Green smell identification for software energy efficiency optimization

Research paper

Corrado Ballabio
Universitá degli Studi di Milano-Bicocca
Milan, Italy
c.ballabio9@campus.unimib.it

Marco Bessi
CAST Software Italia
Milan, Italy
m.bessi@castsoftware.com

Francesca Arcelli Fontana
Universitá degli Studi di Milano-Bicocca
Milan, Italy
arcelli@disco.unimib.it

***Abstract*—Green IT tries to overcome the environmental issue by optimizing the energy efficiency in hardware and software execution. In this work we analyzed well-known Java bad practices related to performance, efficiency and robustness, we tested their energy consumption and we eventually identified a set of energy hostspots, named *green smells*. Then, we investigated the motivations behind the factors that determine this energy inefficiency, performing additional tests on the execution time, the memory usage and their bytecode instructions. Although we observed that execution time and memory usage do not have a direct correlation with energy consumption we found that specific bytecode instructions, the one related to objects instantiation and methods invocation, determine a peak in the power that green smells require. We eventually overcame a lack in the present literature, defining a methodology called *Greenability* index, that using the identified green smells is capable to quantify the energy inefficiency of sample applications. The Greenability index supports energy efficient software to identify pieces of software that could be optimized to require less CPU resources. This is not about software pieces that require a lot of resources but this is about software pieces that waste a lot of resources or use slowest functionality/instructions.**

***Index Terms*—Green software; Green smells; Energy efficiency; Energy metrics; SAST.**

I. INTRODUCTION

The capabilities of IT systems have grown constantly over the years, making our life easier and helping our jobs. However, as well as the several benefits, IT has been contributing to environmental issues. The massive amount of energy IT systems require for operating has an impact on global CO_2 total emission, and the constant growth IT had in recent past made this emission value no more negligible. The total amount of energy consumed is expected to perform a slight increase in the near future, growing 4% from 2014 to 2020, the same rate as the previous five years. Based on current trend estimations, U.S. data centers are projected to consume approximately 73 billion kWh in 2020 [1], corresponding to the level of electricity required by approximately 6 million average households.

This constant growth in energy consumption leads to two major consequences. A first direct effect is an increase in the overall maintenance cost: even though each facility has different values according several variables such as the performed operations or its usage rate, a typical large data center costs between $10-$25 million per year to operate [2]. The second effect is an increase in CO_2 emissions: the total amount of greenhouse gas production caused by ICT sector are projected to rise to 1.3 $GtCO_2$ (gigatonnes of equivalent carbon dioxide) by 2020, accounting for 2.3% on a global scale [3].

"Green IT is the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems efficiently and effectively with minimal or no impact on the environment" [4]. In particular, it can either refer to the research for reducing the environmental impact that IT products have during all their life cycle, and to Energy Efficiency improvement. Whereas research has primarily focused on improving hardware components of IT systems in order to reduce power draining, especially in CPU performances optimization and battery life elongation, it must be acknowledged that also software plays a central role in energy consumption. Software does not consume energy directly, however it is responsible of the computational behavior, affecting the energy consumption of the hardware underneath. Bad implementation choices can have a sort of propagation effects that leads to an improper use of power resources. For each Watt that the processor spends to elaborate information using a non-efficient algorithm, the total energy consumed by the system can be multiple times higher, due to over-work of drivers, memory, cooling system, back-up batteries and all the other auxiliary components. hence such issue must be carefully addressed by programmers and administrators. Hence, such issue must be carefully addressed by programmers and administrators.

While this causal relationship between software and the inducted hardware energy consumption has been proved [6], [7], [21], a common yet overly simplistic belief is that the main factor in energy consumption is the run time of applications. Accordingly to the *race-to-idle* approach [12], faster programs will theoretically consume less energy because the system can reach the idle state in faster time. Unfortunately it is quite hard to generalize the relationship between performance and energy consumption, especially for concurrent systems [11]. The nature of the problem to be solved, the technique used

to manage concurrent execution and the CPU clock frequency are some of the factors that can contribute in making this trade-off less obvious. Faster is not an absolute synonym for greener, that is why Green Software studies investigate the deeper reasons that make software energy inefficient.

II. RELATED WORKS

In their works [9] [10], Bessi et al. created a methodology to estimate software energy efficiency analyzing the application resource consumption and then defined a set of approaches to reduce the energy consumption of most inefficient applications. The memoization approach used in [8] could be applied to Java pure functions. They implemented an automatic Java bytecode analyzer to identify the best set of pure functions in order to be instrumented with the memoization technique and injected bytecode artifacts without any source code refactoring. Final results show that the approach provides a significant energy saving of over 86% of the overall value.

Multiple studies by Pinto et al. focus on the impact that different programming choices can have on the energy usage and performance. In [13] was performed an analysis on the main methods of 16 types of commonly used collection in the Java language, grouped in List, Set and Map. According to the operation implemented –insertion, removal or traversation–, or other "tuning knobs" such as the initial capacity and load factor, the results shows how energy consumption is not equal on every different configuration. In [14] they investigate how concurrent programming practices may have impact on energy consumption, finding that main factors are the choice of thread management constructs, the number of threads, the granularity of tasks, the size of the data, and the nature of data access. Bunse et al. [15] analyze the energy consumptions of different sorting algorithms. Since this class of algorithms is essential for managing data, results show that resources management plays a central role in power usage: in-place algorithms like insertion sort, even if less performing in specific settings, are more energy efficient than others.

Taking inspiration from the Fowler and Beck definition of code smells [16], which refers to bad code structures that are difficult to understand, read and maintain; the term *energy code smell* is often used to indicate energy hotspots, software-intrinsic elements or properties that have a measurable and significant impact on energy consumption, at any level of abstraction of the system architecture.

