On Energy Debt

Managing Consumption on Evolving Software

Marco Couto, Daniel Maia, João Saraiva
HasLab/INESC TEC & Universidade do Minho
marco.l.couto@inesctec.pt
a77531@alunos.uminho.pt
saraiva@di.uminho.pt

Rui Pereira
C4 - Centro de Competências em Cloud Computing
(C4-UBI) Universidade da Beira Interior
Rua Marquês d'Ávila e Bolama, Covilhã
ruipereira@di.uminho.pt

ABSTRACT

This paper introduces the concept of energy debt: a new metric, reflecting the implied cost in terms of energy consumption over time, of choosing a flawed implementation of a software system rather than a more robust, yet possibly time consuming, approach. A flawed implementation is considered to contain code smells, known to have a negative influence on the energy consumption.

Similar to technical debt, if energy debt is not properly addressed, it can accumulate an energy "interest". This interest will keep increasing as new versions of the software are released, and eventually reach a point where the interest will be higher than the initial energy debt. Addressing the issues/smells at such a point can remove energy debt, at the cost of having already consumed a significant amount of energy which can translate into high costs.

We present all underlying concepts of energy debt, bridging the connection with the existing concept of technical debt. We describe our approach with a preliminary motivational example, showing how to compute the energy debt of a real-world application. A prototype is under development, which already includes the detection of several Android energy smells, and calculates the energy debt and energy interest of Android applications.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software performance**.

KEYWORDS

Green Software, Energy Debt, Code Analysis

ACM Reference Format:
Marco Couto, Daniel Maia, João Saraiva and Rui Pereira. 2020. On Energy Debt: Managing Consumption on Evolving Software. In *TechDebt '20: International Conference on Technical Debt, May 25–26, 2020, Seoul, Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Technical Debt (TD) describes the gap between the current state of a software system and the ideal state of that same software. The key idea of technical debt is that software systems may include artifacts which can be hard to understand/maintain/evolve, causing higher costs in the future software development and maintenance activities. These extra costs can be seen as a type of debt that developers owe the software system.

Although technical debt is still a recent area of research, it has gained significant attention over the past years: A recent systematic mapping study [25] identified ten different types of technical debt, namely *requirements*, *architectural*, *design*, *code*, *test*, *build*, *documentation*, *infrastructure*, *versioning*, and *defects* technical debt. In fact, TD is a concern both for researchers and software developers. The current widespread use of non-wired computing devices is also making energy consumption a key aspect not only for hardware manufacturers, but also for researchers and software developers [41]. Indeed, several *energy inefficient* programming practices have been reported in literature, namely, energy patterns for mobile applications [6, 9], the energy impact of code smells [4], energy-greedy API usage patterns [27], energy (in)efficient data structures [31, 37], programming languages [38], etc. which do have significant impact on the energy consumption of software.

All these research works show that energy-greedy programming practices, also called energy smells, do often occur in software systems. These can be attributed to the current lack of knowledge software developers have in order to build energy efficient software, and the lack of supporting tools [42].

This paper defines *energy debt* as the additional estimated energy cost of executing a software system, due to the occurrence of energy smells in the software's source code, when compared to the estimated energy cost of executing the non-energy smelly (*i.e.* energy ideal) version of that same software. To express energy debt we consider a set of energy code smells presented in the current state of the art literature on green software, together with the energy savings reported in the studies where such smells have been presented. Thus, the energy debt of a program is computed after knowing the number of occurrences and their locations in the program's source code: energy smells inside loops/recursion, single statements, or inside dead code do have different debt weights. In order to compute the energy debt of a software system, we have implemented the automatic detection of the full catalog of energy smells in a prototype tool, which is currently being ported to the *SonarQube* framework. Moreover, we have conducted a preliminary experiment of detecting energy smells and computing the evolution of the energy debt on consecutive releases of an open source (GitHub) software system and this paper presents our very first results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
TechDebt '20, May 25–26, 2020, Seoule, Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

This paper is structured as follows: Section 2 thoroughly describes the notion of energy debt, and how it should be expressed/-calculated; A preliminary/motivational example, where we analyzed the evolution of energy debt in a real-world open source project, is described in Section 3; Section 4 presents the related work; finally, our conclusions and future work are included in Section 5.

2 INTRODUCING ENERGY DEBT CONCEPTS

In this section, we explain our definition of energy debt. We start by presenting in Section 2.1 the general idea behind the energy debt concept: what it is, under what assumptions can it be defined, and what are the underlying concepts. After presenting the concepts, we clarify each one in detail. First, we define how to express the smells catalog, i.e., the considered must-fix problems and their associated energy cost, we present an instantiation of such a catalog (Section 2.2). Afterwards, in Section 2.3 we explain how the energy debt can be estimated for a given software release version, using the occurrences of smells found for a given release. Finally, we discuss how this debt can be be transformed into interest with the appearing of new releases, and how to estimate such an interest (Section 2.4).

2.1 Concept Overview

Before we present the definition of energy debt, let us recall the metaphor of technical debt. Technical debt reflects the cost arising from performing additional work on a software system, due to developers taking "shortcuts that fall short of best practices" [1]. In other words, this cost can be defined as the technical effort, in working hours, required for fixing all issues associated with bad programming practices, in a given software release. The cost keeps increasing, as new versions (with new issues) keep getting released, and if the initial issues are not properly addressed, they accumulate *interest* [5].

Based on the underlying concept of technical debt, we define **Energy Debt** as *the amount of unnecessary energy that a software system uses over time, due to maintaining energy code smells for sustained periods.*

A visual comparison of the two concepts is depicted in Figure 1. The left-hand side of the figure illustrates the well-known representation of technical debt, including the concepts of refactoring and maintenance effort, along with the definition of interest. On the right-hand side we present the definition of energy debt, where we assume that evolving the software (i.e., introducing new features on new releases) will eventually result in the addition of new (energy) code smells, hence the Energy Debt (*ED*) increases per version.

The main difference between technical and energy debt, at this point, is the fact that the former can be presented as a unique cost value expressing how much effort would be necessary to address the issues, whereas the same approach cannot be applied to the latter. The cost of maintaining energy code smells in a software release is always directly proportional to the amount of time that the same release operates. As an example, if two software systems S_1 and S_2 have the exact same energy code smells, the amount of excessive energy consumed by S_1 might be higher than S_2 if it is intended to be used longer, during the same timespan.

