_value/2` decreases energy consumption by a factor of three.

Comparing the energy consumption of finding in maps and dictionaries, we found that finding in a dictionary consumes about 10 times more energy than finding in a map. Despite this huge difference, both data structures consumed an insignificant amount of energy compared to lists, as a result of the previously mentioned different algorithmic complexity.

Other Operations As mentioned previously, we also measured other operations on all three data structures. In all of these cases, we found that dictionaries are less energy efficient than maps, and both maps and dictionaries still consume a negligible amount compared to lists. Table 1 contains the energy cost of some operations on a list expressed as multiples of the energy consumed by `lists:keyfind/3`.

Table 1. Energy consumption of different operations

Function \ Length	10^5	10^6	10^7	Average
<code>lists:keyfind/3</code>	1	1	1	1
<code>lists:keystore/4</code>	7.7	5.3	6.7	6.5
<code>lists:keytake/3</code>	10.1	8.7	6.2	8.3
<code>maps:from_list/1</code>	318.1	302.2	322.3	314.2

3.2 Higher-Order Functions

Higher-order functions (HOFs) are an essential part of functional programming languages, and they are omnipresent in Erlang code bases as well. We wanted to find out if higher-order function calls, such as calls to `lists:map/2` or `lists:filter/2` cause any overhead in energy consumption.

We investigated the two HOFs mentioned before and also the `lists:filtermap/2` function. For each higher-order function, we created our own implementation and also tried using list comprehensions and defining new, specific recursive functions to get rid of higher-order function calls. The results showed the same pattern for all three functions, so here we only present the results for the `map` function. The results are shown in Figure 3.

We found that no matter which implementation of the higher-order function we used, or whether we used implicit or explicit fun-expressions, the higher-order function calls always consumed more energy than when we eliminated them. List comprehensions and the recursive function definition used almost the same energy, and they were more efficient than HOF calls.

3.3 Parallel Language Elements

We also investigated some parallel language constructs. Our main goal was to find the cost of sending the data structures shown in Section 3.1.

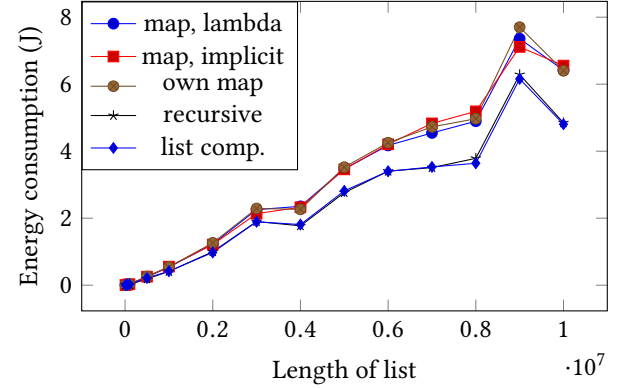


Figure 3. Different versions of the map functionality

We found that it was always more efficient to send maps than lists, while the results for dictionaries were inconclusive. The energy consumption values of sending a dictionary had vast fluctuations. Sometimes it seemed to be more efficient than maps, in other cases the results showed that it consumed even more energy than sending a list.

We also investigated if it is worth to partition a list before sending, and only sending smaller chunks of the list at once. We found that using small chunk sizes ranging from 1 up to 10 increased energy consumption. As the size of the chunks increased, the energy consumption came really close to when we sent the entire list at once. We concluded that if a program uses a producer-consumer pattern, it is worth to accumulate elements of a list up to a few hundred elements and send those elements together. The gains in energy efficiency from accumulating more elements are negligible.

3.4 Algorithmic Skeletons

The final area we investigated was the use of algorithmic skeletons for parallelization. Here we investigated a simple task farm implementation. We compared it to the task farm available in the Skel library [12]. We found that for small computations using the Skel implementation provides a quite significant overhead in energy consumption, but as the computational load on the workers increase, this overhead becomes more and more negligible.

We also investigated the effect of the number of processes on energy consumption. The computer we used for the measurements provides 8 logical cores. We found that starting the same number of workers in the task farm as there are logical cores in the CPU is the most energy efficient solution. An interesting finding is that when only 2 workers were started, the runtime of the program was significantly higher than when 8 workers were started, but the energy consumption values were almost exactly the same.

4 HiPE

All previous measurements were conducted by evaluating functions that were compiled by the standard BEAM compiler [1]. We wanted to repeat the measurements using the HiPE (High-Performance Erlang) compiler [11] as well.