Related studies by Gottschalk et al. [17]–[19] operate towards improving energy savings on mobile phones. Smart energy management in this specific field assumes crucial importance: bad energy usage in mobile devices such as smartphones or laptops directly leads to a reduction in battery lifetime, in facts it represents one of the biggest complains in users negative reviews [20]. Gottschalk identifies a set of six energy code smells that cause an excessive consume of battery resources on android smartphones and then defines an automatic detection and reengineering approach for their resolution. Final results show that the refactoring of those smells can lead to energy savings up to 56.6%.

Another relevant study from Vetró et al. [21], investigates on energy code smells in C language. They choose this specific programming language for investigating power consumption in embedded systems. Results lead to the identification of 5 energy code smells, which cause an increase on energy consumption. The execution time of those smells is also monitored to investigate whether there is any correlation between this dimension and energy consumption. Final results show that energy code smells do not introduce any performance decrease, and the absence of a correlation between energy inefficiency and execution time.

III. BACKGROUND

CAST[1] is a progressive software vendor pioneering Software Intelligence and world leader in Software Analysis and Measurement (SAM), with a mission is to provide a standard unit of measure for those who build, buy or sell software.

CAST provides a software analysis services and technology helping to determine the overall hidden risk of applications, and to identify some of the root causes of current resilience issues, as well as any risk of future performance degradation. This assessment procedure uses the CAST Application Intelligence Platform[2] (AIP) to automatically scan the implementation of these applications to review monitor and check the architecture, design, and code against current industry best practices and known design flaws that may impact resilience. The AIP applies over 1000 engineering checks based on standards and measurements developed by the Software Engineering Institute (SEI), International Standards Organization (ISO), Consortium for IT Software Quality (CISQ), the Institute of Electrical and Electronics Engineers (IEEE), the Open Web Application Security Project (OWASP), the Common Weakness Enumeration (CWE) and the technology provider industry. The resulting analysis identifies specific flaws in the software and aggregates this information into metrics to objectively quantify the structural quality of the application.

The capability which makes CAST an emerging technology player in the static security as well as in the static software efficiency analysis is that of building a metamodel of the applications on which checks of efficiency, performance and injection can be made.

IV. GREEN SMELLS RESEARCH QUESTIONS

Current literature mainly focuses on assessing the impact that software can have when executed in specific system that strictly depend on the energetic consumption, such as embedded systems or mobile devices. The goal of our work is to remove those system-specific implications, and find the motivations behind the existence of software energy hotspots in generic, device-independent executions. Moreover, current literature lacks the definition of a metric for the practical assessment of the energy consumption of software applications.

[1]http://www.castsoftware.com/

[2]https://www.castsoftware.com/products/application-intelligence-platform

We will then use the results of our research to empirically define a new metric that can bridge this lack.

Our study is hence aimed to identify a set of Java code patterns that reflects higher energy consumptions. In order to reflect the positive impact that their detection can bring, we called them *green smells*. Our work follows two main research questions:

RQ1: which Java patterns represent an energy hotspot (i.e. which are green smells)?

RQ2: which are the main variables that are responsible of the increase in energy consumption?

Following these two research questions, the first part of our work focuses on the identification of the initial set of suspicious patterns. We then performed a series of test that identified firstly the set of green smells, then the possible reasons that generate a higher electrical consume. We took in consideration execution time, memory usage and bytecode instructions.

Last part of our work focused in the creation of a new index that assesses the energetic efficiency of given Java applications. This metric, named Greenability, was implemented using the Assessment Model of CAST Application Intelligence Platform (AIP) and the set of the detected *green smells*. The name of this metric, Greenabiility, is meant to reflect the quality of being environmental-friendly for given applications.

A. ENERGY HOTSPOT RESEARCH

For the creation of the initial set of patterns to test, we looked into the definition of some of the most common efficiency, robustness, and bad-practice violations reported in the documentation of CAST and other Automatic Static Analysis (ASA) tools.

CAST Documentation[3] offers a complete list of all the Quality Rules that are taken into account for the measurement of the Technical Quality indexes. These rules are divided into several categories depending on the Health Factor they influence: Changeability, Efficiency, Robustness, Security, Maintainability and Transferability.

Similarly, FindBugs [4]and PMD[5] report a wide range of software anomalies and flaws perform a static analysis.

We implemented a pair of Java methods that are representation of the identified violations, one method of the pair contains the bugged code, and the other method contains the resolution of that bug. During the analysis of the documentations we followed some self-imposed criteria in order to take into consideration the only patterns that could report significant results to our analysis:

- every selected violation must have a resolvable counterpart, and the execution of code that contains the bug has to produce the same result as the execution that has the specified solution.
- violations that consist in either compile time or run time errors must be rejected. Faulty code is considered untestable, nevertheless it must be noted that code that causes frequent system crashes can quickly increase its carbon footprint. A better focus on the structural quality of software can improve applications efficiency as well as reduce all those defects that cause interruptions;
- rules specifying bugs that, even if not generating any error, do not have any solution must not be taken into account.
- "coding style"rules, i.e. all those best practices or naming convention that does not interfere with the effective computation, are rejected. This set of bugs, when does not interfere with syntactic rules, is intuitively logically not intended to affect energy management in any way;
- rules without a fixed solution must not be taken into account. Solutions that are too general, or vary according to specific situations, are not good for a valid test case;
- the selection of the rules also follows some sort of intuition. Rules that intuitively do not interfere with energy efficiency at all are discarded from the selection.

