Energy Consumption Measurement of C/C++ Programs Using Clang Tooling

MÁRIO SANTOS and JOÃO SARAIVA, University of Miñho
ZOLTÁN PORKOLÁB and DÁNIEL KRUPP, Ericsson Ltd.

The green computing has an important role in today's software technology. Either speaking about small IoT devices or large cloud servers, there is a generic requirement of minimizing energy consumption. For this purpose, we usually first have to identify which parts of the system is responsible for the critical energy peaks. In this paper we suggest a new method to measure the energy consumption based on Low Level Virtual Machine (LLVM)/Clang tooling. The method has been tested on 2 open source systems and the output is visualized via the well-known Kcachegrind tool.

Categories and Subject Descriptors: B.8.2 [**Performance and reliability**]: Performance Analysis and Design Aids—*Energy consumption measurement*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation / methodology*

General Terms: Energy consumption, Human factors

Additional Key Words and Phrases: Energy consumption, Code Instrumentation, Visualization

1. INTRODUCTION

While in the previous century computer manufacturers and software developers primary and single goal was to produce very fast computers and software systems, in this century this has changed: the widespread use of nonwired but powerful computer devices is making battery consumption/lifetime the bottleneck for both manufacturers and software developers. Unfortunately there is no software engineering discipline providing techniques and tools to help software developers to analyze, understand and optimize the energy consumption of their software! As a consequence, if a developer notices that his/her software is responsible for a large battery drain, he/she gets no support from the language/compiler he/she is using. The hardware manufacturers have already realized this concern and much work in terms of optimizing energy consumption by optimizing the hardware has been done. Unfortunately, the programming language and software engineering communities have not yet completely realize that bottleneck, and as consequence, there is little support for software developers to reason about energy consumption of their software. Although is the hardware that consumes energy, the software can greatly influence such consumption [Bener et al. 2014], very much like a driver that operates a car influences its fuel consumption.

In this paper we introduce an automated instrumentation-based method to measure the process level energy consumption for C/C++ programs. The source code is compiled by our Clang tooling based compiler to produce an instrumented code. The generated executable will measure the energy con-

Author's address: Mário Santos, João Saraiva, University of Minho, Departamento de Informática, Campus de Gualtar, 4710 057 Braga, Portugal, email: santos.mario125@gmail.com jas@di.uminho.pt; Zoltán Porkoláb, Dániel Krupp, Ericsson Hungary Ltd., Irinyi J. út 4-20, Budapest, Hungary, H-1117, email: zoltan.porkolab@ericsson.com daniel.krupp@ericsson.com;

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

sumption and emit the results in machine usable format. We convert the output to the well-known Kcachegrind [Weidendorfer 2015] format.

This paper is structured as follows: In Section 2 we overview how we can measure the energy consumption of a processor. Our code instrumentation is based on the LLVM/Clang compiler infrastructure, which we discuss in Section 3. In Section 4 our instrumentation is described in details. The results are evaluated in Section 5. We overview the related work in Section 6. Our paper concludes in Section 7.

2. ENERGY MEASUREMENT ON PROCESS LEVEL

In this section we overview how one can measure the energy consumption of a processor using the Intel's RAPL interface. We also discuss our extensions to retrieve the necessary information in function-level.

