Technical Debt Management and Energy Consumption Evaluation in Implantable Medical Devices: The SDK4ED approach

Charalampos Marantos[1], Angeliki-Agathi Tsintzira[2], Lazaros Papadopoulos[1], Apostolos Ampatzoglou[2], Alexander Chatzigeorgiou[2], and Dimitrios Soudris[1]

[1] School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
`hmarantos@microlab.ntua.gr`, `lpapadop@microlab.ntua.gr`,
`dsoudris@microlab.ntua.gr`
[2] Department of Applied Informatics,
University of Macedonia, Thessaloniki, Greece
`angeliki.agathi.tsintzira@gmail.com`, `ampatzoglou@uom.edu.gr`,
`achat@uom.edu.gr`

Abstract. The design constraints of Implantable Medical Devices (IMD), such as the low energy consumption, impose significant challenges to application developers. Software tools that improve the quality of the source code by means of technical debt management and provide energy consumption estimations are useful to IMD application developers for addressing such challenges. In this work, we demonstrate the effectiveness of tools that manage the technical debt and provide energy consumption estimations applied to an IMD application for seizure detection.

Keywords: Technical Debt · Energy Consumption · Implantable Medical Devices · Embedded Systems

1 Introduction

The Implantable Medical Devices (IMDs) are objects surgically inserted into the human body for medical purposes [7]. They are used to treat conditions such as cardiac disorder, epilepsy, numerous autoimmune diseases and psychological disorders (among others), thus contributing to the normal quality of patient's lives. Nowadays, IMDs are a common part of modern medical care, support physicians to diagnose and treat diseases and enable the quality of life of patients.

The design requirements of IMDs include the small volume in terms of size and weight, long lifespan, low energy consumption, high biocompatibility and reliability [8]. As the clinical demand for IMDs increases, addressing design and efficiency challenges is even more urgent. From application development perspective, increasing the quality of source code (improving maintainability and reusability) and evaluating the energy efficiency contribute to meeting the aforementioned challenges.

The SDK4ED platform integrates a number of toolboxes for application developers of embedded systems that enable the optimization of various source code qualities, such as the source code maintainability (e.g. the management of technical debt), the security and the energy consumption optimization [3]. Additionally, it identifies trade-offs between the optimization of the maintainability, security and energy qualities at application source code-level and enables decision support. More specifically, the toolboxes are the following: i) Technical Debt Management ii) Security iii) Energy consumption iv) Forecasting and v) Decision support. The toolboxes recommend source-to-source optimizations, while the decision support toolbox provides guidance to developers about the optimizations that should be applied based on user-selected priorities.

In this paper, we leverage specific tools from the SDK4ED platform to efficiently manage the technical debt of a seizure detection IMD application and estimate the energy consumption. Therefore, this work contributes to the evaluation of the SDK4ED tools on a real-word use case from the IMD domain and we reach interesting conclusions.

The rest of the paper is organized as follows: Section 2 provides more details about the SDK4ED tools used in the present work, as well as related work about the technical debt management and energy consumption approaches in the IMD domain. Section 3 describes the implementation of the tools in the IMD application and in Section 4 we draw conclusions.

2 Related Work

Technical debt (TD) in software engineering refers to the additional maintenance costs caused by quality compromises which are often taken for short-term benefits during the software development process. The TD metaphor which was first coined in 1992 by Ward Cunningham [6] has proved highly effective as a means of conveying to nontechnical product stakeholders the need for what we call “refactoring” [9]. The concept of Technical Debt Management (MTD) encompasses all processes that should be undertaken by software development teams to identify, measure, prioritize and repay TD.

