22nd International Conference on Modeling & Applied Simulation
20th International Multidisciplinary Modeling & Simulation Multiconference

2724-0037 © 2023 The Authors.
doi: 10.46354/i3m.2023.mas.003

Identifying Energy Efficiency Patterns in Sorting Algorithms via Abstract Syntax Tree Mining

Oliver Krauss[1,*] and Andreas Schuler[2]

[1]Advanced Information Systems and Technology, University of Applied Sciences Upper Austria, Softwarepark 13, Hagenberg, 4232, Austria
[2]ELGA GmbH, Treustraße 35-43, Wien, 1200, Austria

[*]Corresponding author. Email address: oliver.krauss@fh-hagenberg.at

Abstract

Energy efficiency is an important topic in the area of mobile computing. Developers are often unaware of the impact their choices on data type use and algorithm design have on this non-functional property. Software energy consumption profiling can be utilized to identify the energy behaviour of implemented methods, while pattern mining can be utilized to identify recurring patterns in the methods being run. We present a methodology to combine energy consumption profiling and discriminative pattern mining to identify energy efficiency patterns. In a study of eight sorting algorithms implemented in Java with the data types *int*, *double* and *Comparable*, profiled on the Android platform, we manage to identify significant patterns in the source code of these 24 implementations. The results show that patterns can be identified for both, the data type in use, and for the energy behaviour of *efficient* or *inefficient* sorting algorithms, that explain the observed energy profiles.

Keywords: Genetic Algorithms; Decision Support Systems; Knowledge Based Systems

1. Introduction

Understanding and optimizing the energy consumption of software has become an important concern for the software engineering research community. This applies particularly in areas with limited resources, e.g. developing applications for mobile devices with constrained battery lifetime. Due to the omnipresence of mobile devices, nowadays for developers it is crucial to have a solid knowledge about an applications' energy consumption, in order to develop energy efficient applications. As pointed out by Hasan et al. (2016), developers often lack the knowledge or experience to optimize a given source code for energy inefficiencies. In addition to that, as described by de Oliveira Júnior et al. (2019), developers often do not share the knowledge of tools and methodologies to assess the energy consumption of an application. This is particularly interesting, given the fact, that the software engineering research community has been spending considerable effort to provide methodologies and tools to better comprehend the connection between software structure, design and API utilization (Hasan et al., 2016; de Oliveira Júnior et al., 2019; Hindle et al., 2014; Hindle, 2012; Schuler and Anderst-Kotsis, 2019; Rocha et al., 2019; Jabbarvand et al., 2015; Wang et al., 2012; Wu et al., 2016).

Energy mining research has already shown that there is a direct correlation between different algorithms for the same problem and energy (e.g. sorting (Schuler and Anderst-Kotsis, 2019)). It has also been shown that the data types utilized play a role in energy consumption as well. While previous research in that area already shows the importance of good algorithm design for energy efficiency, this does not help a software developer make informed decisions about their design. There is a significant gap between the awareness that an algorithm has an impact on energy consumption and *how* and *where* that impact can be attributed to individual statements in the algorithm.

In order to gain insights how software and algorithm design affects energy consumption, we propose a novel methodology to combine software pattern mining with energy profiling. This combination allows a developer to identify energy costly algorithms, and consequently identify specific patterns in the code of the algorithm that are responsible for the energy costs. This enables a rewrite of the algorithm to be more energy efficient.

Software pattern mining has been used in various areas both in research and industry, e.g. mining duplicate code (Qu et al., 2014), or mining faults (Di Fatta et al., 2006), and also to the approach, e.g. the control flow of a function (Henderson and Podgurski), mining in multiple threads (Oßner and Böhm) or the source code itself (Balanyi and Ferenc, 2003). In this work, we utilize mining on the source code in the form of abstract syntax trees (ASTs). The result of this are patterns, i.e. code snippets, that are responsible for high energy consumption.