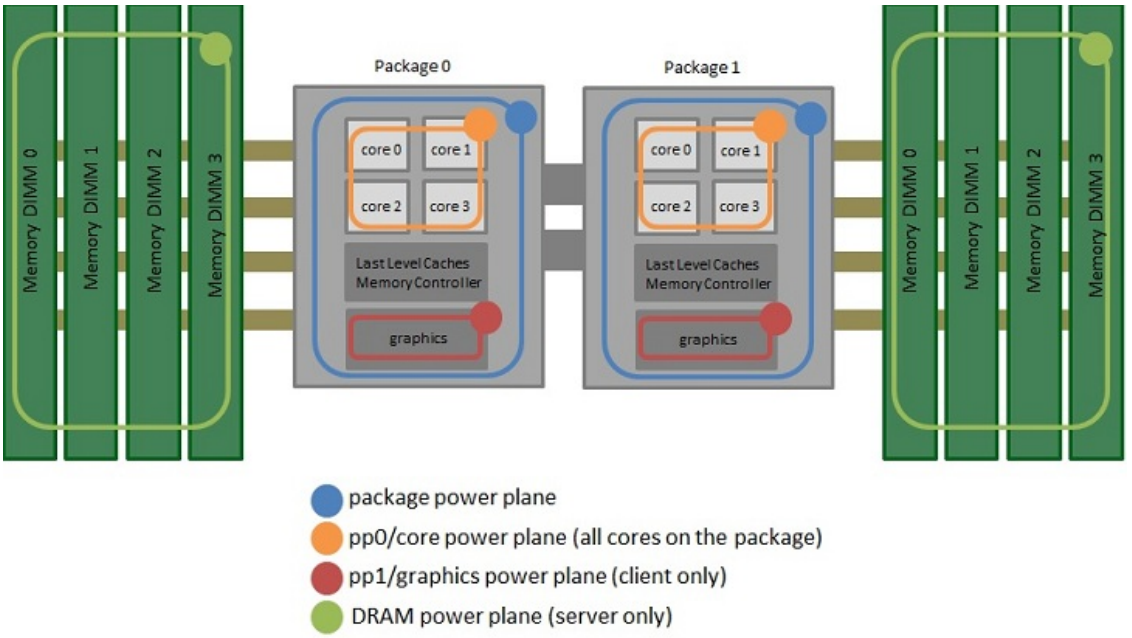


Fig. 1. Power domains for which power monitoring/control is available.

Originally designed by Intel, RAPL (Running Average Power Limit) [Dimitrov et al. 2015] is a set of low-level interfaces with the ability to monitor, control, and get notifications of energy and power consumption data of different hardware levels. It is supported in today's Intel architectures, like i5 and i7 CPUs. The architectures, that support RAPL, monitor energy consumption information and store it in Machine Specific Records (MSRs). These MSRs can be accessed by the Operating System.
As you can see by Fig.1, RAPL allows energy consumption to be reported in a practical way, by monitoring CPU core (pp0), CPU uncore (pp1) and DRAM separately.

Our extension of RAPL for C (CRAPL), can be viewed as a wrapper (example code below) to access the MSRs during the execution of a C/C++ programme. Through this interface we are able to have

an estimate of the power consumption in order to study which components (methods\functions) have absurd energy spikes in our source code.

```
CRapl rapl = create_rapl(0);
rapl_before(rapl);
doSomething();
rapl_after(0, rapl);
```

3. CLANG TOOLING

The LLVM project, started at the University of Illinois, is a collection of modular and reusable compiler and tool-chain [Lattner 2006]. LLVM has grown into an umbrella project and now includes various open source activities from compilers to static analysis. The flagship compiler for the LLVM project is Clang, the "native" compiler of LLVM. Clang supports C, C++, Objective-C and Swift languages in the advanced level [Groff and Lattner 2015]. The modular, object-oriented design of Clang make it ideal for research projects require compiler-level understanding of the source code [Lattner 2008]. Having a well-defined interface for building the Abstract Syntax Tree (AST), exploring it in various ways and even on-the-fly modify it, we can apply the tool-chain for instrumenting the source.

In the center of our activity is the Abstract Syntax Tree (AST). The AST contains all important information (even the formatting informations via the stored positions of every element). The structure of the AST is representing the logical structure of the original program. For example the node which belongs to a `for` loop has four children: a declaration statement to introduce the loop variable, a logical expression as loop condition, an iteration expression and the body. Note that the parentheses and the semicolons in the loop header are excluded.

In the AST there are different type of nodes such as `ForStmt`, `FunctionDecl`, `BinaryOperator`, etc. These types are organised to an inheritance hierarchy which has three roots: `Decl`, `Stmt` and `Type`. Since the fundamental part of build process is compilation of translation units, the type of the root node is `TranslationUnitDecl`.