Embedded systems form a software-intensive domain where platform-specific run-time constraints such as performance, energy consumption and memory usage, have to be strictly satisfied. However, embedded systems exhibit long lifetime expectancy, often beyond a decade, resulting in intense maintenance activities. To limit the effort spent on maintenance, companies could invest in boosting design-time quality through the management of TD. A case study involving seven embedded software industries revealed that quality attributes such as functionality, reliability, and performance are indeed given higher priority compared to managing TD [3]. However, developers of embedded software clearly acknowledge the need for low TD on components that are expected to have a longer lifetime [3].

[3] The SDK4ED platform: https://sdk4ed.eu/

Energy efficiency challenge for IMDs is usually addressed through approaches such as energy harvesting [10] and the design of ultra-low power hardware devices [1]. For instance, the integration of dedicated hardware blocks that perform the computational expensive operations, as well as frequency and voltage scaling are typical techniques applied at OS/hardware-level to improve energy efficiency [7]. However, source-to-source energy optimization techniques, such as cache utilization improvement, which are widely applicable in embedded systems are also applicable in the IMD domain. The tools used in the context of this work enable such optimizations by providing relevant information about the energy efficiency of the application. More specifically, the SDK4ED tools for energy consumption optimization extend advanced machine-learning techniques described in the literature for estimating energy consumption [11][5] and identifying acceleration opportunities [2].

3 Technical Debt and Energy Consumption Evaluation

3.1 Overview of the application and source code

The use-case application targets modern Implantable Medical Devices (IMDs), which are battery-powered embedded devices with high safety and reliability standards. These devices are designed to operate for long time (up to 10 years) implanted in the human body. To support the treatment capabilities of these devices, they are equipped with wireless transceivers, able to communicate with external reader/programmer or a base station for local and/or remote monitoring of patient health, performing a device test, reading sensors, updating device settings.

The application provides primary implant functionality, e.g. Neuro-stimulation, seizure detection, cardiac pacing etc. More specifically, the application performs the following tasks:

- The sensor (ECoG/EEG) values are received via ADC periodically (using interrupts)
- An FIR filter operation is performed on the input samples. This filter accurately approximates a continuous complex Morlet wavelet
- Based on the filter output a decision whether the seizure is detected or not is made
- Optogenetic or electrical stimulus are applied via GPIO in order to suppress the seizure

3.2 Technical Debt Management

Methodology In this subsection we present the methodology that has been followed to measure the levels of Technical Debt for the software that belongs to the Implantable Medical Devices application domain.

In the context of the SDK4ED platform, Technical Debt management consists of monitoring all key aspects of TD, namely Principal, Interest and Interest

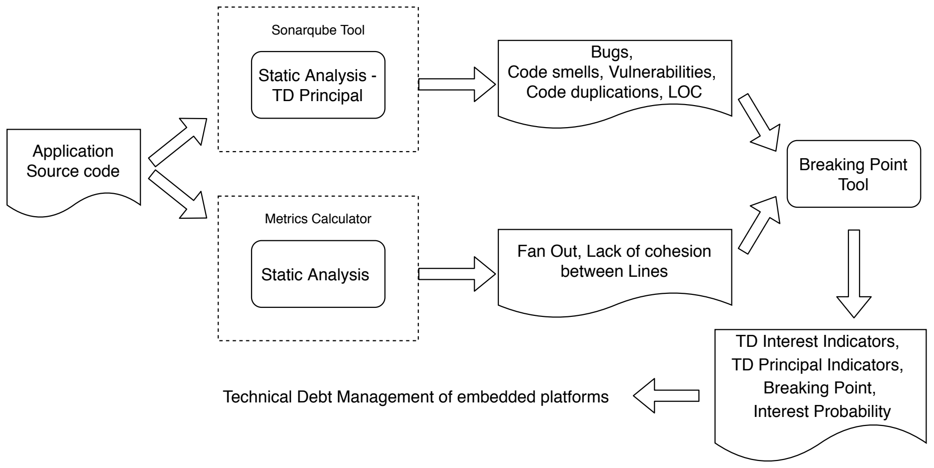

Sonarqube Tool
Static Analysis - TD Principal
Bugs, Code smells, Vulnerabilities, Code duplications, LOC
Application Source code
Breaking Point Tool
Metrics Calculator
Static Analysis
Fan Out, Lack of cohesion between Lines
TD Interest Indicators, TD Principal Indicators, Breaking Point, Interest Probability
Technical Debt Management of embedded platforms