A thorough research on available studies in the field yielded that the presented approach is the first one to combine energy profiling with software pattern mining. In summary, this paper makes the following contributions:

- A novel combination of energy profiling and software pattern mining based on abstract syntax trees.
- A methodology to identify patterns derived from an in-depth analysis of abstract syntax trees which have a negative impact on the energy consumption of the examined source code.
- We present experimental results on the connection between extracted patterns and software energy consumption in Java. The results are obtained using a benchmark that covers energy consumption of sorting algorithms (Schuler and Anderst-Kotsis, 2019), and available under (Krauss and Schuler, 2021)
- Patterns identifying energy costly source code. These patterns can further be used as foundation to better comprehend the energy implications in Java development.

The remainder of this work is structured as follows. In section 2 we give an overview on the basis of our methodology for energy and pattern mining, and also introduce the benchmark suite used to evaluate our approach. Section 3 presents our approach to obtain patterns and examine its correlation with energy consumption. In section 4 we describe the data-set used to evaluate the presented approach. The results of our experiment are summarized in section 5. Section 6 outlines possible threats to the validity of the presented approach. Finally, section 7 concludes the paper and gives perspectives on future work.

2. Background

2.1. Software Energy Consumption Profiling

Hoque et al. (2016) defines software energy profiling as an approach to assess the energy consumed by an examined software. The foundation for obtaining an energy profile is an energy or power model, that is able to adequately approximate the energy characteristics of a software Researchers further categorize respective models in 3 groups: *utilization-based*, *event-based* and *code-analysis-based* models. A utilization-based model is computed via the actual use of a specific component. E.g. specific Hardware components CPU, Memory, GPS or Display. Event-based models are obtained by correlation of system events (e.g. system-calls) with energy recordings (Aggarwal et al., 2014). The third category solely examines models for energy consumption based on the inspection of source code (Hoque et al., 2016). The approach presented in this paper, falls into the third category, as we seek to understand the energy implications of algorithm design by an in depth examination of syntax trees. The studied sample being used for this purpose was originally obtained using a testbed (Schuler and Anderst-Kotsis, 2019). For better comprehension of this study we included an overview on the approach below.

Algorithm 1: Test method after probe insertion.

```
input  : algorithm - the sorting algorithm
         len - the length of the array
         type - the data type of the array elements
begin
    PauseFor(2);
    toSort ← LoadDataToSort(len, type);
    PauseFor(3);
    LogStartOfExecution(algorithm);
    sorted ← Sort(algorithm, toSort);
    LogEndOfExecution(algorithm);
    PauseFor(3);
```

2.1.1. *The MANA Approach*

The MANA approach was initially described by Schuler and Anderst-Kotsis (2018) and applies a combination of application instrumentation and energy consumption measurements to examine the energy characteristics of an application. The approach further enables fine granular attribution of energy consumption readings to particular software entities of interest, e.g. methods.

The process applied in course of the MANA approach in order to obtain energy readings for an application an attribute them to software entities consists of a series of consecutive steps. First the subject application being analysed is instrumented. Using a gradle plugin, MANA directly alters the Android compilation process. The instrumentation step is crucial, as for the analysis and the correct attribution of the recorded energy profile, it needs to be

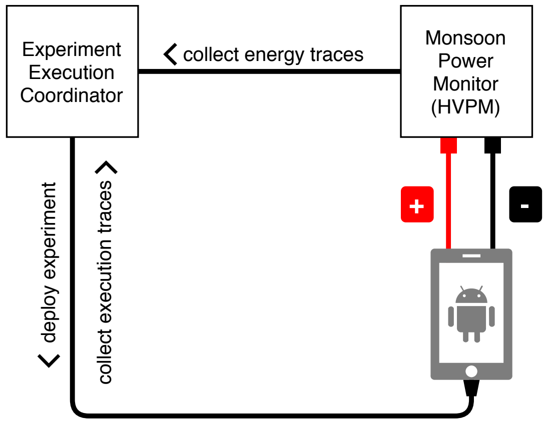

Figure 1. Structure of our experiment test-bed, an Experiment Execution Coordinator is both connected to the Monsoon Power Monitor and the mobile device.

determined when a particular method was entered and exited. To achieve this, MANA relies on the Android Debug class, which allows to collect the call trace of an application during run time. In order to give the reader a general idea how the instrumentation is carried out, refer to algorithm 1, it shows the probes that are inserted during compilation which start and stop the method tracing.