One way of using the Clang AST is to visit its nodes [Horváth and Pataki 2015; Clang 2016]. The visitor design pattern can be used to reach every node of the tree and perform some action when the process comes to a given type of node. Clang compiler provides a very efficient way of tree traversal by `RecursiveASTVisitor` template class. Our visitor class has to inherit from this template class of which the template parameter is our class itself. The reason of this is that with this solution our class also becomes an AST visitor by the inheritance, but we do not have to pay for virtual function calls every time when running the given visitor function for the next AST node.

4. THE INSTRUMENTATION

Based on LLVM/Clang, our LibTooling tool starts by reading the input files and will run them up our FrontendAction. It will create an Abstract Syntax Tree (AST) with the parsed text of each file. For each of these generated trees we will recursively go through each node so we can make the necessary modifications to include our CRapl interface in the source code of the program that we want to analyse.

The main nodes to be visited are the following:

—**VisitFunctionDecl:** It visits all the nodes that are functions. If the analysed function does not refer to a header file and has the minimum number of statements (lines of code) that the user requested (-l = N) then the tool will insert the information of that function into an index (array) of functions that will suffer the respective modifications until the end of the recursive reading of their child nodes.

—**VisitIfStmt:** to maintain code consistency we need to insert braces in each If or Else statement that they are not already limited by them. This because we always need to insert a rapl_after before every single return statement of the given function.

—**VisitReturnStmt:**If it catches a return statement anywhere in the code for that function, it will insert a ”rapl_after” to end the analysis of the power consumption in that call.

—**VisitCallExpr:** When we tested for the first time the totality of the Plugin (Instrumentation + CRapl) we verified that there were functions to consume more energy than the main function itself, which is impossible since main is the first to be executed and the one that finishes the program. With this we realized that we were not handling recursive functions in the best way. So the best solution we found for this was to limit blocks of code before and after the recursive call (i.e rapl_after and rapl_before instrumentations for each of these calls). This was one of the biggest challenges until we came up with a good generic solution, regardless of what kind of recursive call it is.

In the end of crossing each tree, it will also insert the dependencies of the CRapl libraries and save the changes in the corresponding file (or create a new case the -o = ”example.cpp” flag is enabled).

When it finish all the instrumentation of the files, it will create the **index.txt** file with all the information of all the modified functions so that later they will be analysed by the CRapl:

```
0:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:102:1:printFile
1:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:138:1:error
2:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:145:1:fatalError
3:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:154:1:resolveEntity
4:/home/name/ClangRapl/xerces-c/tests/src/XSTSHarness/XSTSHarness.cpp:240:1:main
```

5. EVALUATION

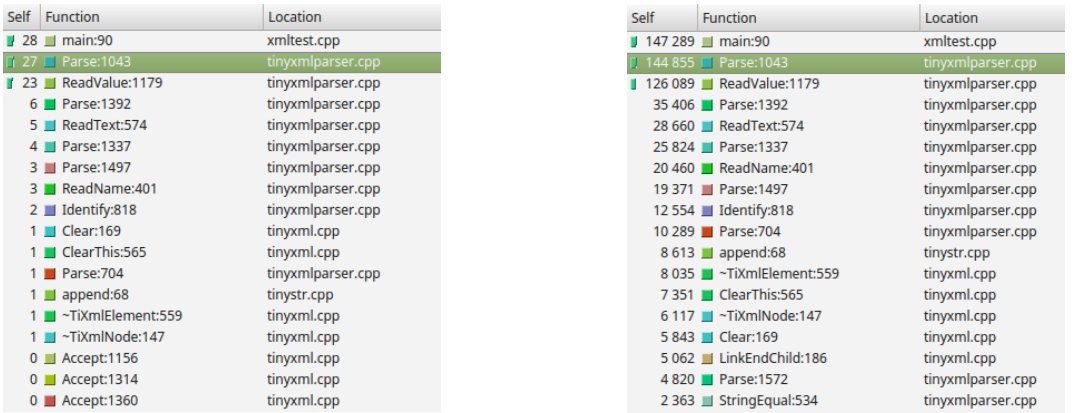
After the program execution, a file will be written with the values of the energy consumed by each function and the number of times it was called.

Currently our tool has been tested on two projects, tiny xml (small) and some xerces-c-3.1.4 (medium) samples / tests, both XML parsers written in C++.