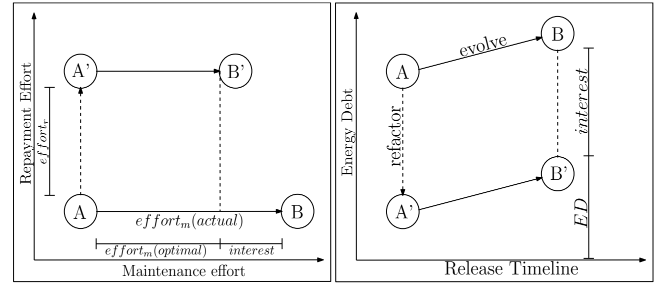

Figure 1: Comparing Technical Debt and Energy Debt Terminology

Given the previous assumptions, we argue that the energy debt *ED* of a software release must be expressed not as a cost value, but as a cost function, which receives, as input, two variables: a software release r, and a usage time t. Equation 1 defines such a function, and it allows us to obtain, for a given release r, its energy debt *ED* after a given usage time of t:

$$ed(r, t) = cost(r) * t \tag{1}$$

The $cost(r)$ function included in the equation represents the energy cost of release r, per unit of time. In other words, it relates to the existing number of energy code smells in that version, and the energy cost (per unit of time) of each one. The definition of that function is expressed as Equation 2:

$$cost(r) = \sum_{i=1}^{N} w_i(r) \times E(i) \tag{2}$$

Here, N is the number of smells included in the considered smell catalog, while $w_i(r)$ returns a weight value for smell i, which is affected by the number of i smells found in release r and the context in which they were found (we will discuss this with greater detail in Section 2.3). $E(i)$ returns the expected energy debt per time unit of smell i, as defined in the smell catalog.

The formulas presented thus far assume that each considered energy code smell has an associated energy debt value, expressed in function of time units (for instance, per minute). Nevertheless, when studying the energy consumption impact of code smells, researchers often tend to present the potential gains/savings as an interval (i.e., highest and lowest observed energy saving). This is due to the fact that measuring energy is not a completely deterministic task, for example, the CPU/room temperature greatly affects energy consumption.

Following the highest/lowest saving approach adds valuable information regarding potential energy savings. A certain smell can have a maximum savings of, for instance, 3000 mJ per minute, and a minimum of 150 mJ per minute. When compared to another smell with respective maximum and minimum savings of 900 mJ and 300 mJ per minute, we know that in a best-case scenario refactoring the first one would result in higher gains, but in a worst-case scenario the second smell presents better savings. Hence, a developer can use this information to decide how to properly focus their attention when refactoring code smells, depending on the project goals [6].

In accordance with the previous assumption, we decided that our approach for energy debt should consider, for each code smell, two energy values: the highest (E_{max}) and lowest (E_{min}) observed energy savings. Since energy debt must be expressed in a function of the usage time, it is expected that E_{max} will be much higher with the increase of usage time, as depicted in Figure 2.

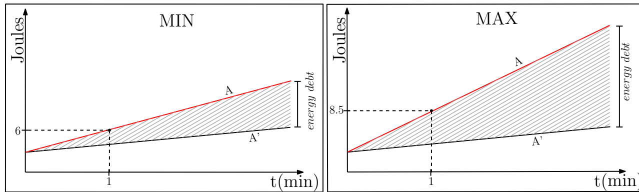

Figure 2: Energy Debt Thresholds Increase Over Time

There are two represented versions of a release in this figure: the optimal version, with all smells removed (A'), and the *energy smelly* version (A). The optimal version already has a constantly increasing energy consumption, as it would be expected. Energy debt can be summed up as the area between the line for A', and the (red) line for A, which becomes much larger when considering the maximum values. This will introduce changes to Equation 1, which will consider two cost values, in the form of two functions:

$$ed(r, t) = \left(cost_{min}(r) \times t; cost_{max}(r) \times t\right) \quad (3)$$

The energy debt will therefore always be presented in the form of an interval. Consequently, each of the cost functions will need to refer to the proper energy debt per time unit. In other words, the $E(i)$ function in Equation 2 will be $E_{min}(i)$ for the lowest savings, and $E_{max}(i)$ for the highest savings.

Next, we introduce a catalog of code smells from current state-of-the-art research, which contain a study an/or report of the code smell's energy impact. We excluded smells which did not provide a way to infer a highest/lowest energy saving value per minute, either because it was not reported by the authors or because the raw results data was not publicly available.

2.2 The Smells Catalog

As previously explained, in order to report the technical debt of a software system/release, it is necessary to consider a set of issues and their severity in terms of refactoring time. Hence, for our energy debt approach, the first step is to define those issues, and the highest/lowest energy gain which can be obtained from refactoring/removing each one.

The energy code smells catalog we consider has been reported in the green software literture and is widely used and studied specifically within the Android ecosystem, where enery consumption is one of the main software concerns. For each energy code smell, we indicate where its energy impact was studied, a brief description on why it has a negative influence on energy consumption, and the corresponding reported maximum and minimum energy savings in milliJoules per minute (E_{max} and E_{min}, respectively).

DA - `Draw Allocation`.

- E_{max} : 158 mJ per minute
- E_{min} : 36 mJ per minute

This is the first of five smells whose energy impact analysis was included in [6, 9, 10]. The authors aimed at understanding how fixing code patterns detected by Android *lint*[1] can improve energy efficiency. *Lint*'s issues are divided into categories, such as Performance or Security. `Draw Allocation`, as well as the next 4 smells, is a **Performance** issue.

`Draw Allocation` occurs when new objects are allocated along with draw operations, which are very sensitive to performance. In other words, it is a bad practice to create objects inside the `onDraw` method of a class which extends a `View` Android component, as we see in the following snippet:

```
public class CloudMoonView extends View {
  @Override
  protected void onDraw(Canvas canvas) {
    RectF rectF1 = new RectF(); ✘
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

The recommended alternative for this smell is to move the allocation of independent objects outside the method, turning it into a static variable, as shown next:

```
public class CloudMoonView extends View {
  RectF rectF1 = new RectF(); ✔
  @Override
  protected void onDraw(Canvas canvas) {
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

WL - `Wakelock`.

- E_{max} : 194 mJ per minute
- E_{min} : 10 mJ per minute

`Wakelock` is the second Android *lint* performance issue [6, 9, 10, 32, 49]. Essentially, *lint* detects whenever a wake lock, a mechanism to control the power state of the device and prevent the screen from turning off, is not properly released, or is used when it is not necessary.