After an accurate analysis of the several documentations, we identified 53 patterns to be tested, and we divided them into different categories for grouping all those violations belonging to the same semantical area. The sections are: *Expensive Calls in Loops*, *DB calls*, *Expensive Object Instantiation*, *Garbage Collector calls* and *Poor Programming Choices*, as reported in Table I. These divisions help to better evaluate the results of the testing phase, and to confront the energy spending differences between similar patterns.

B. CONSIDERATIONS ON THE SELECTED PATTERNS

A considerable number of patterns concerns inefficient Object instantiations. One of the fundamental OO performance management principle is to avoid excessive object creation. This suggests being wary of object creation risks when executing performance-critical code –like for example inside of loops–, since this operation involves memory allocation for all the Object fields, all its superclasses fields and all its subclasses fields. From the amount of operations involved, it is evident that Object creation is a procedure expensive enough to reconsider its usage in temporary or intermediate objects. If it is not necessarily required, it should always be considered to use an equivalent static.

In relation to the instantiation problem there is also the issue of String concatenation: during each concatenation between two Strings `a` and `b`, a new StringBuilder Object `c` is created. Firstly empty, it is then appended with the values of `a` and `b`, and eventually re-casted to a String for the return. This –only apparently– trivial operation actually consists in a series of non-negligible ones, and if used serially it results in the creation of a large number of temporary Objects.

Additional attention should be put when sensible operations are executed inside loops. Operation that may seems "harmless" became performance-affective when executed in large number inside `for` or `while` loops.

[3]http://doc.castsoftware.com/display/DOC82/Metrics+and+Quality+Rules+-+details

[4]http://findbugs.sourceforge.net

[5]https://pmd.github.io/pmd-6.0.0/index.html

TABLE I
IDENTIFIED PATTERNS

section	method identifier
Expensive Calls in Loops	callPropertiesThatCloneValuesInLoop; indirectExceptionHandlingInsideLoops; indirectStringConcatenationInLoops; instantiationInsideLoop; stringConcatenationInLoop; terminationExpressionInLoop;
DB calls	closeDBResourcesASAP; usingLONGDatatype; nullableColumn; selectAll; implicitConversionInWHERE; preferUNIONALL; usingOldStyleJoin; mixAnsiWithOracleSyntax; existsIndipendentClause; withoutPrimaryKey; useDedicatedStoredProcedure; sqlQueriesInsideLoop; tooManyIndexes; firstIndexInWhere; redundantIndexes;
Expensive Object Instantiation	dynamicInstantiation; instantiatingBoolean; largeNumberOfStringConcat; stringInitializationWithObject; declareStaticMethod; boxingImmediatelyUnboxed; numberCtor; randomUsedOnlyOnce; needlessInstantiation; boxedPrimitiveToString; newObjectForGetClass;
Garbage Collector calls	callSuperFinalizeInFinally; callingFinalize;
Poor Programming Choices	arrayCopy; collapsibleIfStatement; deadLocalStoreReturn; formatStringNewLine; localDoubleAssignment; mutualExclusionOR; nonShortCircuit; repeatedConditionals; selfAssignment; staticFieldCollection; switchRedundantAssignment; useArraysAsList; uselessControlFlow; uselessSubstring; usingHashtable; usingRemoveAllToClearCollection; usingStringEmptyForStringTest; usingVector; variableDeclaredAsObject;

Some patterns are related to the choice in using certain methods rather than similar others, according to specific situation. Using the method `c.clear` rather than preferring `c.removeAll(c)` when clearing a Collection `c` has no difference at all in the final results. But for how the methods are implemented, their performance is different. Equivalently, some similar methods are tested for telling which solution is better for an energy saving approach.

Garbage Collector is a controversial subject in Java. Directories say that the garbage collector should not be explicitly used in the code, since it is an automated process scheduled by the Java Runtime Environment. Its execution is non-deterministic, so there is no way to predict when garbage collection will occur at run time. It is although possible to include hints in the code to run the garbage collector with the System.gc() or Runtime.gc() methods, but they provide no guarantee that the Garbage Collector will actually run. The purpose of the patterns here defined is then to determine whether these calls have some effect on the actual invocation of the deletion process and thus how much they affect the energy spending.

Pattern that tests database connection handling issue with resource leak or data manipulation. Both situation are factors that can lead to inefficiencies related to also the Java side of the connection. On the other side, for example, executing all the energy-intensive operation on the database side of the communication rather than the Java side can lead to several benefits. Firstly, SQL and the other query languages have better capability in managing data and structures than Java. Furthermore, data results optimizations on that side leads to transfer smaller quantity of data. All these factor have a positive impact on the total energy consumption. For better results, each database-related pattern has been implemented for three different management systems: MySQL 14.14, PostgreSQL 9.5.12 and Oracle 11g Express Edition 11.2.0.2.0. In this way results can be more generalized and free of any platform-specific influence.

C. EXPERIMENTAL SETUP

For each identified pattern, we defined a pair of Java methods: one that implements the violation, the other that implements the corresponding resolution. In order to obtain a reliable amount of data to analyze, we repeated each pattern 1 million times, measuring the total energy spent for this operation, and repeating this procedure 50 times collecting 50 different samples. Measurements were executed using jRAPL library [22], on a laptop with the following technical specifications:

- HP Pavilion x360 - 13-a105nl with 4×Intel Core i3-4030U @1.90GHz, 8GB DDR3 RAM, 500GB SSHD with Ubuntu 16.04 LTS 64-bit

A pair of hypothesis, a null one and an alternative one, is formulated for each pattern under test. The comparison between the values of each patterns that contains the inefficiency with the one that has the resolution helps the identification of those which are green smells. The hypothesis are the following:

$H1_0$: E_{with} and $E_{without}$ are similar
$H1_a$: E_{with} and $E_{without}$ are independent

where E is the energy consumption of the pattern under test, with or without the potential smell. The null hypothesis is rejected in favour of the alternative one only if one of the pattern consumes less or more energy than its relative counterpart. The hypothesis is tested with a given $\alpha = 0.05$. If this condition is met, the impact that the resolution of the bug has on energy consumption decides whether a green smell is found or not. As a precaution against possible noise during energy measurements, we imposed a threshold in order to considers relevant only the results with an impact greater than 5% on the overall value.