Fig. 1: Tool flow for technical debt management

Probability. Fig.1 graphically depicts the tool flow of TD management (TDM). TDM toolkit relies on three tools: i) SonarQube ii) Metrics Calculator and iii) Breaking point tool.

SonarQube is considered by many the world's leading software quality dashboard. It is based on the SQUALE method and: (a) contrasts the source code of an application with a set of predefined rules, so as to identify violations called code smells, and (b) for each identified violation it calculates a remediation time that is required to resolve it. The sum of the remediation time for all identified violations is recorded as the SQUALE index, representing TD principal. In addition, SonarQube calculates the number of bugs, vulnerabilities and the percentage of duplicated code.

The Metrics Calculator has been developed in the context of SDK4ED with the purpose of calculating maintainability metrics for object oriented and non-object oriented software. For non-object oriented languages (like C) it calculates coupling (Fan out) and cohesion (Lack of cohesion between Lines) per file. Fan out refers to the number of modules called from a file and Lack of cohesion between Lines represents the coherence between all possible pairs of lines of code of a method (aggregated at file level as an average). For object oriented languages (like Java) it calculates 10 metrics: Message-passing couple (MPC), Depth of inheritance tree (DIT), Number of children (NOCC), Response for class (RFC), Lack of cohesion in methods (LCOM), Weighted methods per class (WMPC), Data abstraction coupling (DAC), Number of methods (NOM), Lines of Code (LOC), Number of Properties (NOP).

The results of SonarQube and Metrics Calculator are being used as input to the Breaking Point tool to calculate TD Interest and the time point at which the accumulated interest will exceed TD principal (breaking point) and Interest Probability. TD Interest is calculated based on the FITTED Framework [4].

The FITTED framework has been introduced to measure software sustainability, which is the ability of a system to meet "*the needs of the present without compromising the ability of future generations to meet their own needs*". FITTED measures this as the period in which the cumulative interest is lower than the saved principal. When cumulative interest is equal to the saved principal the system is on its breaking point which means that any savings resulting from the decision to not repay TD will vanish due to increased maintenance effort during evolution. To calculate TD Interest, FITTED suggests the following steps:

1. identify the five artifacts that are most structurally similar to the artifact under consideration
2. based on the values of the selected object-oriented metrics for all structurally similar artifacts, compile an artificial optimal one
3. calculate the average distance of the artifact under analysis from the artificial optimal one—this distance is referred as the ratio of additional maintenance effort
4. calculate the average maintenance product (i.e., lines of code maintained) in each version
5. multiply the ratio of additional maintenance effort with the average maintenance product (extract from past changes on the artifact under analysis)
6. divide the previous outcome with the average lines of code maintained in one hour, so as to retrieve the interest in minutes; and
7. calculate interest in currency using the same hourly rate as in principal calculation.

Finally, Interest probability is a measure of how frequently a file changes in the sense that a file that changes frequently adds more debt (interest).

Results In this subsection we present the results on TD quantification, focusing on TD Principal, TD Interest and Interest Probability for the target system. The results are summarized in Table 1.

The TD Principal varies across system files; in relative terms on can identify file 'reader.cpp' as the one holding the largest principal. This particular file requires 302$ to fix the 29 identified code smells. Even more important is the fact that it also exhibits the highest interest (9.68$) and a very high interest probability (0.8). In other words, this file violates several of the rules checked by SonarQube, its metrics indicates that the maintainability is quite low and has a high probability of being changed in the subsequent version. All these signals provide an imminent risk which should be mitigated by the development team. Similar observations can be made for file 'cisc.cpp'.