The following snippet shows an example of a wake lock being acquired, but not released when the activity pauses.

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire();  ✔
  }

  @Override()
  public void onPause() { super.onPause(); ✘ }
  }
}
```

The alternative here would be to simply add a `release` instruction as shown next:

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire();    ✔
  }
```

[1] *Lint* is a code analysis tool, provided by the Android SDK, which reports upon finding issues related with the code structural quality. Website: developer.android.com/studio/write/lint

```
    @Override()
    public void onPause() {
      super.onPause();
      if (wakeLock.isHeld()) wakeLock.release();  ✔
    }
}
```

RC - `Recycle`.

- E_{max} : 533 mJ per minute
- E_{min} : 15 mJ per minute

`Recycle` is another Android *lint* performance issue [6, 9, 10]. It detects when collections or database related objects, such as `TypedArrays` or `Cursors`, are not recycled nor closed after being used. When this happens, other objects of the same type cannot efficiently use the same resources.

The following snippet shows a `Cursor` instance being used without being recycled:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  return summoner;  ✘
}
```

The alternative in this case would be to include a `close` method call before the method's return:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  c.close();  ✔
  return summoner;
}
```

OLP - `Obsolete Layout Parameter`.

- E_{max} : 561 mJ per minute
- E_{min} : 94 mJ per minute

The fourth Android *lint* performance issue [6, 9, 10], `Obsolete Layout Parameter`, is the only one that is not Java-related. The view layouts in Android are specified using `XML`, and they tend to suffer several updates. As a consequence, some parameters that have no effect in the view may still remain in the code, which causes excessive processing at runtime. The alternative is to parse the `XML` syntax tree and remove these useless parameters.

The next snippet shows an example of a view component with parameters that can be removed:

```
<TextView android:id="@+id/centertext"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="remote files"
  layout_centerVertical="true"    ✘
  layout_alignParentRight="true"> ✘
</TextView>
```

VH - `View Holder`.

- E_{max} : 2105 mJ per minute
- E_{min} : 892 mJ per minute

`View Holder` is the last Android *lint* performance issue [6, 9, 10], whose alternative intends to make a smoother scroll in *List Views*. The process of drawing all items in a *List View* is costly, since they need to be drawn separately. However, it is possible to make this more efficient by reusing data from already drawn items, which reduces the number of calls to `findViewById()`, known to be energy greedy [27].

In order to better describe this smell, we introduce the following snippet:

```
public View getView(int pos, View cView, ViewGroup par) {
  LayoutInflater inflater = (LayoutInflater) context
    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);

  cView = inflater.inflate(R.layout.apps, par, false);
  TextView txt=(TextView) cView.findViewById(R.id.label); ❶
  ImageView img=(ImageView) cView.findViewById(R.id.logo);❷
  return row;
}
```

Every time `getView()` is called, the system searches on all the view components for both the `TextView` with the id "label" (❶) and the `ImageView` with the id "logo" (❷), using the energy greedy method `findViewById()`. The alternative version is to cache the desired view components, with the following approach:

```
static class ViewHolderItem {
  TextView txtView; ImageView imgView;
}

public View getView(int pos, View cView, ViewGroup par) {
  ViewHolderItem hld; LayoutInflater inflater = ...

  if (cView == null) {                                    ❸
    cView = inflater.inflate(...);
    hld = new ViewHolderItem();
    hld.txtView = (TextView) cView.findViewById(...);    ❹
    hld.imgView = (ImageView) cView.findViewById(...);   ❺
    cView.setTag(hld);
  } else {
    hld = (ViewHolderItem) cView.getTag();               ❻
  }
  TextView txt = hld.txtView;  ImageView img = hld.imgView;
  ...
}
```

Condition ❸ evaluates to true only once, which means instructions ❹ and ❺ execute once, i.e., `findViewById()` executes twice, and its results are stored in the `ViewHolderItem` instance. The following calls to `getView()` will use cached values for the view components `txt` and `img` (❻).

HMU - `HashMap Usage`.

- E_{max} : 229 mJ per minute
- E_{min} : 28 mJ per minute

This smell is related to the usage of the `HashMap` collection [4, 6, 30, 32, 44]. In fact, as stated in the Android documentation page, the usage of `HashMap` is discouraged, since the alternative `ArrayMap` is allegedly more energy-efficient, without decreasing the performance of `map` operations[2].

The alternative is to simply replace the type `HashMap`, whenever it is used, with `ArrayMap`.

EMC - `Excessive Method Calls`.

- E_{max} : 9529 mJ per minute
- E_{min} : 557 mJ per minute

Unnecessarily calling a method can penalize performance, since a call usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards. This penalty was explored by [4, 6, 22], showing that the energy consumption in Android applications can be decreased by removing method calls inside loops that can be extracted from them. An example of an extractable method call would be one which receives no arguments, and is accessed by an object that is not altered in any way inside the loop.

[2] *ArrayMap* documentation: http://bit.ly/32hK0y9.

The alternative is to replace the method call by a variable that is declared outside the loop, and is initialized with the return value of the method call extracted.

MIM - `Member Ignoring Method`.

- E_{max} : 7844 mJ per minute
- E_{min} : 88 mJ per minute

This smell addresses the issue of having a non-static method inside a class, and which could be static instead [4, 6, 32], i.e., it does not access any class fields, it does not directly invoke non-static methods, and it is not an overriding method. Static methods are stored in a memory block separated from where objects are stored, and no matter how many class instances are created throughout the program's execution, only an instance of such method will be created and used. This mechanism helps in reducing energy consumption.

2.3 Counting Expenses & Estimating Debt

With a defined energy smell catalog, the next step is to define a strategy to analyze the occurrence of such smells in a given release. The starting point for this task will be to use a common source code analysis tool capable of detecting code smells. There are several ways to achieve this. For instance, *SonarQube*[3], which is a widely used tool for technical debt estimation, provides an API for defining detection rules for issues/smells of different languages.[4]

Detecting the occurrence of a smell, however, is a necessary but not the sole requirement to properly analyze its impact on energy debt. A smell can be detected, for instance, inside a block of dead/unreachable code, or it can be placed inside a procedure which may only be executed once in a software lifecycle (eg. an initial setup). On the other hand, a code smell can also be part of a mechanism designed to be re-utilized several times, such as a loop, a thread, or a *Service* (very common in Android). These kind of scenarios should be considered when estimating energy debt, and since our energy debt approach implies the usage of statical analysis mechanisms, we can follow already defined strategies for static energy analysis.