All measurements in Section 3 have been repeated using the HiPE compiler. In this section, we show the effects of HiPE using the measurements for the `lists:map/2` higher-order function. All the conclusions regarding the energy efficiency of HiPE hold for the other measurements as well.

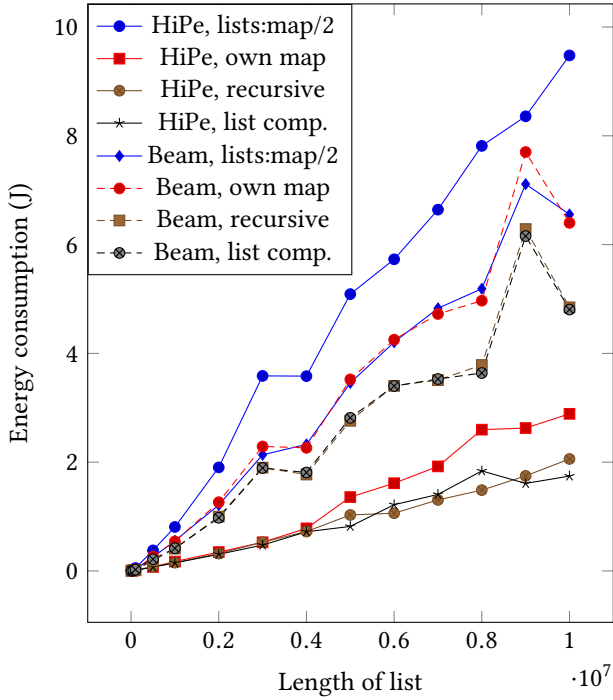


Figure 4. Differences between BEAM and HiPE

The results of using both BEAM and HiPE are shown on Figure 4. Compared to BEAM, HiPE decreased energy consumption in most cases, on average by about 50%. This decrease in energy applied to the map versions where we either used recursion, a list comprehension or created our own map implementation.

The only case when using HiPE actually increased energy consumption is when we called the `lists:map/2` function. The reason for this is that the `lists` library module was compiled using BEAM, while our module was compiled using HiPE. This caused the Erlang abstract machine to perform a mode switch between byte code and native code for every element of the input list, when applying a HiPE compiled function passed as a parameter to the BEAM compiled `lists:map/2` function. This phenomenon is documented to have a negative effect on performance [11], and now we have confirmed that mode switches also have a negative effect on energy consumption.

Looking at the versions that used the HiPE compiler correctly, we conclude that the relations between different versions are the same as when using BEAM. Generally, higher-order function calls consumed more energy than using list comprehension or recursion, even with HiPE. What we found is that when the length of a list is less than 400 thousands elements, HiPE can heavily optimize higher-order function calls, so that they consume almost the same energy as the recursive or list comprehension versions. However, this optimization disappears after the size of the list exceeds this limit. Beyond this threshold, applying higher-order functions cause an increase in energy consumption.

The fact that HiPE mostly preserves the relations between the different implementations is true for all conducted measurements. The extent of the decrease in energy consumption is varying. In general, it can be stated that HiPE decreases energy consumption when frequent mode switches are avoided.

5 Implemented Refactorings

Based on the presented results (Section 3), it is possible to propose some refactorings in order to reduce the energy consumption of Erlang code bases. We have implemented and validated the following refactorings using the RefactorErl [2, 13] tool¹.

RefactorErl is a static source code analysis and transformation tool for Erlang. The tool was designed for refactoring large scale applications, and at the same time, it became a useful platform for code comprehension.

This tool represents Erlang programs in a Semantic Program Graph. It provides an extensible framework for implementing user-defined refactorings by manipulating the syntax tree while the semantic consistency is automatically preserved by the analyser framework.

5.1 Transforming List to Map

The investigation of data structures showed that performing common operations on maps consumes much less energy than on lists. The only area in which maps proved to be less efficient is creating the data structure. Based on the energy consumption values of converting a list to a map, and the cost of basic operations on lists, shown in Table 1, we are able to give a limit to how many of each operation needs to be performed on a list for conversion to be worth it. In the case of finding an element, this limit is around 320, when updating it is around 50 and for deletions, it is around 40. This means that the cost of converting a list to a map is equivalent to, for example, 50 calls to the `lists:keystore/4` function.

These limits can be easily reached if the operations are performed on a list in a recursive function. In most of the cases, it should be beneficial to the energy consumption of

¹RefactorErl is available at plc.inf.elte.hu/erlang

a program to transform a recursive function that takes a proplist as its parameter to use a map instead.