We defined a different testing environment for the set of patterns belonging to the *DB calls* section. We implemented and tested a pair of function for each of these patterns for three different management systems: MySQL 14.14, PostgreSQL 9.5.12 and Oracle 11g Express Edition 11.2.0.2.0. For this reason, we established an additional condition for their validation: we can consider them green smells only if an energy improvement is experienced on all the tested systems. This condition is necessary to minimize platform-specific optimization and to make the green smells as general as possible.

D. EXPERIMENTAL RESULTS

Experiment results are reported in Table II, divided into the already mentioned partitions: Expensive Calls in Loops, Expensive Object Instantiation, Garbage Collector calls and Poor Programming Choices. From the total 38 patterns, 19 are proven to be green smells. The largest number of the smells belongs to two categories of patterns: *Expensive Object Instantiation* and *Expensive Calls in Loops*, while *Garbage Collector calls* has no patterns that are green smells. An additional value named rank divides the green smells into different categories, according to the gravity of the impact they have on energy consumption. If the impact they have is heavier, then the rank is reported higher, otherwise if they have a minimal impact rank value is smaller.

TABLE II
ENERGY RESULTS

method name	impact	rank
callPropertiesThatCloneValuesInLoop	**-114.56**	**9**
indirectExceptionHandlingInsideLoops	**-130.98**	**9**
indirectStringConcatenationInLoops	**-56.77**	**9**
instantiationInsideLoop	**-22.60**	**4**
stringConcatenationInLoop	**-11.69**	**7**
terminationExpressionInLoop	-1.32	
boxedPrimitiveToString	**-35.00**	**7**
boxingImmediatelyUnboxed	**-20.86**	**4**
declareStaticMethod	**-12.35**	**4**
dynamicInstantiation	**-107.52**	**9**
instantiatingBoolean	**-18.72**	**4**
largeNumberOfStringConcat	43.41	
needlessInstantiation	**-5.44**	**2**
newObjectForGetClass	**-21.34**	**2**
numberCtor	**-12.73**	**4**
randomUsedOnlyOnce	**-119.69**	**9**
stringInitializationWithObject	**-24.88**	**4**
callSuperFinalizeInFinally	0.37	
callingFinalize	0.53	
arrayCopy	-3.12	
collapsibleIfStatement	-4.66	
deadLocalStoreReturn	-2.69	
formatStringNewLine	0.74	
localDoubleAssignment	3.30	
mutualExclusionOR	2.35	
nonShortCircuit	**-26.96**	**4**
repeatedConditionals	0.89	
selfAssignment	-3.06	
staticFieldCollection	33.58	
switchRedundantAssignment	0.80	
useArraysAsList	**-155.73**	**9**
uselessControlFlow	-2.36	
uselessSubstring	-4.74	
usingHashtable	**-56.08**	**9**
usingRemoveAllToClearCollection	**-36.45**	**7**
usingStringEmptyForStringTest	-1.86	
usingVector	2.10	
variableDeclaredAsObject	-0.51	

Instantiation of objects that are no strictly necessary is proved to bring a large waste in energy resources. Object allocation involves some collateral effects that have a non-negligible impact on the computation, such as the assignment of the required memory, the invocation of Class constructors and so on. It is straightforward that avoiding an instantiation prevents all these expensive instructions to be executed. All the patterns of *Expensive Calls in Loops* report an improvement when the violation is resolved, this reflect the substantial impact that loops have on inefficient code. Every pattern in this category is then included in the green smells set, except from *terminationExpressionInLoop*: even if it resolution brings a benefit in energy consumption, its value does not meet the necessary requirements, failing to exceed the 5% threshold.

For what concern *DB calls* section, unfortunately the tests relative to the Oracle management system presented several problems during the execution. Due to the high number of repetition required for generating a sample, the executions were occasionally slowed down, affecting the results of energy samplings. These results are then discarded from final evaluation. Only the values from MySQL and PostreSQL executions are considered. Table III reports the outcomes of energy measurement, marking the green smells in bold. The only patterns that are energetically inefficient in both the configurations are *existsIndipendentClause* and *sqlQueriesInsideLoop*. This result reflects the fact that for these two queries, a larger computational effort from Java is required. The computational load of the remaining set of patterns is evidently left to the database.

TABLE III
DB CALLS - ENERGY RESULTS

method name	impact	rank
closeDBResourcesASAP_MYSQL	-1.81	
existsIndipendentClause_MYSQL	**-54.16**	**9**
firstIndexInWhere_MYSQL	-0.40	
implicitConversionInWHERE_MYSQL	-0.51	
mixAnsiWithOracleSyntax_MYSQL	-0.49	
nullableColumn_MYSQL	0.20	
preferUNIONALL_MYSQL	-0.63	
redundantIndexes_MYSQL	0.18	
selectAll_MYSQL	182.00	
sqlQueriesInsideLoop_MYSQL	**-124.30**	**9**
tooManyIndexes_MYSQL	-3.48	
useDedicatedStoredProcedure_MYSQL	1.21	
usingLONGDatatype_MYSQL	-0.19	
usingOldStyleJoin_MYSQL	0.08	
withoutPrimaryKey_MYSQL	182.03	
closeDBResourcesASAP_POSTGRE	-1.12	
existsIndipendentClause_POSTGRE	**-77.30**	**9**
firstIndexInWhere_POSTGRE	-0.49	
implicitConversionInWHERE_POSTGRE	3.28	
mixAnsiWithOracleSyntax_POSTGRE	-0.09	
nullableColumn_POSTGRE	-1.20	
preferUNIONALL_POSTGRE	-1.84	
redundantIndexes_POSTGRE	**-181.82**	
selectAll_POSTGRE	0.36	
sqlQueriesInsideLoop_POSTGRE	**-91.96**	**9**
tooManyIndexes_POSTGRE	-1.34	
useDedicatedStoredProcedure_POSTGRE	**-19.45**	
usingLONGDatatype_POSTGRE	23.45	
usingOldStyleJoin_POSTGRE	-0.55	
withoutPrimaryKey_POSTGRE	0.52	