A very common and well-established approach for these situations is to define weights for smells, depending on the context on which they are found. For instance, Oliveira et al. [31] and Jabbarvand et al. [19] defined two different strategies for weighing instructions which might be repeated. The former considers the number of times that a given instruction is found in every path in the program call graph. From the call graph, it also accounts for how many occurences are included inside loops; it then defines a heuristic to weigh the energy impact of such instructions in the program under analysis. The latter starts by extracting the full method *call graph* of a program, and for each method, provides an energy score; such a score depends on three things: *(i)* how many paths can be taken to reach that node from the root node, *(ii)* whether it is found inside a loop, and if so *(iii)* what is the expected loop's bound (if it is possible to infer, otherwise they use a default parameterized one); this statically obtained information is then used to increase/decrease the energy score of the node.

[3] *SonarQube* webpage: http://www.sonarqube.org.
[4] Adding Coding Rules webpage: https://docs.sonarqube.org/latest/extend/adding-coding-rules/

Several strategies have been suggested for this task, all of which have been accepted by the community and have promising results. This leads us to believe that, although it is important to weigh code smells depending on the context in which they are detected, several factors can influence the decision on what approach to follow (eg., the amount of information that can be extracted from the smell detection tool, or balancing information detail with the time it takes to run the analysis). Hence, we argue that the followed strategy is also context dependent, and can also be parameterized to be as simple or detailed as desired. Nevertheless, whatever approach one follows, an update to Equation 2 is necessary to consider it. As an example, we will consider a simplified version of the strategy from Jabbarvand et al. [19]:

$$w(i,r) = \sum_{j=1}^{C} paths(j) \times LB \tag{4}$$

In this equation:

- C is the number of i smells found in the release r;
- $paths(j)$ represents the number of paths in the *call graph* through which the jth occurrence of smell i is reachable;
- LB will be 1 **if** the jth of the smell is outside a loop, or a constant indicating the loop bound; it can be inferred if possible, or pre-established.

In order to better explain how all these concepts connect with each other, when aiming at estimating the energy debt of different software releases, we have prepared a running example, depicted in Figure 3. In this example, we have a catalog with 3 smells, each one with the energy gains thresholds defined (values are in milliJoules per minute), and 3 releases with the analysis report for each. The report is a list of the detected smells, where for each one there is information regarding *(i)* the number of paths through which the smell is reachable (*paths*), and *(ii)* whether it was found inside a loop ($LB > 1$) or not ($LB = 1$).

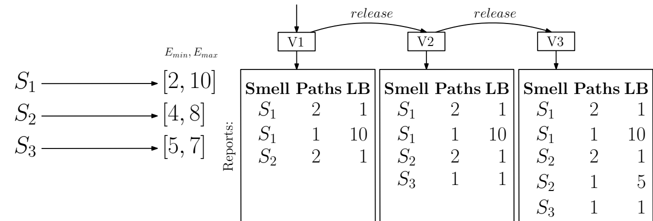

Figure 3: Estimating Energy Debt per Release

Using the formula from Equation 4, we can determine the weight to be applied to each smell. For example, for release $v1$, the weights for smell $s1$ and $s2$ would be:

$$w(s1, v1) = (2 \times 1) + (1 \times 10) = 12$$
$$w(s2, v1) = (2 \times 1) = 2$$

We can then apply the computed weights in the formula from Equation 1, in order to obtain an estimated value for the energy debt of release $v1$. As previously mentioned, our energy debt definition considers two reference values: the lowest and highest estimated

energy debt. This means that the *cost* function in Equation 2 must be computed twice: the first using the lowest estimated gains per smell (E_{min}), and the second using the highest (E_{max}). Once again, for release $v1$, we would have the following $cost_{min}$ and $cost_{max}$ values:

$$cost_{min}(v1) = (w(s1, v1) \times E_{min}(s1)) + (w(s2, v1) \times E_{min}(s2))$$
$$\Leftrightarrow cost_{min}(v1) = (12 \times 2) + (2 \times 4) = 32$$

$$cost_{max}(v1) = (w(s1, v1) \times E_{max}(s1)) + (w(s2, v1) \times E_{max}(s2))$$
$$\Leftrightarrow cost_{max}(v1) = (12 \times 10) + (2 \times 8) = 136$$

These two reference values represent the energy debt for release $v1$. This means that energy debt can vary from a minimum of 32 to a maximum of 136 milliJoules per minute. As explained previously, energy debt is expressed as a function of usage time. Therefore, if one wants to know how much debt this release accumulates after being used for, e.g., one hour, this can be estimated as follows:

$$1h = 60min$$
$$ed(v1, 60min) = \Big(cost_{min}(v1) \times 60; cost_{max}(v1) \times 60\Big)$$
$$\Leftrightarrow ed(v1, 60min) = \Big(1,920mJ; 8,160mJ\Big)$$

The estimated values indicate an energy debt varying between 1.92 and 8.16 Joules per hour. This means that, for every hour that release $v1$ is being used, it could be consuming **at least** $1.92J$ less, and the savings could be **up to** $8.16J$.

Finally, it is important to point out that the accuracy of the estimated thresholds rely on the adequacy/robustness of the analysis components, namely the smells catalog, the code analysis tool, and the weighing function for repeated smell's executions. It is possible to use our energy debt approach to compute reference values for the energy inefficiency of a release, rather then to produce extremely accurate estimates of the potential energy savings per usage time. It depends on how one wants to apply the concept.

2.4 Paying Interests

The concept of *interest* in technical debt has already been formulated [5], and its practical application has also been studied [2, 3, 48]. The concept is based on the fact that, as a software system evolves (i.e., new versions are released), the cost/effort of adding features to a new release (*maintenance effort*, expressed as working hours) keeps increasing if the task of addressing the technical debt keeps being ignored/postponed. Maintaining a release with technical debt requires more effort than to maintain the same release without it; the effort difference between the two is what is called the *technical debt interest*.

The left-hand side of Figure 1 illustrates the interest concept. There is a software version, A, containing code smells, and therefore technical debt. At this point, a decision can be made on what to prioritize: *(i)* invest effort in fixing the smells (*repayment effort*) and release an optimal version of that release, A', without technical debt, or *(ii)* release the version as it is, without dealing with the smells. If the priority is *(ii)*, then the effort of maintaining/evolving to a new release B will be higher than if the priority was *(i)*. This additional effort could be avoided, but the priority was releasing a new version, which can happen for a wide variety of reasons (e.g. client demands, faster market reach, etc.); this resembles the idea of accumulation of debt, and debt needs to be re-payed.