In terms of the particular code smells that appear in the code, SonarQube indicates for example that more unit tests should be added so as to increase coverage. For file 'reader.cpp' one of the most demanding issues reads *'92 more lines of code need to be covered by tests to reach the minimum threshold of 65.0% lines coverage'*. Such an issue requires substantial effort to be resolved (estimated by SonarQube to 3h4min). Another striking issue, which also contributes heavily

Table 1: Technical Debt Management output

Source File	**TD Principal**	**TD Interest**	**Interest Probability**	**Code Smells**	**LOC**	**Complexity**
api.cpp	92	0	0	7	93	15
api.h	50	0	0	15	11	0
body.cpp	14	0	0	4	11	2
body.h	10	0	0	1	1	0
resources/imdcode_v1.3/imdcode.c	191	0	0	17	111	26
resources/imdcode.c	183	0	0	18	111	26
main.cpp	53	0	0	7	53	6
misty1.c	148	0	0	7	145	15
misty1.h	65	0	0	16	58	0
reader.cpp	302	9.68	0.8	29	206	25
reader.h	9	0	0	3	1	0
sec_primitives.cpp	23	2.8	0.25	3	24	4
sec_primitives.h	19	0.7	0.25	5	3	0
sims.cpp	14	0	0	4	11	2
sims.h	10	0	0	4	1	0
sisc.cpp	280	30	0.87	26	199	24
sisc.h	9	0	0	3	1	0

to the total principal, is code duplication. For file 'reader.cpp' *2 duplicated blocks of code must be removed* requiring an estimated time of 30 mins.

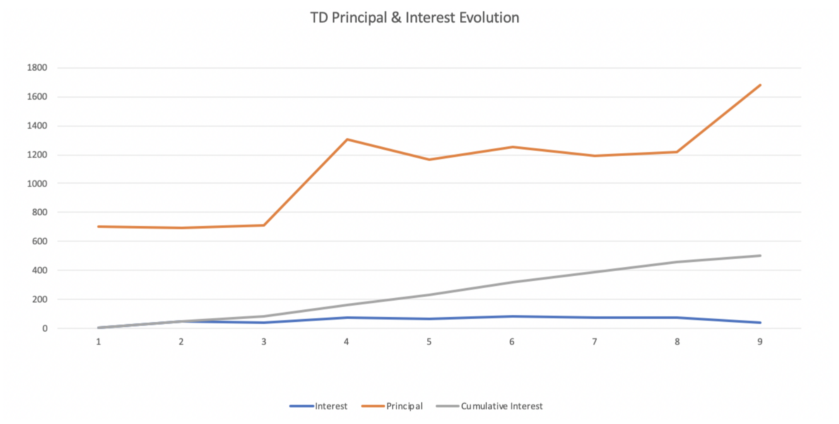
On the other hand, one can observe files with relative high TD Principal (such as imdcode.c) but without any TD Interest or Interest Probability. This is due to the fact that this file has been introduced in the last version and thus it has never been the subject of change. Thus, to be certain whether the development team will face maintainability issues, one should probably wait for data from further revisions. However, we need to note that files with high Principal, but low Interest (that have been maintained for some versions) are usually assigned a low priority for TD management.

These findings imply that a software development team wishing to manage TD in its products, shouldn't focus only on selected figures, but seek a combined interpretation of the findings. In other words, maintenance problems are probable for files with a high TD principal, substantial interest and non-negligible interest probability.