The refactoring is able to transform calls to the following functions from the lists module to the equivalent calls using either map pattern matching syntax or functions from the maps module: keydelete/3, keyfind/3, keymember/3, keystore/4 and keytake/3.

We present only the context-dependent transformation of lists:keyfind/3 in detail. The transformation of all other functions can be done similarly. In the case of keyfind, we first need to decide if the result is matched to a two-tuple. If this is not the case, or the expression is the last in the current clause, then maps:find/2 must be used to simulate the behaviour of keyfind. Otherwise, we can use pattern matching, since badmatch error is thrown if the pattern matching fails when matching the result of keyfind to a two-tuple.

If the first parameter of keyfind is either an atom, an integer literal or a variable we do not need to extract and bind the expression to a separate variable. Otherwise, it is necessary to extract the expression to a separate variable, since in the case of pattern matching it may be an illegal expression and in the case of using maps:find it would be evaluated twice, which is not desired because of possible complex computations and side effects.

Validation To make sure these refactorings improve energy efficiency, we performed some measurements using the code example showed on Figure 5. The results showed that using the refactoring described above can significantly reduce the energy consumption of Erlang programs.

```
recursive([], _) -> ok;
recursive(L, K) ->
  {_, V} = lists:keyfind(K, 1, L),
  NL = lists:keystore(K, 1, L, {K, V+1}),
  {value, _, NL2} = lists:keytake(K, 1, NL),
  recursive(NL2, K+1).
```

(a) Before

```
recursive(L, _) when map_size(L) == 0 -> ok;
recursive(L, K) ->
  #{K:=V} = L,
  NL = L#{K=>V+1},
  {value, _, NL2} =
    case maps:take(K, NL) of
      error -> false;
      {V1, L1} -> {value, {K, V1}, L1}
    end,
  recursive(NL2, K+1).
```

(b) After

Figure 5. Transforming a recursive function to use maps instead of lists

5.2 Eliminating Higher-Order Functions

Based on the results regarding higher-order functions, we found that they are worth eliminating either by creating a new specialized recursive function or creating a list comprehension. We have implemented both transformations to eliminate map and filter calls. Some code examples showing the code resulting from the elimination of HOFs are shown on Figure 6.

Validation We have measured the effect of this refactoring on energy consumption. We found that for our test programs, after the elimination of HOFs by using our refactoring, energy consumption has decreased by 20-25%. The results of these measurements are shown on Figure 7.

```
f(L) -> Y = 12,
  lists:map(fun(X) ->
    A = X + Y, A := 0 end, L).
```

(a) Before

```
f(L) -> Y = 12,
  [begin A = X + Y, A := 0 end || X<-L].
```

(b) After - list comprehension

```
f(L) -> Y = 12, rec(L, Y).
rec([], _) -> [];
rec([X|T], Y) ->
  A = X + Y, [A := 0 | rec(T, Y)].
```

(c) After - recursion

Figure 6. Eliminating higher-order functions.

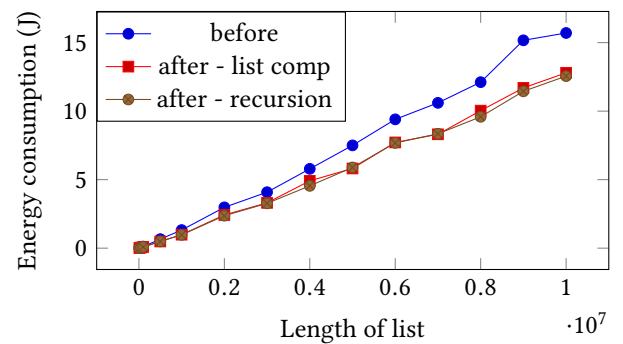


Figure 7. Differences between the refactored and non-refactored test programs.

6 Related Work

In recent years there has been lots of research done with regards to energy consumption and green computing [10].

There are specific results with regards to the energy consumption of software in mainstream languages, such as Java [9] and C++ [3].

For functional languages, we found that in Haskell the energy consumption is also measured using the RAPL tool and found that even really small code changes can lead to large differences in energy consumption [5].

Erlang is not a well researched language in terms of energy consumption. There was a talk given by Varjão at the Erlang Factory SF in March 2017, where he presented a RAPL based tool for measuring the energy consumption of Erlang programs [14], but this tool has not been made publicly available.

The authors of this paper have been submitted two scientific theses to the Scientific Student Association conference of the Eötvös Loránd University and also published a paper about the energy consumption of Erlang programs [7].

Regarding the use of refactorings to improve energy efficiency, there have been attempts to reduce the energy consumption of Android Apps by automatic refactorings [4].