Chatzigeorgiou et al. [5] presented a technique to predict the technical debt *breaking point*, i.e., when will the accumulated interest be higher than the initial effort to remove the technical debt (i.e., from the initial release, which is called *principal*). With this approach, it is possible to present developers with another choice: if technical debt is not addressed now, and it keeps being "ignored", then they have approximately until release number N to properly deal with it; otherwise, from that moment on, the additional *maintenance effort*, arising from not dealing with technical debt, will always be higher than the effort to deal with the *principal*. This ultimately means that, at that point, they are wasting development time.

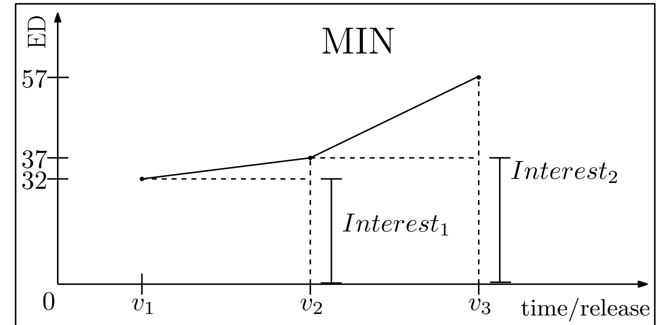

Figure 4: Accumulation of Interest

When considering energy instead of technical debt, the concept of interest needs a different approach to be defined. For starters, it will not tell developers (or project managers) how much additional maintenance effort is being applied. This is due to the fact that energy debt does not measure effort, but the drain of a particular resource. In that sense, the *energy debt interest* is the amount of excessively consumed energy over time, that could be avoided if the issues which caused it were properly addressed earlier. In simple terms, it is the accumulated energy debt after n software releases. This concept complements energy debt in the sense that it can be used to estimate the "real-world" cost of not fixing the smells, which can be (for instance) monetary (since there is always a monetary cost associated with energy) or uptime related (if the software under analysis is targeted for all kinds of mobile devices).

Figure 4 illustrates our perception of interest within energy debt, using the running example from Figure 3. Once again, three software releases are included here ($v1$, $v2$, and $v3$), and the presented values refer to the minimum estimated debt. For this particular example, we are assuming that, as new versions are released, the issues from previous versions are not addressed in any way. Hence, the energy debt is always increasing. For the initial release, $v1$, we consider that no interest was accumulated thus far. This is due to the fact that energy debt is estimated in function of the usage

time, so at the exact instant when the version was released, it was certainly never used.

Considering releases $v2$ and $v3$, we can compute the interest for each one. Starting from $v2$, it was reported that this release added a new smell $s3$. If **all** smells from its previous version, $v1$, were fixed upon releasing $v1$, then this version's minimum energy debt would be 5. However, since $v1$ contained smells, from the time interval comprised between the two releases, the software was excessively consuming 32 mJ for each minute it was being used (i.e., the energy debt from $v1$). Therefore, for release $v2$, the accumulated debt (i.e., the interest) is exactly 32 mJ per minute.

When considering release $v3$, however, the reasoning to infer the interest requires adjustments. For once, $v3$ has two predecessors, while $v2$ has only one. Between $v1$ and $v2$, the energy debt was 32, and between $v2$ and $v3$ it was 37, so the amount of excessively consumed energy over time was not always the same. To estimate how much debt was accumulated, we should infer a value based on the two. One possible way to tackle this is to compute the average of **all** energy debts from previous releases. In this particular case, the minimum accumulated interest would be $\frac{(32+37)}{2} = 34.5$.

Following the presented approach for interest estimation might seem over-simplified at first, but it is a fair portrayal of debt repayment. On one hand, if the energy debt always keeps increasing with new releases, the interest will also keep increasing. On the other hand, if it is gradually reduced, the accumulated interest will also be reduced, which will ultimately mean that the initial debt is being re-payed. Finally, it is important to interpret interest similar to how energy debt is interpreted: an interval describing the minimum/-maximum energy being excessively consumed **per usage time**. With this, we argue that, when considering the interest for the nth release, the expected usage time should be higher than the one for any m^{th} release, where $n > m$. Therefore it is guarantee that, even though the energy debt can be reduced from one release to another, the accumulated interest will be inflated for later releases.

3 PRELIMINARY EXPERIMENT

At this point, we have already presented the concepts which define energy debt. In order to validate this concept, this section will describe an experiment performed over an existing software system, retrieved from GitHub, where we estimate the energy debt for all its releases. We start by presenting the software used for the experiment and the used approach for detecting the energy code smells(Section 3.1); afterwards, having detected the energy smells, we applied an energy debt analysis on each of the software's releases and present the obtained results (Section 3.2).

3.1 The Experimental Software System

In order to fully showcase the energy debt analysis described in the previous sections, we obtained a software system to use for a preliminary experiment, which followed two main requirements. First, it needed to be a system with several explicit releases. This way we could analyze the evolution of energy debt over time, and also present the energy interest values for each release. Second, it must be an Android application. This is due to the fact that 5 of the smells in our catalog are Android specific, and we wanted to explore the catalog as much as possible.

	versions				
	v0.4	**v0.5**	**v0.6**	**v0.7**	**v0.8**
	OLP: 5	*OLP:* 9 ▲	*OLP:* 12 ▲	*OLP:* 17 ▲	*OLP:* 17
smell	*EMC:* 1	*EMC:* 2 ▲	*EMC:* 1 ▼	*EMC:* 1	*EMC:* 1
count			*HMU:* 8	*HMU:* 8	*HMU:* 8
			MIM: 2	*MIM:* 3 ▲	*MIM:* 3

Table 1: Detected smells on *EscapeApp* releases

We searched for applications on GitHub, as it is a well-known and widely used platform to host open-source projects. Using GitHub's search engine, we searched for repositories with two characteristics: *(i)* the repository's main language should be `Java`, as it is the development language for Android, and *(ii)* there should be specific references to the Android framework (i.e., `Java` imports to Android APIs, since we were filtering projects by code). GitHub's search engine only allows the retrieval of the first 1000 search results, hence we only selected the repositories referenced in those 1000 results containing several releases of Android applications.