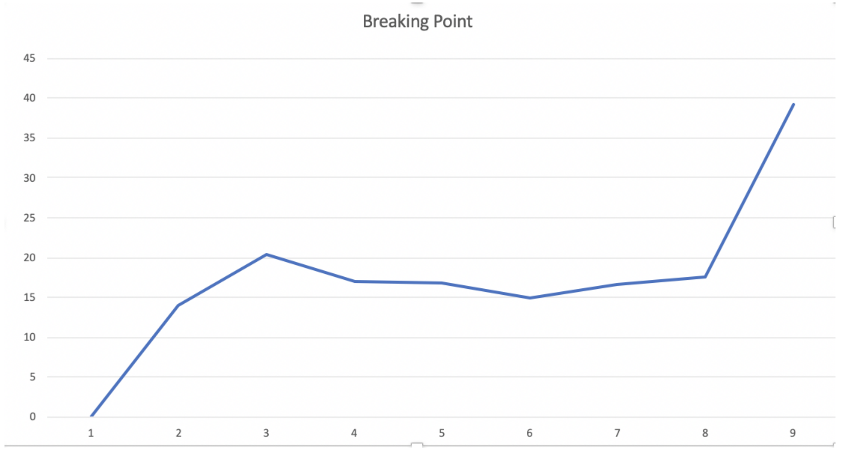
It is well known that all contemporary software systems evolve frequently over time. Thus, beyond the analysis of the current snapshot of each system, it makes sense to observe the evolutionary trends of TD-related concepts. In Figures 2a and 2b we plot the evolution of Principal/Interest and Breaking Point across system versions, respectively.

From Figure 2b, we can observe that the system is slightly deteriorating over time, in terms of TD principal, presenting some high spikes (introduction of considerable TD principal at once) in version 4 and 8. Its interest remains relatively stable; nevertheless, the cumulative interest is naturally increasing almost linearly, as the interest of each version is added to the already existing interest.

However, the evolution of the Breaking Point in Figure 2b reveals a rather healthy project status: The Breaking Point, i.e. the time at which the accumulated interest will exceed the TD Principal lies for most of the versions 20 versions ahead. The Breaking Point is doubled in the final version. This is primarily due



(a) Evolution of TD aspects



(b) Evolution of TD Breaking Point

Fig. 2: Evolution of TD aspects and Evolution of TD Breaking Point

to the addition of new code in the last version, which increases Principal - see Figure 2a (resulting in additional rule violations being detected by SonarQube). However, it seems that the new code has not introduced additional interest, which possibly implies that it is well designed in terms of coupling and cohesion. Considering the rather short history of the project (9 versions), a Breaking Point of 40 version indicates that the development team is not expected to face significant additional maintenance costs in the near future.