E. CORRALATION BETWEEN ENERGY, TIME AND RESOURCE CONSUMPTION

The second research question aims to find a correlation between the energy consumption and other variables: execution time and memory usage of the tested patterns. A description of the tests and an analysis of the obtained results gives an answer to the research question.

Using the same setup of the previous tests, these two new analyses aim to detect improvements in performance time and in resource usage.

For what concert execution time, the pair of null and alternative hypothesis that are formulated for this test are:

$H2_0$: T_{with} and $T_{without}$ are similar
$H2_a$: T_{with} and $T_{without}$ are independent

where T is the execution time of the pattern under test, with or without the violation. Final decision is still taken according to the values of Mann-Whitney test using α = 0.05 and to the impact on the overall time, in the same way as the previous test required. Patterns that satisfy the conditions are then named *performance smells*.

The same approach is also used for the analysis of memory usage. A new couple of null and alternative hypothesis is now considered:

$H3_0$: M_{with} and $M_{without}$ are similar
$H3_a$: M_{with} and $M_{without}$ are independent

with M being the bytes of resources used by each pattern, with or without the violation. Memory allocation is calculated getting the free space left in the Java Virtual Machine. Results of this test are still evaluated according to the values of Mann-Whitney test and to the impact of total used bytes. The resulting patterns that pass all the thresholds are named *memory smells*.

Correlations between performance and memory results on the green smells are reported in Tables IV and V, each row has a mark that indicates whether the pattern is a performance or a memory smell. Even if those three sets of smells have a lot of patterns in common, it is evident that they do not coincide. More specifically, for what concern execution time correlation, *needlessInstantiation* does not presents performance improvements although being a green smell, and *uselessSubstring* on the contrary experience a faster execution that does not bring any energy saving. Similarly, memory consumption analysis shows that resources has correlations with the other tests: *largeNumberOfStringConcat*, *formatStringNewLine* and *staticFieldCollection* are reported to be memory smells but they actually do not affect neither energy consumption nor execution time.

TABLE IV
PERFORMANCE AND MEMORY RESULTS

method name	perf. smell	mem. smell
callPropertiesThatCloneValuesInLoop	×	
indirectExceptionHandlingInsideLoops	×	×
indirectStringConcatenationInLoops	×	×
instantiationInsideLoop	×	×
stringConcatenationInLoop	×	×
terminationExpressionInLoop		
boxedPrimitiveToString	×	×
boxingImmediatelyUnboxed	×	×
declareStaticMethod	×	×
dynamicInstantiation	×	
instantiatingBoolean	×	×
largeNumberOfStringConcat		×
needlessInstantiation		
newObjectForGetClass	×	×
numberCtor	×	×
randomUsedOnlyOnce	×	
stringInitializationWithObject	×	×
callSuperFinalizeInFinally		
callingFinalize		
arrayCopy		
collapsibleIfStatement		
deadLocalStoreReturn		
formatStringNewLine		×
localDoubleAssignment		
mutualExclusionOR		
nonShortCircuit	×	
repeatedConditionals		
selfAssignment		
staticFieldCollection		×
switchRedundantAssignment		
useArraysAsList	×	×
uselessControlFlow		
uselessSubstring	×	
usingHashtable	×	
usingRemoveAllToClearCollection	×	×
usingStringEmptyForStringTest		
usingVector		
variableDeclaredAsObject		

F. BYTECODE ANALYSIS

We lastly performed an inspection on Java bytecode operations of the only patterns that are green smells, in order to detect those instruction that may be responsible of excessive energy consumption. We gave particular attention to all those operations that handle methods invocation (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`), the number of interactions with the stack (`push`, `load`, `pop` of values), and `new`, since these are the most common operations in the code.

`invokeinterface` is used to invoke a method declared within a Java interface; `invokespecial` can be used to invoke a private method of `this` Class, a method in a superclass of `this`, or principally to invoke the instance initialization method during the construction phase for a new object. `invokestatic` calls static methods, also known as class methods. Finally, `invokevirtual` is used in Java to invoke all the methods that do not fall into the previous cases. `new` is used when creating object instances, it determines the size in bytes of the given class and allocates memory for the new instance from the garbage collected heap. A reference to the new object is also pushed onto the operand stack.