With the filtered Android applications, the next step was to decide on which one to use for the experiment. Applications without smells had no interest to be analyzed, so defining an approach for detecting smells was necessary at this point. For this purpose, we used our tool called *eSmell Tracker*[5], which is an extension to *lint*[6], the code inspection tool for Android. *Lint* has already built-in detection rules for 5 of the 8 smells in our catalog, thus we developed new rules for the remaining 3 smells. Our tool runs an optimized *lint* check on an Android application's code (i.e., only searching for the 8 energy smells, instead of performing a full analysis), and processes the output to indicate how many occurrences of each smell were found.

After performing the *lint* analysis on all releases from the obtained Android applications, we selected the application with *(i)* a fluctuating number of smells per release and *(ii)* the highest number of smells detected in the latest release. The application that met this criteria was the *EscapeApp*[7], a client application used for a virtual reality escape game, which had 5 explicit releases. Table 1 presents the results of the *lint* analysis for this application. For each release, we indicate the number of occurrences of each detected smell, and how that number varied from the previous release.

3.2 Energy Debt Analysis

The smell detection performed on*EscapeApp* only provided the list of existing smells on each release. The *lint* tool does not offer a built-in mechanism to report the context of each detected smell, so information such as loop bounds could not be obtained automatically. As previously explained in Section 2.3, this information enhances the energy debt analysis. Thus we performed this analysis manually for each release and looked at each smell report to determine if it was being used witin a loop. If so, a loop bound value was inferred considering the loop's code. Although we argue that it is not critical to have this analysis in order to estimate energy debt, for the *SonarQube* extension that we are working on, we intend

[5] *eSmell Tracker* webpage: (omitted to preserve anonymity).
[6] *Lint* webpage: http://tools.android.com/tips/lint.
[7] *EscapeApp* code: https://github.com/alanvanrossum/kroketapp.

to include the smell detection with context information regarding reachable paths and loop bounds inference to further enhance the analysis.

Based on the report presented in Table 1 and the manual code inspection performed, we can estimate the energy debt interval for each release. For instance, after inspecting each smell occurrence in release $v0.4$, we realized that the **EMC** smell was the only one found within a loop. After carefully inspecting and understanding the code, we estimated a loop bound of 10 iterations. Hence, the $cost_{min}$ and $cost_{max}$ values for release $v0.4$ can be calculated as follows:

$$cost_{min}(v0.4) = (5 \times E_{min}(OLP)) + (w_{EMC} \times E_{min}(EMC))$$
$$\Leftrightarrow cost_{min}(v0.4) = 5 \times 94 + 10 \times 557$$
$$\Leftrightarrow cost_{min}(v0.4) = 5,027\ mJ\ (per\ min)$$

$$cost_{max}(v0.4) = (5 \times E_{max}(OLP)) + E_{max}(EMC)$$
$$\Leftrightarrow cost_{max}(v0.4) = 5 \times 561 + 10 \times 9529$$
$$\Leftrightarrow cost_{max}(v0.4) = 98,095\ mJ\ (per\ min)$$

The computed energy debt for each of the 5 versions is illustrated in Figure 5. As expected, the energy debt increases from release $v0.4$ to $v0.5$, as the total number of smells also increases. An interesting aspect, however, is the fact that the energy debt decreases after release $v0.5$. If we look at the detection report on Table 1, we see that it has 2 **EMC** smells. This smell has the highest value for maximum energy debt. It is also the smell with the highest weight, since it is always found inside loops. In that sense, addressing this smell before releasing $v0.6$ eventually payed off, even though other smells were introduced.

Nevertheless, it is important to also look at the concrete values of both the maximum and minimum energy debt. From release $v0.5$ to $v0.6$, energy debt might be reduced **at most** by $\approx 68\ J$, which is a considerable improvement. It is, however, the most profitable scenario in potential, and if we consider a more cautious analysis to the data (i.e., looking at the minimum energy debt variations) we can see that energy debt might only reduced by $\approx 4\ J$. Once again, it is up to the developers and/or product managers to decide how they look at the trade-off between refactoring effort and potential energy savings.

Estimating the accumulated interest for each release is also a straightforward task. If we consider, for example, releases $v0.5$ and 0.7, the minimum interest would be 6.04 J per minute for the former, and 8.67 J per minute for the latter, as the following calculations demonstrate:

$$interest(v0.5) = average(ed(v0.4)) = \left(6.04\ J\ ;\ 98.095\ J\right)$$
$$interest(v0.7) = average(ed(v0.4), ed(v0.5), ed(v0.6))$$
$$\Leftrightarrow interest(v0.7) = \left(8.67\ J\ ;\ 140.198\ J\right)$$

With these reference values, one could easily provide an estimate on how much energy was excessively consumed since the first release until release $v0.5/v.07$. Let us assume that the total amount of time an application is estimated to be used between these releases was, for instance, 5 full days (120 hours or 7,200 minutes). This would mean that when $v0.5$ was released, the energy debt interest would vary between a minimum of $43,488\ J$, to a maximum of $706,284\ J$. When considering $v0.7$, with 3 prior releases, the expected usage time increases by: $3 \times 7,200 = 21,600$ minutes. Therefore, the interest values increase significantly as expected: a minimum of $187,272\ J$ and a maximum of $3,028,276\ J$. Even though this is merely a motivational case study, it is safe to consider that neglecting the energy debt for several releases proves to be highly costly.

4 RELATED WORK

Technical debt is a term which refers to the pitfalls of creating sub-optimal software to fit a shorter interval, introduced by Cunningham [13]. This is a common practice employed to meet short term development deadlines, with the intent of completing it at a later time. As software evolves, it's liable to take on debt from several sources: "technological obsolescence, change of environment, rapid commercial success, advent of new and better technologies, and so on — in other words, the invisible aspects of natural software aging and evolution." [21]. However, more mundane issues can also plague software development, born out of poor coding practices or general ignorance, bringing about many different aspects to technical debt [47].

As already known, allowing technical debt to continuously build up without a level of debt management raises the risk of producing unmanageable and inefficient code, which can hamper the addition of new or updating existing functionalities. This makes it so the longer such code goes unattended, the more resources will be needed to correct it and with diminishing returns [5].

One such inefficiency in software is of high energy consumption. In fact, the profiling, analyzing and improving the energy efficiency of software has become a vey active research field. Studies have shown that developers are aware of the energy consumption problem, and often times seek help in solving such issues [28, 34, 41, 42]. Currently, there is a broad range of work done on understanding what aspects in programming languages can contribute to high energy costs such as different data structures [15, 26, 37, 39, 40], languages [7, 38], memoization [43], design patterns [45], code refactoring [46, 50], and even the testing phase [24].