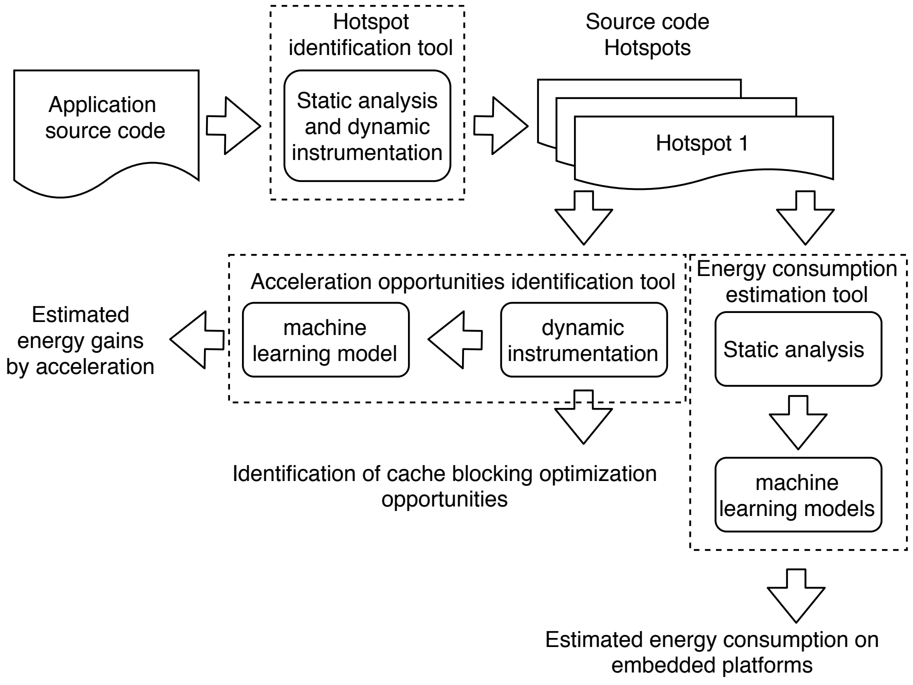

Application source code
Hotspot identification tool
Static analysis and dynamic instrumentation
Source code Hotspots
Hotspot 1
Acceleration opportunities identification tool
machine learning model
dynamic instrumentation
Estimated energy gains by acceleration
Identification of cache blocking optimization opportunities
Energy consumption estimation tool
Static analysis
machine learning models
Estimated energy consumption on embedded platforms

Fig. 3: Tool flow for energy consumption estimation and optimization.

3.3 Energy Consumption Evaluation

Fig.3 shows the tool flow of for energy consumption estimation and optimization. It consists of three tools: i) Hotspots identification ii) Acceleration opportunities identification and iii) Energy consumption estimation.

The *hotspot identification* tool parses the application source code and uses dynamic instrumentation to identify the parts of the application that are computationally expensive in terms of CPU cycles. The main purpose of the tool is to provide the parts of the application source code in which optimizations are expected to have major impact in energy and/or performance. From technical perspective, the hotspot identification tool is based on CLANG and on the Cachegrind tool from the Valgrind benchmark suite. CLANG is used to identify loops and functions through source code static analysis. Valgrind performs dynamic instrumentation and estimates the number of CPU cycles across the application in line-by-line granularity. By combining the outputs of the static analysis and Cachegrind the CPU cycles spent in each loop and function are estimated. The most computationally expensive loops and functions of the application source code are the "hotspots".

The results of the IMD application analysis are shown in Table 2. 5 computationally expensive functions are identified and one critical loop. The starting

Table 2: Hotspot identification output

Hotspot	**Line start**	**Line end**	**CPU cycles**	**Cache miss**	**Function name**	**Source file**
Function-level granularity						
1	218	705	5 %	16 %	main	imdcode.c
2	106	153	4 %	0 %	misty1_encrypt_block	misty1.c
3	32	44	3 %	9 %	fi	misty1.c
4	191	211	3 %	0 %	cmac	imdcode.c
5	47	62	4 %	0 %	fo	misty1.c
Loop-level granularity						
1	202	206	3 %	0 %	-	imdcode.c

and ending line of each hotspot is reported, as well as the percentage of CPU cycles spent in each one. Based on Cachegrind analysis, the cache miss ratio is also reported for each hotspot. Thus, developers may consider applying optimizations that improve cache utilization (e.g. cache blocking) in hotspots with high cache miss ratio.

The hotspots are further analyzed by the *acceleration opportunities identification* tool. This tool is based on dynamic instrumentation techniques. It extracts information from each hotspot, such as ILP level and memory access pattern. This information feeds a machine learning model, which provides an estimation of the energy gains of offloading the specific hotspot on a GPU accelerator. For the specific application no acceleration opportunities were identified. In other words, none of the hotsposts is estimated to provide energy gains by being executed on a GPU.

The *energy consumption estimation* tool, processes the hotspots through source code static analysis and extracts information from the assembly instructions, such as the type of instructions and their sequence. This information is the input of a machine learning model that estimates the execution time and the energy consumption. The tool can provide estimations for any embedded platform provided that the dataset will be prepared and the model will be trained for each one.

