An effort allocation method to optimal code sanitization for quality-aware energy efficiency improvement

Marco Bessi*, Gabriella Carrozza†, Roberto Pietrantuono‡, Stefano Russo‡,
*CAST Software, Via San Vittore n. 49, 20123 Milan, Italy
†Accenture Operations - Infrastructure Services Manager, Piazzale dell'Industria n. 40, 00144, Rome, Italy
‡Università degli Studi di Napoli Federico II, DIETI, Via Claudio 21, 80125, Naples, Italy.
Email: M.Bessi@castsoftware.com, gabriella.carrozza@accenture.com, {roberto.pietrantuono, stefano.russo}@unina.it

Abstract—**Software energy efficiency has been shown to remarkably affect the energy consumption of IT platforms. Besides "performance" of the code in efficiently accomplishing a task, its "correctness" matters too. Software containing defects is likely to fail and the computational cost to complete an operation becomes much higher if the user encounters a failure. Both performance-related energy efficiency of software and its defectiveness are impacted by the *quality* of the code. Exploiting the relation between code quality and energy/defectiveness attributes is the main idea behind this position paper. Starting from the authors' previous experience in this field, we define a method to first predict the applications of a software system more likely to impact energy consumption and with higher residual defectiveness, and then to exploit the prediction for optimally scheduling the effort for code sanitization – thus supporting, by quantitative figures, the quality assurance teams' decision-makers.**

Index Terms—**Reliability allocation, effort allocation, test planning, quality assurance, static analysis, prediction, optimization**

I. Introduction

Energy consumption of IT systems is increasingly becoming a concern, especially for large infrastructure managers. Research on Green IT is developing at a rapid pace, spanning from fields centered on hardware device and architectural design, to software design and development, as well as to operations management and usage practices.

Software energy efficiency has been shown to remarkably affect energy consumption of all the other infrastructural levels [1] [2]. The efficiency of the code is indeed related to the computational expense needed to perform the intended task. We claim that the overall energy efficiency due to software is not only related to *how fast* the code performs a task, but also to the correctness of its execution. A software containing defects is likely to fail upon a request is made; the computational cost to accomplish a requested operation becomes much higher if the user encounters a failure and must retry the operation, or, even worse, if the system has to be restarted or switched off in favour of a replica.

Both performance-related efficiency of software and its defectiveness are related to the *quality* of the code. Quality is a multi-facet attribute, including reliability, usability, portability, security, performance, efficiency [3], which are all affected by how the software is designed and ultimately implemented. In this position paper, we first explore the relationship between code quality and performance-related energy consumption as well as between code quality and software defectiveness. Code quality is expressed in terms of degree of adherence to predefined sets of well-programming rules. The violation of rules is checked by means of a well-known tool, the Automatic Static Analysis (ASA) [4]. Some preliminary results are presented in this regard. Then, a method is proposed to:

- *Predict* the applications of a software system more likely to impact performance-related energy consumption as well as the applications more likely to contain defects. This is accomplished by machine learning algorithms to train predictive models inferring the relation between ASA rule violations and energy/defectiveness attributes. By such a knowledge, quality assurance team leaders can rank the applications according to the criticality (both in terms of performance-related energy consumption and defectiveness) before the testing stage starts, and plan actions to improve the code where needed.
- *Optimize* the allocation of the effort for code improvement to each application, based on the prediction results. Effort allocation models are a very useful practice for quality assurance planning on quantitative bases, especially in large systems [5] [6]. We formulate a multi-objective optimization model suggesting the best distribution of effort *i)* across applications and *ii)* across ASA rules, to sanitize the code under specific objectives in terms of: cost to fix violations, expected impact on energy, expected defectiveness reduction.

In the following, we first present our experience on both energy efficiency measurement and software defects analysis (Section II), with preliminary results about correlation with ASA rule violations; then, the method is presented in Section III; a roadmap is in Section IV.

†G. Carrozza was with CAST Software when this paper was prepared.