Due to a high number of calls from each function (total of 181889), we received a huge overhead from CRapl. In order to remove this overhead first we would have to test only the main function to collect the true value of the energy consumed by the program and then run our script in python (mainly converts the format of the output file of the CRAPL in the Callgrind format to be read by Kcachegrind) to do the necessary normalizations (rule of three simple) of the energy results.

In this phase we will mainly present the results with this overhead, and since the Kcachegrind reads only number interns the units will be presented in **Milijoules** (10^{-3} J) mJ.

As we can see from Figure 2 when we run the CRapl in all functions of tinyxml we get a huge overhead (the number of calls to rapl is too big) but being the main objective of the project to realize which are the functions that spend more energy this process is necessary. Regardless, overhead does not prevent us from doing an analysis on the project (by percentage of energy consumed per function). We can verify that the **Parser** function (at line 1043) and its child nodes (including other functions) represent about 97% of the energy (package) consumed during the execution of the xmltest. Since the **Parser** function uses recursion, which means that at the beginning of each function you have to push the arguments to the stack and at the end pop them, it causes the function to have a longer execution



Self	Function	Location
28	main:90	xmltest.cpp
27	Parse:1043	tinyxmlparser.cpp
23	ReadValue:1179	tinyxmlparser.cpp
6	Parse:1392	tinyxmlparser.cpp
5	ReadText:574	tinyxmlparser.cpp
4	Parse:1337	tinyxmlparser.cpp
3	Parse:1497	tinyxmlparser.cpp
3	ReadName:401	tinyxmlparser.cpp
2	Identify:818	tinyxmlparser.cpp
1	Clear:169	tinyxml.cpp
1	ClearThis:565	tinyxml.cpp
1	Parse:704	tinyxmlparser.cpp
1	append:68	tinystr.cpp
1	~TiXmlElement:559	tinyxml.cpp
1	~TiXmlNode:147	tinyxml.cpp
0	Accept:1156	tinyxml.cpp
0	Accept:1314	tinyxml.cpp
0	Accept:1360	tinyxml.cpp

Self	Function	Location
147 289	main:90	xmltest.cpp
144 855	Parse:1043	tinyxmlparser.cpp
126 089	ReadValue:1179	tinyxmlparser.cpp
35 406	Parse:1392	tinyxmlparser.cpp
28 660	ReadText:574	tinyxmlparser.cpp
25 824	Parse:1337	tinyxmlparser.cpp
20 460	ReadName:401	tinyxmlparser.cpp
19 371	Parse:1497	tinyxmlparser.cpp
12 554	Identify:818	tinyxmlparser.cpp
10 289	Parse:704	tinyxmlparser.cpp
8 613	append:68	tinystr.cpp
8 035	~TiXmlElement:559	tinyxml.cpp
7 351	ClearThis:565	tinyxml.cpp
6 117	~TiXmlNode:147	tinyxml.cpp
5 843	Clear:169	tinyxml.cpp
5 062	LinkEndChild:186	tinyxml.cpp
4 820	Parse:1572	tinyxmlparser.cpp
2 363	StringEqual:534	tinyxmlparser.cpp

Fig. 2. Tinyxml results without and with overhead

time and consequently a higher consumption of energy [Fakhar et al. 2012].

With about 4926 instrumented functions we performed 6 tests of the **xerces** project and these were the results (package energy) obtained for some random functions with the overhead (in **Joules**):

Table I.

	Functions					
Tests	Main	Initialize:162	BuildRanges:93	getUniCategory:229	Terminate:328	match:995
CreateDOM	296,2	292	194,1	5,6	3,6	1,5
DOMCount	316,1	352,8	225,7	6,7	2,8	1,9
DOMPrint	318,3	303,9	198,5	6,6	3,3	1,2
DTest	973	329,5	210,1	6,6	3,8	399,2
RangeTest	286	275,8	179,7	5,5	3,2	0,75
Traversal	335,9	331,1	215,6	6,8	3,5	0,8
Total (%)	100%	75%	48%	1,5%	0,8 %	16%

From these results (Table I) we can not draw great conclusions except that most of the functions with high energy consumes are child nodes of the Initializer:162 because it is a function with only one call and with the highest percentage of energy spent. Also we can verify that the best and most interesting case to be analyzed in detail is the **DTest** because of the irregularity of the function **match:995**.