Specific to the Android ecosystem, there has been research in topics such as the classification of Android applications as being more/less energy efficient [19], identifying energy green APIs [27], estimating energy consumption in code fragments [8, 18], etc. In fact, research in the reduction of energy consumption in the Android system is most likely the most explored environment in this research field over the past decade [4, 9–12, 20, 22, 30, 32, 44, 49]. The results of most of these studies are able to quickly translate into our energy smell catalog to be used in the calculation of energy debt.

Additionally, much research has been conducted in providing several approaches to the measurement of energy consumption. For example for Android energy analysis there is: *eCalc* [14], *vLens* [23], *eProf* [33], *Trepn*[16–19]. There is also work in automatic tools to

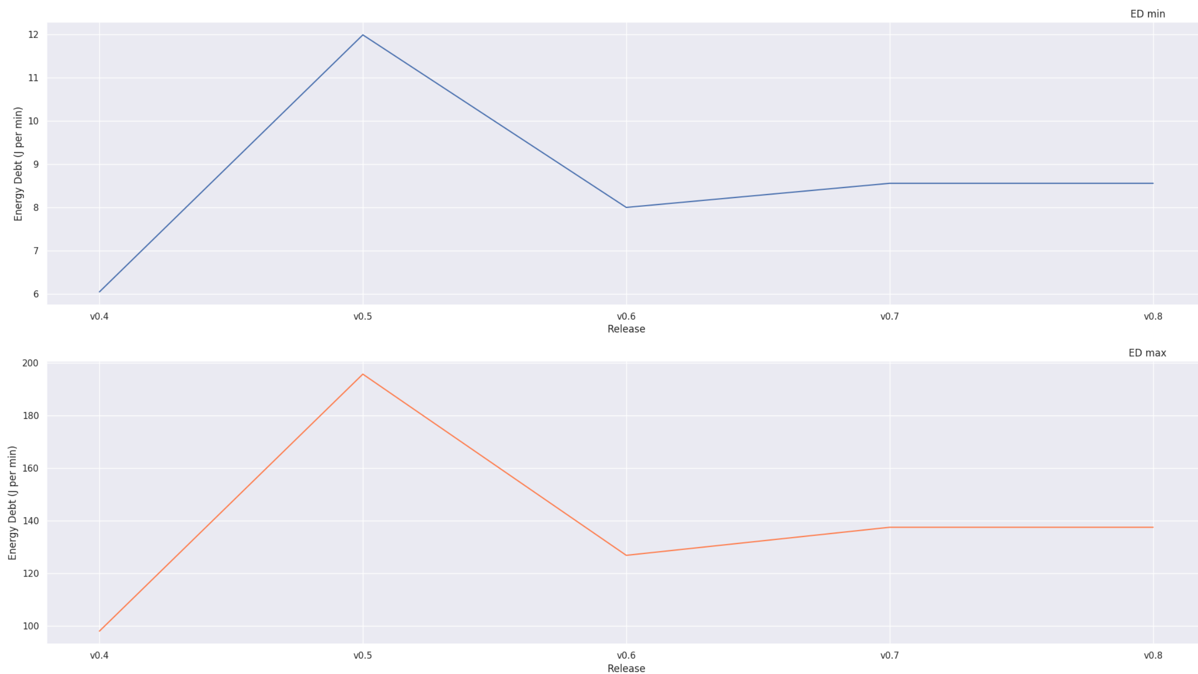

Figure 5: Minimum and Maximum Energy Debt for *EscapeApp* releases

help detect energy greedy code spots [35, 36], automatically refactoring for the most energy efficient data structure [29, 39], or automatically refactoring energy greedy Android patterns [4, 6, 10].

The aforementioned works on the different approaches or tools to reduce the energy consumption of software systems, regardless of their novelty and contributions, do not yet translate their potential gains across a period of time into the actual energy savings (or energy costs) a developer or business can have on his/her software by applying such transformations. It is our belief that, with this work, we have helped close this gap in not only knowing if an alternative solution is more energy efficient, but by how much can we save (in energy consumption or even monetary savings) over time if and when we adopt the energy efficient alternative.

5 CONCLUSIONS AND FUTURE WORK

This paper presented the concept of energy debt as the additional energy cost over time of a software system due to the occurrences of energy code smells in its source code. We have presented a catalog of reported state-of-the-art energy code smells, their known energy costs per usage time, and have expressed energy debt as a function considering *(i)* the number of smells, *(ii)* the context in which they were detected, and *(iii)* the expected usage time of the application. Energy debt interest is also expressed as the accumulation of energy debt per release, which could be avoided by eliminating energy smells in previous releases.

The automatic detection of the presented energy code smells, and the computation of energy debt and interest, has been achieved using our *eBugs Tracker* tool. This tool was used to perform our very first experiment: an analysis on the evolution of energy debt and interest of the *EscapeApp*, a real open source software application which contains four different smells with more than twenty occurrences spread over five releases. Our tool is now being extemded into the *SonarQube* framework, where it will support the inference of the context of detected smells.

The smell catalog which we consider will also be updated with more energy code smells. While the energy impact of code smells is a broad research subject, several research studies do not present enough data to infer energy gain values per usage time. Thus, we focused on two recent studies which presented the energy savings of correctin certain code smells presented in other studies. The accuracy of energy debt estimation can be enhanced if the smells' gains are standardized, to which we plan to improve the catalog to properly address this.

REFERENCES

[1] Eric Allman. 2012. Managing Technical Debt. *Commun. ACM* 55, 5 (May 2012), 50–55. https://doi.org/10.1145/2160718.2160733

[2] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, and Alexander Chatzigeorgiou. 2015. Establishing a Framework for Managing Interest in Technical Debt. In *Proceedings of the Fifth International Symposium on Business Modeling and Software Design - Volume 1: BMSD,*. INSTICC, SciTePress, 75–85. https://doi.org/10.5220/0005885700750085

[3] Areti Ampatzoglou, Alexandros Michailidis, Christos Sarikyriakidis, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2018. A Framework for Managing Interest in Technical Debt: An Industrial Validation. In *Proceedings of the 2018 International Conference on Technical Debt (TechDebt '18)*. Association for Computing Machinery, 115–124. https://doi.org/10.1145/3194164.3194175

[4] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy. 2017. Investigating the energy impact of Android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 115–126.