After the comparison of bytecode between both the element of every couple, it appears that the number of interactions with the stack does not affect the energy consumption, as reported in [23] stack operations consist in a constant small number of cycles, so they do not have a big impact. On the contrary, the main responsible in the computation are invoke-operations. Their execution requests a high number of cycles: `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual` account respectively for 114+6*r*+*l*, 74+3**r*+*l*, 74+3**r*+*l* and 100+4*r*+*l* number of cycles, with *r* representing the constant for memory read and *l* the time required for loading into the cache. These specific instructions are among the ones that require most cycles to be performed, so it is clear that their execution requires a bigger compu-

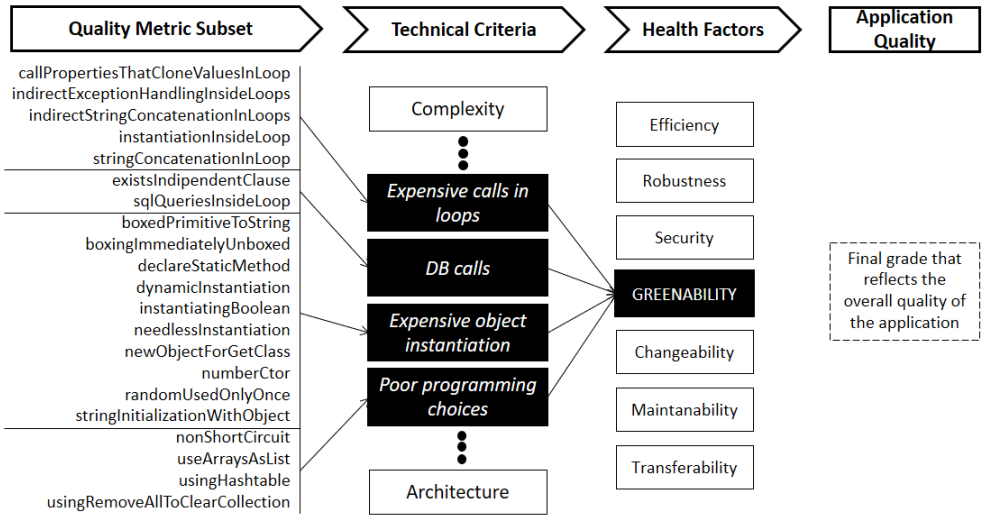


Fig. 1. CAST Assessment Model structure with the definition of the *Greenablity* index

TABLE V
DB CALLS - PERFORMANCE AND MEMORY RESULTS

method name	perf. smell	mem. smell
closeDBResourcesASAP_MYSQL		
existsIndipendentClause_MYSQL	×	
firstIndexInWhere_MYSQL		
implicitConversionInWHERE_MYSQL		
mixAnsiWithOracleSyntax_MYSQL		
nullableColumn_MYSQL		
preferUNIONALL_MYSQL		
redundantIndexes_MYSQL		
selectAll_MYSQL		
sqlQueriesInsideLoop_MYSQL	×	×
tooManyIndexes_MYSQL		
useDedicatedStoredProcedure_MYSQL		
usingLONGDatatype_MYSQL		
usingOldStyleJoin_MYSQL		
withoutPrimaryKey_MYSQL		
closeDBResourcesASAP_POSTGRE		
existsIndipendentClause_POSTGRE	×	
firstIndexInWhere_POSTGRE		
implicitConversionInWHERE_POSTGRE		
mixAnsiWithOracleSyntax_POSTGRE		
nullableColumn_POSTGRE		
preferUNIONALL_POSTGRE		
redundantIndexes_POSTGRE		
selectAll_POSTGRE		
sqlQueriesInsideLoop_POSTGRE	×	×
tooManyIndexes_POSTGRE		
useDedicatedStoredProcedure_POSTGRE	×	
usingLONGDatatype_POSTGRE		×
usingOldStyleJoin_POSTGRE		
withoutPrimaryKey_POSTGRE		

tational effort by the processor, leading to a higher energy consumption. On the other hand, `new` execution time depends on the size of the created object: the amount of memory required for the allocation of the object is firstly filled with zeros, then instantiated.

When new instances are created, `invokespecial` and `new` are always executed: the first one is needed for allocating required memory for the new Object, the second for invoking the Class constructor. It is clear now why a big part of green smells is related to useless Object instantiations: for the operations that it involves, it leads to a great energy spending.

The analysis of bytecode also supports the motivation behind the energy consumption increment caused by String concatenations. Due to the series of collateral operation that the + operator involves, for a single concatenation are executed a `new` for allocating memory for the StringBuilder object, an `invokespecial` for its constructor, and 3 `invokevirtual` for appending the intermediate values and for the conversion of the object to a String. Using the `append` method on the other hand only involves a single `invokevirtual`.

V. GREENABILITY INDEX

We lastly defined a new index to monitor and check patterns energy inefficiency in the source code in an automated way. The new index is named *Greenability*.

A. METRIC DEFINITION

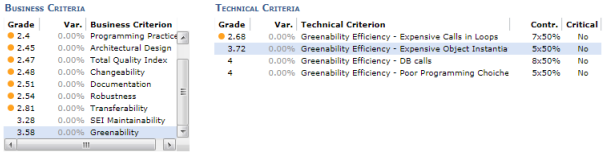
CAST Assessment Model defines a set of rules that allows the generation of a Portfolio of Technical and Quality Assessment for given applications. The model is structured as reported in Figure 1: a list of Quality Rules belongs to one or more Technical Criteria, whose in turn belongs to one or more Business Criteria. Business Criteria can be either Health Factors (e.g. Changeability, Efficiency, Robustness, Security, Maintainability or Transferability) or Rule Compliance set (e.g. Architectural Design, Documentation and Programming Practices). When executed, the analysis assigns a final *grade* value to each Quality Rule, Technical Criteria and Health Factor, using an algorithm that takes into account both the number of matches of the quality rules inside the application under analysis and their weights, a value that reflects the gravity that the violation represents. The final outcome is the Application Quality Index.

In order to define the new metric, we extended the Assessment Model adding a new Health Factor named *Greenability*, able to provide a quality assessment on the energy efficiency of given applications. The quality rules that compose this Business Criterion are all the green smells that we detected during the empirical analysis described in previous chapters, that is all the pattern marked in bold and with a *rank* value in Tables II-III. Basic requirements for the quality rules are the *weight* value, and a regular expression that is able to detect the violation inside application code. Since some of these rules were already present in CAST Quality Rules set, we changed their *weight* using the *ranks* values we identified during the testing phase. We also implemented the regular expressions for the detection for all the remaining rules instead.