7 Conclusion

We have created the GreenErl framework [6] for measuring the energy consumption of Erlang software. Using this tool, we measured the energy cost of the usage of different data structures, eliminating higher-order functions and parallel language constructs.

We found that in certain cases, it is worth to transform between maps and lists. Thus we have created a refactoring to transform recursive functions to use maps instead of lists.

Eliminating HOFs by either recursion or list comprehension also decreases energy consumption, so we have created a refactoring that makes it possible to eliminate map or filter HOF calls.

When investigating parallel language constructs, we found that sending a map is more efficient than sending a list. We also concluded that lists can be efficiently sent in chunks, but these chunks must not be too small.

When looking at algorithmic skeletons, we found that the Skel library has an overhead in energy consumption and that it is ideal to have the same number of processes as logical cores in the CPU.

We have also investigated the effect of using the HiPE compiler and found that it can decrease energy consumption but does not change the relation between different constructs.

Acknowledgments

This work was partially supported by the ÚNKP-18-1 New National Excellence Program of the Hungarian Ministry of Human Capacities and by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

- [1] Ericsson AB. 2019. Compilation and Code Loading. http://erlang.org/doc/reference_manual/code_loading.html. (2019). [Accessed: 2019.05.24].
- [2] I. Bozó, D. Horvács, Z. Horváth, R. Kitlei, J. Köszegi, M. Tejfel, and M. Tóth. 2011. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, ISBN 978-9949-23-178-2. TUT Press, Institute of Cybernetics at Tallinn University of Technology, Tallin, Estonia, 138–148.
- [3] Alexander Chatzigeorgiou. 2003. Performance and power evaluation of C++ object-oriented programming in embedded processors. *Information and Software Technology* 45 (03 2003), 195–201. [https://doi.org/10.1016/S0950-5849\(02\)00205-7](https://doi.org/10.1016/S0950-5849(02)00205-7)
- [4] L. Cruz, R. Abreu, and J. Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, Buenos Aires, Argentina, 205–206. <https://doi.org/10.1109/MOBIESoft.2017.21>
- [5] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. 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. IEEE, Suita, Japan, 517–528. <https://doi.org/10.1109/SANER.2016.85>
- [6] A.A. Meszaros and G. Nagy. 2019. Measuring the energy consumption of Erlang programs and energy conscious refactorings. Scientific Thesis, Scientific Student Association Conference, ELTE-IK, Budapest, Hungary, Received 1st prize. (2019).
- [7] A.A. Meszaros, G. Nagy, I. Bozo, and M. Toth. 2018. Towards Green Computing in Erlang. *Studia Universitatis Babeş-Bolyai Informatica* 63, 1 (2018), 64–79. <https://doi.org/10.24193/subbi.2018.1.05>
- [8] Srinivas Pandravad. 2014. RUNNING AVERAGE POWER LIMIT - RAPL. <https://01.org/blogs/2014/running-average-power-limit---rapl>. (2014). [Accessed: 2019.03.27].
- [9] Rui Pereira, Marco Couto, João Saraiva, Jácume 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, New York, NY, USA, 15–21. <https://doi.org/10.1145/2896967.2896968>
- [10] Vimal P. Parmar, Apurva K. Pandya, and Dr. CK Kumbharana. 2011. Optimization of Energy Usage for Computer Systems by Effective Implementation of Green Computing. *International Journal of Advanced Networking Applications (IJANA)* Sp. issue of National Conference held at Marvadi Educational Institution (2011), 87–90. <https://www.ijana.in/Special%20Issue/16.pdf>
- [11] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl. 2003. All You Wanted to Know About the HiPE Compiler: (but Might Have Been Afraid to Ask). In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang (ERLANG '03)*. ACM, New York, NY, USA, 36–42. <https://doi.org/10.1145/940880.940886>
- [12] Skel Tutorial. 2014. Available at <http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html>. (2014). [Accessed: 2019.05.24].
- [13] M. Tóth and I. Bozó. 2012. Static Analysis of Complex Software Systems Implemented in Erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFPP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451–514, Springer-Verlag, ISSN: 0302-9743. (2012).
- [14] Filipe Varjão. 2017. Measuring Erlang energy consumption, and why this matters. <http://www.erlang-factory.com/sfbay2017/filipe-varjao.html>. (2017). [Accessed: 2018.03.25].
- [15] Vincent M. Weaver. 2015. Reading RAPL energy measurements from Linux. <http://web.eece.maine.edu/~vweaver/projects/rapl/>. (2015). [Accessed: 2019.03.27].