[5] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis. 2015. Estimating the breaking point for technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 53–56. https://doi.org/10.1109/MTD.2015.7332625

[6] Marco Couto, João Paulo Fernandes, and João Saraiva. 2020. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, Ontario, Canada.

[7] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP 2017)*. ACM, 7:1–7:8.

[8] M. Couto, Carção T., J. Cunha, J. P. Fernandes, and J. Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages*, Fernando Magno Quintão Pereira (Ed.). LNCS, Vol. 8771. Springer Int. Publishing, 77–91.

[9] Luis Cruz and Rui Abreu. 2017. Performance-based Guidelines for Energy Efficient Mobile Applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, 46–57.

[10] Luis Cruz and Rui Abreu. 2018. Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. *CoRR* abs/1803.05889 (2018).

[11] Luis Cruz and Rui Abreu. 2019. Catalog of energy patterns for mobile applications. *Empirical Software Engineering* 24, 4 (01 Aug 2019), 2209–2235.

[12] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. 2019. Do Energy-oriented Changes Hinder Maintainability? arXiv:cs.SE/1908.08332

[13] Ward Cunningham. 1992. The WyCash Portfolio Management System. https://c2.com/doc/oopsla92.html

[14] S. Hao, D. Li, W.G.J. Halfond, and R. Govindan. 2012. Estimating Android applications' CPU energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. 1–7.

[15] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 225–236.

[16] Nagaraj Hegde, Edward L. Melanson, and Edward Sazonov. 2016. Development of a real time activity monitoring Android application utilizing SmartStep. In *Proceedings of the 2016 IEEE 38th Annual International Conference of the Engineering in Medicine and Biology Society (EMBC)*. 1886–1889.

[17] Mohammad Ashraful Hoque, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma. 2015. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.* 48, 3 (2015), 39:1–39:40.

[18] Yan Hu, Jiwei Yan, Dong Yan, Qiong Lu, and Jun Yan. 2018. Lightweight energy consumption analysis and prediction for Android applications. *Science of Computer Programming* (2018), 132–147. Special Issue on TASE 2016.

[19] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *Proc. of 4th Int. Workshop on Green and Sustainable Software (GREENS '15)*. IEEE Press, 8–14.

[20] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. 2017. Detecting Energy Bugs in Android Apps Using Static Analysis. In *Formal Methods and Software Engineering*, Zhenhua Duan and Luke Ong (Eds.). Springer International Publishing, 192–208.

[21] Phillipe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. https://ieeexplore.ieee.org/document/6336722

[22] D. Li and W. G. J. Halfond. 2014. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In *Proc. of 3rd Int. Workshop on Green and Sustainable Software (GREENS 2014)*. ACM, 46–53.

[23] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *Proc. of 2013 Int. Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 78–89.

[24] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. 2014. Integrated Energy-directed Test Suite Optimization. In *Proc. of 2014 Int. Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 339–350.

[25] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and Its Management. *J. Syst. Softw.* 101, C (March 2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[26] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 517–528.

[27] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proc. of 11th Working Conf. on Mining Software Repositories (MSR 2014)*. ACM, 2–11.

[28] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An empirical study of practitioners' perspectives on green software engineering. In *International Conference on Software Engineering (ICSE), 2016 IEEE/ACM 38th*. IEEE, 237–248.

[29] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 503–514.

[30] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. 2018. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering* 44, 12 (Dec 2018), 1176–1206.

[31] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto. 2019. Recommending Energy-Efficient Java Collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 160–170. https://doi.org/10.1109/MSR.2019.00033

[32] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology* 105 (January 2019), 43–55.

[33] A. Pathak, Y. C. Hu, and M. Zhang. 2012. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proc. of 7th ACM European Conf. on Computer Systems (EuroSys '12)*. ACM, 29–42.

[34] Rui Pereira. 2018. *Energyware Engineering: Techniques and Tools for Green Software Development*. Ph.D. Dissertation. Universidade do Minho.

[35] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proc. of the 39th International Conference on Soft. Eng. Companion (ICSE-C 2017)*. ACM, 238–240.

[36] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2020. SPELLing out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software* 161 (2020), 110463. https://doi.org/10.1016/j.jss.2019.110463

[37] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

[38] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. ACM, 256–267.

[39] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 856–859. https://doi.org/10.1145/3238147.3240473

[40] G. Pinto and F. Castor. 2014. Characterizing the Energy Efficiency of Java's Thread-Safe Collections in a Multi-Core Environment. In *Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS*, Vol. 14.

[41] Gustavo Pinto and Fernando Castor. 2017. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* 60, 12 (Nov 2017), 68–75. https://doi.org/10.1145/3154384

[42] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions about Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. Association for Computing Machinery, 22–31. https://doi.org/10.1145/2597073.2597110

[43] Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, and João Saraiva. 2019. Towards using Memoization for Saving Energy in Android. In *Proceedings of the XXII Iberoamerican Conference on Software Engineering (CIbSE)*. 279–292.

[44] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2018. Getting the most from map data structures in Android. *Empirical Software Engineering* 23, 5 (2018), 2829–2864.

[45] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. 2012. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. IEEE, 55–61.

[46] C. Sahin, L. Pollock, and J. Clause. 2014. How Do Code Refactorings Affect Energy Usage?. In *Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, 36:1–36:10.

[47] Edith Tom, Aybüke Aurum, and Richard Vidgen. 2013. An exploration of technical debt. https://www.sciencedirect.com/science/article/pii/S0164121213000022?via=ihub

[48] Angeliki Tsintzira, Apostolos Ampatzoglou, Oliviu Matei, Areti Ampatzoglou, Alexander Chatzigeorgiou, and Robert Heb. 2019. Technical Debt Quantification through Metrics: An Industrial Validation. In *15th China-Europe International Symposium on Software Engineering Education (CEISEE' 19)*. IEEE, –.

[49] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. 2012. Towards Verifying Android Apps for the Absence of No-sleep Energy Bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. USENIX Association.

[50] Roberto Verdecchia, Ren\'e Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. 2018. Empirical Evaluation of the Energy Impact of Refactoring Code Smells. In *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability (EPiC Series in Computing)*, Birgit Penzenstadler, Steve Easterbrook, Colin Venters, and Syed Ishtiaque Ahmed (Eds.), Vol. 52. 365–383. https://doi.org/10.29007/dz83