II. Past Experience and Preliminary Results

A. ASA-Energy Relationship

Measurement (and, in our case, prediction) of energy efficiency is as much important as underestimated. Previous studies shown that 86% of ICT departments in UK do not know their CO2 emissions and 80% of companies do not know their electric bill [1]. In the work we have done about this topic [7] [8], we introduced and tested a model to measure software application energy consumption that decouples the usage of the computational resources from their energy consumption. Such a method allows estimating the energy efficiency of software applications independently of the hardware infrastructure. Let us model the usage of resources by monitoring three key entities: the CPU, the storage (e.g., let us assume a DB that is the common storage means on large systems), the network. According to our model, energy consumption is expressed as:

$$\begin{aligned} E_{TOT} &= E_{idle} + E_{CPU} + E_{DB} + E_{NET} \\ &= P_{idle} \cdot t_{ex} + \gamma \cdot \beta_u + D_{DB} \cdot e_{DB} + D_{NET} \cdot e_{NET} \end{aligned} \tag{1}$$

in which:

- P_{idle} is the power absorption of the CPU in the idle state and t_{ex} is the time in which the system is idle;
- γ is the usage of the CPU, and can be written as: $\frac{1}{f}\int_{t_1}^{t_2} U_{\%}(t)dt \cdot \omega(t)$ where: $U_{\%}$ is the percentage usage of the CPU at time t, ω is a function denoting the application-dependent consumption pattern, which may depend on many factors (e.g., number of concurrent users, type of operations, type of application, state of the DB, etc.); f is the clock frequency of the CPU; t_1 and t_2 are the initial and final measurement time;
- β_u is the power absorption function of the CPU per percent usage (it is approximated by a constant assuming the power grows linearly with usage);
- D_{DB} is the amount of data (in number bytes) exchanged with the DB, while e_{DB} is the energy consumed to exchange 1 byte with the DB;
- D_{NET} is the amount of data (in number bytes) exchanged with the network, while e_{NET} is the energy consumed to exchange 1 byte with the network.

In this expression, the factors γ, D_{DB} and D_{NET} are intrinsic characteristics of the software application, while β_u, e_{DB} and e_{NET} depend on the hardware infrastructure. The sole estimation of the former parameters allows computing the energy consumption of the software application. In previous studies [1], it was shown that the consumption of storage is almost independent of usage, as dynamic RAMs are constantly refreshed and most of the energy consumed by disks is used for spinning. In addition, focusing on CPU-intensive applications, the E_{DB} and E_{NET} are negligible. Thus we assume that energy mainly depends on the usage pattern γ. In [7], we have tested the model by using both ammeter clamp directly connected to the server's CPU electric source and specific HP Performance Center's instrumentations.

Resource usage by the application can be estimated by dynamic analysis, namely by means of profiling during testing. However, there would be three problems with this approach: *i)* the result would depend on the goodness of test cases, namely on their ability to well represent the operational usage; *ii)* the result does not provide direct hints about how it is possible to optimize the code for performance improvement; *iii)* the result of such an evaluation could be provided too late, as (operational) testing is done at the end of the lifecycle, whereas developers would like to benefit from the feedback earlier.

We therefore explore the performance indications that can be given by ASA on the source code. In fact, the adherence or violation to predefined programming rules and patterns is able to suggest possible performance issues. This, in turn, is likely to impact energy consumption, as software with higher performance is expected to be more energy-efficient. We are exploring if there is any relation between Automatic Static code Analysis (ASA) rules and the usage pattern factor γ, which is the most relevant contributor to E_{TOT}. As preliminary result we are obtaining, we have observed that some ASA rules, defined by means of the CAST© tool, are highly correlated to the γ metric, as reported in Table I.

TABLE I: Spearman correlation among # violations with γ for each group of rules. Text in bold indicates confidence $> 95\%$

Programming Rules	**Spearman ρ**
Avoid using Driver Manager	**0.93**
Avoid the use of "Instanceof" inside loops	**0.87**
Avoid using Hashing Table	**0.85**
Avoid String initialization with String objects	**0.85**
Avoid String concatenation in loops	**0.75**
Avoid using Dynamic instantiation	**0.72**

Based on this result, we are confident that the infringements in terms of code rule violations as obtained by ASA are potentially correlated with energy efficiency. This is going to be exploited for our method formulation.