B. A PRACTICAL APPLICATION

We finally performed a practical evaluation of the index through the execution of a sample analysis on WebGoat[6], a deliberately flawed application that allows interested developers to test vulnerabilities commonly found in Java-based applications.

After we performed the analysis, the CAST Engineering Dashboard shows for each Business and Technical Criterion, every occurrence of the violated Rules. Every Business and Technical Criterion as well as every rule has a grade (normalized compliance ratio) from 1 to 4, reflecting the gravity of the metric under analysis present in current analyzed source code. In Figure 2, WebGoat analysis results are divided into different Business Criteria; Greenability is highlighted alongside its corresponding Technical Criteria: Expensive Calls in Loops, Expensive Object Instantiations, DB Calls and Poor Programming Choices. The overall value of WebGoat Greenability is 3.58, indicating an acceptable level of energy efficiency, while the less efficient Technical Criteria is *Expensive Calls in Loops*, with a grade of 2.68. Figure 3 illustrates all the Quality Rules taken into account for the computation of the Expensive Object Instantiation criteria, also reporting their grade. For every detected rule, the Dashboard also points the portion of code that triggers the violation, highlighted in red (Figure 4).



BUSINESS CRITERIA

Grade	Var.	Business Criterion
2.4	0.00%	Programming Practice
2.45	0.00%	Architectural Design
2.47	0.00%	Total Quality Index
2.48	0.00%	Changeability
2.51	0.00%	Documentation
2.54	0.00%	Robustness
2.81	0.00%	Transferability
3.28	0.00%	SEI Maintainability
3.58	0.00%	Greenability

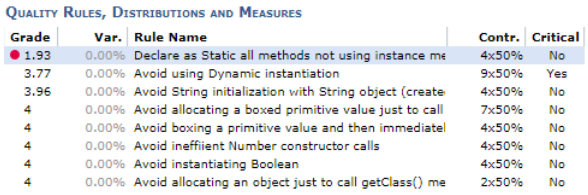
TECHNICAL CRITERIA

Grade	Var.	Technical Criterion	Contr.	Critical
2.68	0.00%	Greenability Efficiency - Expensive Calls in Loops	7x50%	No
3.72	0.00%	Greenability Efficiency - Expensive Object Instantia	5x50%	No
4	0.00%	Greenability Efficiency - DB calls	8x50%	No
4	0.00%	Greenability Efficiency - Poor Programming Choiche	5x50%	No

Fig. 2. Business Criteria that are taken into account for WebGoat analysis, with Greenability metric and its Technical Criteria.

The implementation of this index offers a useful methodology for assessing sample Java applications in order to evaluate their energy efficiency, and gives the ability to analyze and refactor the code from an energy efficient point of view. Greenability index allow the monitor and check the energy

[6]https://github.com/WebGoat/WebGoat/wiki



QUALITY RULES, DISTRIBUTIONS AND MEASURES

Grade	Var.	Rule Name	Contr.	Critical
1.93	0.00%	Declare as Static all methods not using instance me	4x50%	No
3.77	0.00%	Avoid using Dynamic instantiation	9x50%	Yes
3.96	0.00%	Avoid String initialization with String object (create	4x50%	No
4	0.00%	Avoid allocating a boxed primitive value just to call	7x50%	No
4	0.00%	Avoid boxing a primitive value and then immediatel	4x50%	No
4	0.00%	Avoid ineffiient Number constructor calls	4x50%	No
4	0.00%	Avoid instantiating Boolean	4x50%	No
4	0.00%	Avoid allocating an object just to call getClass() me	2x50%	No

Fig. 3. Quality Rules belonging to Expensive Object Instantiations Criterion.

```
        if (lesson != null)
        {
            source = lesson.getSource(s);
        }
    }
    if (source == null)
    {
        return "Source code is not available. Contact webgoat@g2-inc.com";
    }
    return (source.replaceAll("(?s)" + START_SOURCE_SKIP + ".*"
            + END_SOURCE_SKIP, "Code Section Deliberately Omitted"));
}


/**
 *  Description of the Method
 *
 * @param  s                Description of the Parameter
 * @param  response         Description of the Parameter
 * @exception  IOException  Description of the Exception
 */
protected void writeSource(String s, HttpServletResponse response)
        throws IOException
{
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    if (s == null)
    {
        s = new String();
    }

    out.print(s);
    out.close();
}
```

Fig. 4. A sample occurrence of *Avoid String Initialization with String object* violation inside the code.

efficiency of given applications in a structural, automated and consistent way over the time. The vulnerabilities catch by the Greenability index could help you to reduce the carbon footprint of given applications.

VI. CONCLUSIONS AND FURTHER DEVELOPMENTS

Current literature does not present studies on the assessment on the impact that software can have on the energy consumption of general-purpose applications, and still lacks of a metric that is able to quantitatively determine the energy efficiency of given software applications. Driven by these motivations, our work investigated which well-known Java violations also represent energy hotspots. A deep analysis of the tests results gave us better understanding the energy management in software application, and then helped us in the definition of the *Greenability* metric.

Our work followed two main research questions:

RQ1: which of the patterns under test represent an energy hotspot (i.e. which are green smells)?

– After the tests, a final list of 21 patterns were proved to be energy inefficient, and they were therefore defined as *green smells*. Every smell is assigned to a category of similar smells, and a *rank* value is used for describing the impact that it has on the overall consumption. Most common inefficiencies are related to non-optimal Object instantiations and to expensive operation executed inside loops.