We will show some results about the most expensive functions in this test:

Table II.

Functions	Package Energy(J)	%	Calls	Energy/Calls (J)	Time (S)
Main:840	973	100%	1	973	90
testRegex:5393	569,3	58%	1	569,3	55
matches:517	542,9	56%	84	6,5	52
match:995	399,2	41%	52448	0,008	39
Initialize:162	329,5	34%	1	329,5	30

As we can see in Table II, the function **match:995** has a very high power consumption compared to the other tests due to be used many times (52448). So, from this table we can say that the function **match:995** is a function that spends less energy in relation to the rest (only 0.008J). Knowing that

this function is used mostly by the function **matches:517** we can conclude that 70-75% of the energy consumed by **matches:517** comes from **match:995**.

6. RELATED WORK

The energy consumption of software systems is a concern for computer manufacturers, software developers and regular users of computer devices (include users of mobile (phone) devices). While computer manufacturers began developing energy efficient hardware since the wide adoption of non wired computers, only recently energy became a concern for software developers as shown by the questions addresses on stack overflow report in [Pinto et al. 2014]. In fact, nowadays the energy efficiency of software systems is an intensive area of research. CPU manufacturers already provide frameworks to analyse the energy consumption of their processors, namely the energy estimators provided by Intel - the Intel RAPL [Dimitrov et al. 2015; Weaver et al. 2012; Hähnel et al. 2012; Liu et al. 2015; Fu et al. 2015] - or by Qualcomm - the Qualcomm TrepN framework [Qualcomm 2014; Bakker 2014]. Together with the use of external energy measurement devices, such as [McIntire et al. 2012; Bessa et al. 2016], it is possible to instrument and analyse the energy consumption of software systems.

Indeed, several techniques have been proposed to reason about energy consumption in software systems. For example, in the area of database systems, one of the first approaches to evaluate the energy consumption is presented in 2009 on the Claremont report [Agrawal et al. 2008], which expressed clearly the importance of taking into account the energy consumption from the very beginning of designing a database system. In [Goncalves et al. 2014] it was presented a technique to infer the most energy efficient query execution plan.

Researching and designing energy-aware programming languages is an active area [Cohen et al. 2012]. For example, there are works to analyse the energy efficiency of Java and Haskell data structures [Pereira et al. 2016; Lima et al. 2016; Pinto et al. 2016], to analyse how different coding practices influence energy consumption [Sahin et al. 2014a], energy aware type systems [Cohen et al. 2012], and to study the impact of code transformation [Brandolese et al. 2002], code obfuscation [Sahin et al. 2014b], and testing techniques [Li et al. 2014] software energy consumption. Other researchers have defined techniques to analyse energy consumption in Android mobile applications [Nakajima 2013; Couto et al. 2014; Li and Mishra 2016].

7. CONCLUSION

Our tool can be a good complement for C/C++ programmers who are interested in reducing the energy consumption of their programs. This is a theme that may not emerge much when small programs are used by a single machine but can have a positive environmental impact, or even reduce energy costs economically, when we talk about large scale projects used in servers or millions of personal computers around the world. In summary, we can say that the results are quite convincing in terms of high energy consumption by recursive functions and we can also notice that in sequential programs, the longer it takes a function to perform, the higher its energy consumption. As future work it would be interesting to shape CRAPL so that it would pick up the paths of the executed functions, so that we could study the results in more detail through the visualization of more illustrative graphs in Kcachegrind. In addition, collection of energy measurements from standard libraries functions would also give us a good perspective on which functions we should choose depending on how much data our implementation will receive.

ACKNOWLEDGMENT

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

REFERENCES

Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Hank F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. 2008. The Claremont Report on Database Research. *SIGMOD Rec.* 37, 3 (Sept. 2008), 9–19. DOI:http://dx.doi.org/10.1145/1462571.1462573

Alexander Bakker. 2014. Comparing energy profilers for android. In *21st Twente Student Conference on IT*, Vol. 21.