Table 3: Time and Energy estimation of various platforms integrating different ARM CPU architectures

Platform (CPU)	**Time (us)**	**Energy Consumption (mJ)**
Nvidia Tegra TX1 (ARM Cortex A-57)	16	0.09
Raspberry Pi 4 (ARM Cortex A-72)	17.5	-
Arduino Nano 33 IOT (ARM Cortex M0+)	1400	0.0008

Table 3 shows the energy consumption and execution time estimation for a number of ARM-based embedded platforms. Energy consumption model was trained for A-57 and M0+ only. However, the execution time model was trained for all embedded platforms of Table 3. The selected CPUs belong to the Cortex A family (A-57 and A-72) and to the Cortex M family, which mainly targets microcontrollers. Developers may exploit these results by selecting the most suitable platform, based on the design constraints. For example, by deploying the IMD application on Arduino Nano, execution time is traded for very low energy consumption.

4 Conclusions

Both TD analysis and energy consumption tools aim at assisting application developers in the process of maintaining and optimizing the application source code. More specifically, the analysis of TD does not aim at characterizing a software system as well- or poor-performing. Rather, it can serve the purpose of raising warnings about repeating code or design inefficiencies, such as lack of unit tests or duplicate chunks of code. The development team, considering also the change frequency of the affected files, can value the merit of such warnings, and proceed to code quality improvements. The application of the SDK4ED platform on the IMD application revealed that a unified view of TD principal, interest and interest probability can help to quickly identify and prioritize code quality improvements on selected artifacts so as to increase their maintainability.

The energy consumption analysis set of tools identify critical parts of the application and provide energy consumption and execution time estimations for various embedded platforms. Thus, developers may obtain estimations in a fast and convenient way and identify performance vs. energy trade-offs by application deployment in various architectures. As presented in the previous section, the IMD application is not suitable for acceleration, however, interesting execution time vs. energy trade-offs have been identified through static analysis at instruction-level for three different ARM-based architectures.

5 Acknowledgement

This work has received funding from the EU's Horizon 2020 research and innovation programme, under grant agreement No. 780572 (SDK4ED, www.sdk4ed.eu). The authors would like to thank the group of Ass. Prof. Christos Strydis at Erasmus MC, Netherlands for providing the implantable medical devices application.

References

1. Ahmed, S., Kakkar, V.: An electret-based angular electrostatic energy harvester for battery-less cardiac and neural implants. IEEE Access **5**, 19631–19643 (2017)
2. Alavani, G., Varma, K., Sarkar, S.: Predicting execution time of cuda kernel using static analysis. In: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom). pp. 948–955. IEEE (2018)

3. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., Systa, K.: The perception of technical debt in the embedded systems domain: An industrial case study. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD). pp. 9–16 (2016)
4. Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: A framework for managing interest in technical debt: An industrial validation. In: Proceedings of the 2018 International Conference on Technical Debt. p. 115–124. TechDebt '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3194164.3194175, https://doi.org/10.1145/3194164.3194175
5. Bazzaz, M., Salehi, M., Ejlali, A.: An accurate instruction-level energy estimation model and tool for embedded systems. IEEE transactions on instrumentation and measurement **62**(7), 1927–1934 (2013)
6. Cunningham, W.: The wycash portfolio management system. SIGPLAN OOPS Mess. **4**(2), 29–30 (Dec 1992). https://doi.org/10.1145/157710.157715, https://doi.org/10.1145/157710.157715
7. Kakkar, V.: An ultra low power system architecture for implantable medical devices. IEEE Access **7**, 111160–111167 (2018)
8. Khan, W., Muntimadugu, E., Jaffe, M., Domb, A.J.: Implantable medical devices. In: Focal controlled drug delivery, pp. 33–59. Springer (2014)
9. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012)
10. Kumar, V., Kakkar, V.: Miniaturized resonant power conversion for implanted medical devices. IEEE Access **5**, 15859–15864 (2017)
11. Mendis, C., Renda, A., Amarasinghe, S., Carbin, M.: Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: International Conference on Machine Learning. pp. 4505–4515 (2019)