RQ2: which are the main variables between execution time, resource usage and bytecode instruction, that are responsible of the increase in energy consume?

– Research Question 2 was intended to find those variables that may be the cause of the energy consumption increment. Tests were rerun firstly considering the execution time as variable under analysis, then resource usage; this permitted to define two new sets of *performance smells* and *resource smells*. Results show no direct effect-cause relationship between energy and any of the measurements, since neither performance smells nor resource smells sets coincide with the green smells group. A last possible answer to RQ2 was explored in Java bytecode operations, in order to detect those instructions that may influence energy consumption. This analysis highlighted a higher distribution of principally two instructions: `new` and *invoke*-methods operations were present in higher number inside violation implementations rather than in their healed versions. This result is motivated by the fact that this pair of instructions is principally required during Objects instantiation: `new` for allocating memory for the new instance and `invokespecial` for calling the Class constructor. This indicates auxiliary Objects created at runtime as one of the main causes for energy waste.

Using current results, future works can focus on expanding the research for new energy hotspots, focusing on inefficient Object instantiations.

An interesting approach would be the comparison of energy consumption according to the presence of Fowler [16] code smells. More precisely, confrontations of particular smells like *Large classes* or *Long methods*, in contrast with others like *Feature envy* or *Lazy classes* can lead to significant results. First ones indeed can indicate a reduction in Object instantiations and methods invocations, leading to energy saving. On the contrary the second group of smells reflects a bad use of class resources, this may increase the number of *invoke*-methods in order to access to external features thus increasing power spending.

It should also important to better investigate the impact that specific bytecode operations have on the energy spending. A removal of the `invokespecial` instruction is for example possible through the decapsulation of Class variables. Even if against Object Oriented Programming practices, an analysis on the effects that this specific instruction has on the consumption could be significant.

This example also introduces another subject: improving energy efficiency of software applications through, for example, the detection and correction of green smells, can raise possible trade-off choices. Since energy smells not always are also performance smells, in some situations a greener solution can severely impact the execution time of the applications, and vice versa. Further studies should then quantify the benefits that a solution can give in opposition to the other, finding, if it possible, which are the best trade-off solutions.

REFERENCES

[1] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inłs Azevedo, and William Lintner. United states data center energy usage report. 2016.

[2] Snehanshu Saha, Jyotirmoy Sarkar, Avantika Dwivedi, Nandita Dwivedi, Anand M. Narasimhamurthy, and Ranjan Roy. A novel revenue optimization model to address the operation and maintenance cost of a data center. Journal of Cloud Computing, 5(1), Jan 2016.

[3] John A Laitner and Mike Berners-Lee. Gesi smarter 2020: The role of ict in driving a sustainable future. Technical report, Technical report, Global e-Sustainability Initiative, 2012.(Cited on page 2.), 2012.

[4] San Murugesan. Harnessing green it: Principles and practices. IT professional, 10(1), 2008.

[5] Antonio Vetro, Luca Ardito, Maurizio Morisio, and Giuseppe Procaccianti. Monitoring it power consumption in a research center: Seven facts. 2011.

[6] Giuseppe Procaccianti, Luca Ardito, Maurizio Morisio, et al. Profiling power consumption on desktop computer systems. In International Conference on Information and Communication on Technology, pages 110123. Springer, 2011.

[7] Giuseppe Procaccianti, Hector Fernandez, and Patricia Lago. Empirical evaluation of two best practices for energy-efficient software development. Journal of Systems and Software, 117:185198, 2016.

[8] Marco Bessi. A methodology to improve the energy efficiency of software. 2014.

[9] Giovanni Agosta, Marco Bessi, Eugenio Capra, and Chiara Francalanci. Automatic memoization for energy efficiency in financial applications. Sustainable Computing: Informatics and Systems, pages 105-115, 2012.

[10] Marco Bessi, Eugenio Capra, and Chiara Francalanci. A benchmarking methodology to assess the energy performance of MIS applications. International Conference on Information Systems (ICIS), 2013.

[11] Gustavo Pinto and Fernando Castor. On the implications of language constructs for concurrent execution in the energy efficiency of multicore applications. In Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity, pages 9596. ACM, 2013.

[12] David HK Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In Cyber-Physical Systems, Networks, and Applications (CPSNA), 2015 IEEE 3rd International Conference on, pages 7885. IEEE, 2015.

[13] Gustavo Pinto, Fernando Castor. Characterizing the energy efficiency of javas thread-safe collections in a multicore environment. In Proceedings of the SPLASH2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS, volume 14, 2014.

[14] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In ACM SIGPLAN Notices, volume 49, pages 345360. ACM, 2014.

[15] Christian Bunse, Hagen Hpfner, Essam Mansour, and Suman Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In Mobile Data Management: Systems, Services and Middleware, 2009. MDM09. Tenth International Conference on, pages 600607. IEEE, 2009.

[16] Martin Fowler and Kent Beck. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[17] Jan Jelschen, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter. Towards applying reengineering services to energy-efficient applications. In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, pages 353358. IEEE, 2012.

[18] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. GI-Jahrestagung, 208:441455, 2012.
[19] Marion Gottschalk, Jan Jelschen, and Andreas Winter. Saving energy on mobile devices by refactoring. In EnviroInfo, pages 437444, 2014.
[20] Claas Wilke, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Amann. Energy consumption and efficiency in mobile applications: A user feedback study. In Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International conference on and IEEE Cyber, Physical and Social Computing, pages 134141. IEEE, 2013.
[21] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, and Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. 2013.
[22] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In FASE, pages 316331, 2015.
[23] Martin Schoeberl. Jop: A java optimized processor for embedded real-time systems. VDM Publishing, 2008.