Ayse Basar Bener, Maurizio Morisio, and Andriy Miranskyy. 2014. Green software. *Ieee Software* 31, 3 (2014), 36–39.

Tarsila Bessa, Pedro Quintão, Michael Frank, and Fernando Magno Quintão Pereira. 2016. JetsonLeap: A Framework to Measure Energy-Aware Code Optimizations in Embedded and Heterogeneous Systems. In *Proc. of the 20th Brazilian Symposium on Programming Languages*, Fernando Castor and Yu David Liu (Eds.). Springer Int. Publishing, 16–30.

Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. 2002. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers* 11, 05 (2002), 477–502.

Clang. 2016. Doxygen documentation of recursive AST visitors. (2016). http://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html

Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. 2012. Energy types. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 831–850.

Marco Couto, Tiago Car cão, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages (Lecture Notes in Computer Science)*, FernandoMagno Quintão Pereira (Ed.), Vol. 8771. Springer International Publishing, 77–91.

Martin Dimitrov, Carl Strickland, Seung-Woo Kim, Karthik Kumar, and Kshitij Doshi. 2015. Intel® Power Governor. https://software.intel.com/en-us/articles/intel-power-governor. (2015). Accessed: 2015-10-12.

Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. 2012. Software level green computing for large scale systems. *Journal of Cloud Computing: Advances, Systems and Applications* 1, 1 (2012), 4. DOI:http://dx.doi.org/10.1186/2192-113X-1-4

Zhangjie Fu, Xingming Sun, Qi Liu, Lu Zhou, and Jiangang Shu. 2015. Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing. *IEICE Transactions on Communications* 98, 1 (2015), 190–200.

Ricardo Goncalves, João Saraiva, and Orlando Belo. 2014. Defining Energy Consumption Plans for Data Querying Processes. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing, BDCloud 2014, Sydney, Australia, December 3-5, 2014*. IEEE Computer Society, 641–647. DOI:http://dx.doi.org/10.1109/BDCloud.2014.109

J. Groff and C. Lattner. 2015. Swift's High-Level IR: A Case Study of Complementing LLVM IR with Language-Specific Optimization. (2015). Lecture at The ninth meeting of LLVM Developers and Users.

Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17.

G. Horváth and N. Pataki. 2015. Clang matchers for verified usage of the C++ Standard Template Library. *Annales Mathematicae et Informaticae* 44 (2015), 99–109. http://ami.ektf.hu/uploads/papers/finalpdf/AMI_44_from99to109.pdf

Chris Lattner. 2006. Introduction to the LLVM compiler infrastructure. In *Itanium Conference and Expo*.

C. Lattner. 2008. LLVM and Clang: Next Generation Compiler Technology. (2008). Lecture at BSD Conference 2008.

Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William GJ Halfond. 2014. Integrated energy-directed test suite optimization. In *Proc. of the 2014 Int. Symposium on Software Testing and Analysis*. ACM, 339–350.

Shaosong Li and Shivakant Mishra. 2016. Optimizing power consumption in multicore smartphones. *J. Parallel and Distrib. Comput.* (2016), –. DOI:http://dx.doi.org/10.1016/j.jpdc.2016.02.004

Luís Gabriel 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 *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 517–528.

Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*. Springer, 316–331.

Dustin McIntire, Thanos Stathopoulos, Sasank Reddy, Thomas Schmidt, and William J Kaiser. 2012. Energy-efficient sensing with the low power, energy aware processing (LEAP) architecture. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 2 (2012), 27.

Shin Nakajima. 2013. Model-based Power Consumption Analysis of Smartphone Applications.. In *ACESMB@ MoDELS*.

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*. ACM, 15–21.

Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 22–31.

Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. 2016. A Comprehensive Study on the Energy Efficiency of Javas Thread-Safe Collections. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 20–31.

Qualcomm. 2014. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler. (2014).

Cagri Sahin, Lori Pollock, and James Clause. 2014a. How do code refactorings affect energy usage?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 36.

Cagri Sahin, Philip Tornquist, Ryan McKenna, Zachary Pearson, and James Clause. 2014b. How Does Code Obfuscation Impact Energy Usage?. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE.

Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 262–268.

Josef Weidendorfer. 2015. KCachegrind. (2015).