B. ASA-Defects Relationship

The quality of software can be well assessed by ASA. In our previous experience [9], we have studied the quality of large-scale mission-critical software systems produced by a big Italian company, leader in the market of defence, homeland security and protection, information security, avionics and aerospace. The quality as perceived by the end user can be, however, different. User perceives system's failures: one could have a very poor (internal) code quality in terms of violations, but a good user experience. In fact, ASA violations are not necessarily defects: they can be either false warnings, when a violation does not cause the software to fail, or can be actually a defect, whose activation would lead the software to a failure. In [10], we therefore also explored the quality of such systems in terms of actual defectiveness – an exercise that allowed us to assess not only the *product* but also the *process* quality.

Since defects are detected at later stages (i.e, during testing or even operation) compared to violations (detected soon after coding), their treatment is much more expensive. Thus, it makes sense to try anticipating the knowledge about the presence of defects as earlier as possible. To this aim, we are exploring any possible relationship between programming rules violation and the presence of software defects. There is indeed a reasonable expectation that ASA violations are correlated to defects (as proven in some past papers [4] [11]), and that some types of violations (namely, some rules) are more correlated than others. In Table II, we report preliminary results about such correlation we are obtaining on a dataset of 29 components of the same mission-critical system that we have studied in [9]. There is a significant correlation (with 95% of confidence) in several cases. Results can however be improved by considering rules at finer grain (i.e., not by group) than what we did so far. As in the previous case, this output is going to be exploited for the method formulation.

TABLE II: Spearman correlation among # violations with defects for each group of rules. Text in bold indicates a confidence $> 95\%$

Programming Rules	**Spearman ρ**
Metrics Rules	**0.85**
Naming Convention Rules	**0.46**
Possible Bugs	**0.45**
Coding Convention Rules	**0.45**
Formatting Rules	**0.45**
Memory and Resource	**0.44**
Comments Rules	**0.41**
MISRA Rules	**0.37**
OOP Rules	0.29
Optimization Rules	0.29
Security	0.27
Threads & Synchronization	0.27
Resources	0.17
Initialization Rules	0.06
Exceptions Rules	0.01

III. Optimal code sanitization

Based on the preliminary results about correlation, we propose the following allocation method. The method includes two steps: *i)* a **prediction** step, in which we run machine learning algorithms to build predictive models able of spotting the most relevant applications and ASA rules, from both energy and defectiveness point of view; *ii)* an **optimization** step, in which we exploit the output of prediction to orient the fixing of ASA rule violations in terms of: which applications and which rules should be prioritized in the correction. The goal is to provide quality team leaders with a quantitative means to figure out how to sanitize the code to pursue well-defined objectives of energy efficiency, quality and cost.

A. Prediction

The goal of prediction is twofold:

- Predict applications more likely containing defects and applications more prone to consume energy before the integration/system/acceptance testing starts;
- Predict which rules more likely suggest the presence of defects and which ones suggest inefficiency bottlenecks (besides the side feedback of identifying the most critical rules that programmers are keen on violating).

The steps for prediction are:

- **Retrieving data from ASA reports** about a set of applications of a tested or an operational system. Tracking of defects found during testing and of energy consumption measurements[1] should be available.
- **Training of prediction models**. We mean to test several different models, by varying both machine learning algorithms (we opt for *ranking* algorithms, such as regression-based techniques [12], since binary classifiers provide no information about the relative weight of each sample), and the variables of interest (e.g., relying on absolute numbers or relative to the code size).
- **Test of the models** by 10-fold cross-validation, repeated $N = 30$ times per each classifier (for statistical validity), comparison and choice of the best model. Criteria for comparison can be several (the most used ones are the response variable value in the top 20% of samples and Fault Percentile Average (FPA) [12]).
- Actual **prediction** on applications for which we know only the ASA violations but anything about energy consumption (and defects) yet. The output is a list of applications rated by their expected energy consumption and a list rated by expected defectiveness. The ranking-based algorithms associate a ranking score with each sample in the list. We call it p_j (and q_j, for energy consumption) and normalize it in [0,1]: $\omega_{d_j} = \frac{p_j - min(p_j)}{max(p_j) - min(p_j)}$ (similarly for q_j, obtaining ω_{e_j}).
- **Rules rating**. We analyze the impact of attribute selection, by an attribute ranking algorithm such as the *Information Gain*, which rates an attribute according to the gain of information obtained by knowing that attribute. The top rules useful for prediction are derived, and each rule i is assigned a score s_i (h_i for energy consumption list) denoting the importance of that rule for prediction. The scores are then normalized in [0,1] to get the relative *weight* of each rule: $\rho_{d_i} = \frac{s_i - min(s_i)}{max(s_i) - min(s_i)}$ and $\rho_{e_i} = \frac{h_i - min(h_i)}{max(h_i) - min(h_i)}$, in the two cases of defectiveness and energy consumption, respectively.

Prediction information (ω_{d_j}, ρ_{d_i} and ω_{e_j}, ρ_{e_i}) is used to parametrize the optimization model. The steps we are going to implement for optimization are in the next Section.

B. Optimization

Optimization of violation fixing is conceived to exploit the above predictions to prioritize applications and rules to fix, according to predefined objectives and constraints. We aim at providing the team leader with sets of alternative solutions for which the expected impact on cost, quality and energy efficiency is known. Thus a **multi-objective** optimization approach is formulated. The main objective of the proposed approach can be stated as follows: *suggest the number and type of violations to remove and the applications from which they should be removed in order to get the best tradeoffs among: i)*

[1]See [7] for a discussion about measurement of energy consumption; for our purpose, a profiling tool can suffice to get realistic patterns of γ.

minimizing the expected residual defectiveness (i.e., maximize quality), ii) minimizing the expected energy consumption (i.e., maximize energy efficiency), iii) minimizing the expected cost, under user-defined constraints (e.g., maximum budget for code sanitization). Suppose $v_{i,j}$ denotes the number of violation of type i in the j-th application. Let us denote with C_{fix_i} the average expected cost to remove violations of type j (that can be obtained from historical data about violation fixes). Considering the indication about correlation weights, ρ_{d_i} and ρ_{e_i}, each violation type is characterized by C_{fix_i}, ρ_{d_i} and ρ_{e_i}. Furthermore, each rule is assigned a "severity" (S_{d_i} and S_{e_i}, in the defect and energy case, respectively) by domain expert, who usually wants to give priorities to the removal of violations of some types, according to their expected impact (we, discretize the severity levels in the [0,1] range, by equally spacing n levels: $S_1 = 1/n$, $S_2 = 2/n$ ...$S_n = 1$)[2]

It would be extremely expensive to eliminate all violations of all types from all applications. The algorithm finds the best tradeoffs optimizing the objectives. Let us denote the decision variables $x_{i,j}$, being the number of violations of the i-th type that the algorithm proposes to eliminate from application j. The three objectives are expressed as follows:

$$\begin{aligned}
&\text{Min!} \quad C = \sum_{j}^{n}\Big[\sum_{i=1}^{m}(x_{i,j} \cdot C_{fix_i})\Big] \\
&\text{Max!} \quad Q = \sum_{j}^{n}\Big[\sum_{i=1}^{m}(x_{i,j} \cdot \rho_{d_i} \cdot S_{d_i} \cdot \omega_{d_j})\Big] \qquad (2)\\
&\text{Max!} \quad E = \sum_{j}^{n}\Big[\sum_{i=1}^{m}(x_{i,j} \cdot \rho_{e_i} \cdot \cdot S_{e_i}\omega_{e_j})\Big]
\end{aligned}$$

subject to $0 \leq x_{i,j} \leq v_{i,j}$ and to the following bounds:

$$\begin{aligned}
&C < C^* && \text{(Maximum Budget)} \\
&Q > Q^* && \text{(Minimum Quality)} \qquad (3)\\
&E > E^* && \text{(Minimum Energy Reduction)}
\end{aligned}$$

Further constraints on violations can be added, such as: $\frac{1}{N}\sum_{j=1}^{n}\sum_{i=1}^{m}(v_{i,j} - x_{i,j}) \leq \bar{v}_{max}$, meaning that the residual number of average violations must be less than (or equal to) a target $\bar{v}_{max}$. The first objective function minimizes the expected cost for violation removal. The second objective function assesses the expected *quality increase* (to maximize) as sum of violations $x_{i,j}$ to remove weighted by: the likelihood that violation of type i is actually correlated to defects, the likelihood that the interested application j is actually defect-prone, and by the severity of the rule assigned by domain expert. Similarly, the third objective assesses the reduction of energy consumption (to maximize) if we remove $x_{i,j}$ violations. The solution provides a matrix with the amount of violations engineers should remove for each type and for each application, as well as the estimate of removal cost, the expected quality in terms of defects and the expected energy consumption if sanitization actions are taken as suggested. Note that this is just an instance of multi-objective models that can be formulated, and that can give raise to numerous variants targeting specific needs. More generally, the core message is that we can setup code sanitization plans supported by quantitative reasoning, by exploiting prediction and considering quality and energy objectives together in the formulation.

[2]For instance, reliability rules might be assigned a higher defectiveness-related severity; similarly, for performance rules in the energy case.

IV. Roadmap

As first step, we are going to validate the prediction and the optimization steps separately. As for the prediction, we mean to extend the set of test applications (across multiple domains) to improve the generality of predictive models, and to extend the set of metrics to improve their accuracy (e.g., with metrics from version control systems [13]). As for optimization, we will first validate the model numerically to get feedback and refine the model (e.g., by comparing metaheuristics on sample problems). Then, the whole method will be experimented on real case studies. As we did in the past for the mentioned cases, we aim at exploiting the strong collaborations with our industrial partners in order to create a long-term testbed where this kind of models can be validated empirically.

References

[1] E. Capra, G. Formenti, F. C., and S. Gallazzi, "The impact of mis software on it energy consumption," in *European Conference on Information Systems (ECIS)*, 2010.

[2] E. Capra, C. Francalanci, and S. Slaughter, "Is software "green"? application development environments and energy efficiency in open source applications," *Information and Software Technology*, vol. 54, no. 1, pp. 60–71, 2012.

[3] "ISO/IEC 25010 - Software Product Quality." http://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2011.

[4] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proc. 27th Int. Conference on Software Engineering*, pp. 580–586, ACM, 2005.

[5] R. Pietrantuono, S. Russo, and K. Trivedi, "Software reliability and testing time allocation: An architecture-based approach," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 323–337, 2010.

[6] G. Carrozza, R. Pietrantuono, and S. Russo, "Dynamic test planning: a study in an industrial context," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 5, pp. 593–607, 2014.

[7] M. Bessi, E. Capra, and C. Francalanci, "A benchmarking methodology to assess the energy performance of mis applications," in *34th International Conference on Information Systems*, 2013.

[8] G. Agosta, M. Bessi, E. Capra, and C. Francalanci, "Automatic memoization for energy efficiency in financial applications," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 2, pp. 105 – 115, 2012. IEEE International Green Computing Conference (IGCC 2011).

[9] G. Carrozza, M. Cinque, U. Giordano, R. Pietrantuono, and S. Russo, "Prioritizing correction of static analysis infringements for cost-effective code sanitization," in *Proceedings of the Second International Workshop on Software Engineering Research and Industrial Practice*, SER&IP '15, (Piscataway, NJ, USA), pp. 25–31, IEEE Press, 2015.

[10] G. Carrozza, R. Pietrantuono, and S. Russo, "Defect analysis in mission-critical software systems: a detailed investigation," *Journal of Software: Evolution and Process*, vol. 27, no. 1, pp. 22–49, 2015.

[11] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *32nd IEEE Software Engineering Workshop*, pp. 169–174, 2008.

[12] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 234–246, 2015.

[13] D. G. Cavezza, R. Pietrantuono, and S. Russo, "Performance of defect prediction in rapidly evolving software," in *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on*, pp. 8–11, 2015.