Once the application is instrumented, it can be deployed to the test-bed depicted in Figure 1. For a detailed overview on the test-bed refer to Schuler and Anderst-Kotsis (2019). We outline the basic actors and their responsibilities as follows:

The Experiment *Execution Coordinator (EEC)* is responsible for deploying the instrumented application to an Android device *(2)*. It further executes the desired tests an collects the call traces. Besides that, the EEC is also connected to a measurement device. For the MANA approach, we rely on measurements obtained using the Monsoon HVPM (Monsoon Solutions, 2019) *(3)*, a device which is commonly used in mobile software energy consumption research (Schuler and Anderst-Kotsis, 2020; Altamimi and Naik, 2015; Di Nucci et al., 2017; Wang et al., 2013). Next to the obtained call traces, the EEC collects energy readings from the HVPM and further attributes the collected energy profile to individual methods from the recorded call trace. The EEC finally stores the recorded data which comprises the call trace as well as the per method energy consumption in a Neo4j graph database for further processing and analysis. Using this approach, it is possible to assess the energy consumption characteristics of any arbitrary Android application or library.

For this paper the authors rely on a benchmark that was initially recorded using the MANA approach as part of an empirical study to assess and compare the energy characteristics of sorting algorithms on Android (Schuler and Anderst-Kotsis, 2019). The context of the underlying empirical study is sorting algorithms. In what follows, we give a brief overview of the benchmarks characteristics and how it was obtained (Schuler and Anderst-Kotsis, 2019).

2.1.2. *Sorting Algorithms Energy Benchmark*

Using the MANA approach, a selection of 12 sorting algorithms were examined for their energy characteristics when using different input sizes and data types. Examined input data sizes range from (1) 50,000, (2) 75,000 to (3) 100,000 elements. For each group arrays of data types *int* (4 Byte), *double* (8 Byte) and *java.lang.Comparable<Double>* were randomly generated.

Besides covering the average case for each of the selected sorting algorithms, *ordered* and *reverse-ordered* arrays have been defined, respectively. To summarize, the energy consumption benchmark described in (Schuler and Anderst-Kotsis, 2019) consists of 27 different tests being executed per selected algorithm which results in a total amount of 324 tests. In order to account for possible outliers, tests were sampled 25 times and resulting measurements have been averaged.

By applying the MANA approach to the selected data set, for each test being executed on the testbed, the parameters electrical current in *mA*, voltage in *V* and duration of test execution in nanoseconds were recorded. Using equation 1, energy consumption was computed by multiplying the wattage W at time t with the difference between time t_i and t_{i-1}. The collected parameters were attributed to the examined sorting algorithms.

$$\int_{t_1}^{t_2} power(t)dt \approx \sum_{i=0}^{n} power(t_i) \times (t_i - t_{i-1}) \qquad (1)$$

Figure 2 summarizes the original results for the sorting algorithms selected for this study. It is notable that for each algorithm the energy consumption shows an increasing trend when used with different data types. Furthermore, the energy consumption of algorithms that are commonly known to be efficient considering their time and space complexity, is several magnitudes lower than the energy consumption of its inefficient counterparts. Given this difference, and the fact that sorting algorithms are well studied considering their space and time-complexity, makes this a valuable benchmark for further investigating the cause of this effect. By mining for patterns in the algorithms' abstract syntax trees we seek to get insights on the connection between the structural aspects of an algorithm and its implications to the energy behavior.

2.2. Mining Patterns in Software

Mining patterns in software is conducted on many representations of source code. Mining of text sequences has been conducted often (Ishio et al.; Ueda et al.; Wang et al.; Livshits and Zimmermann) but this reduces the options to deal with the specifics of source code, such as differentiating between branching structures that are contained with each other, or calls between methods.

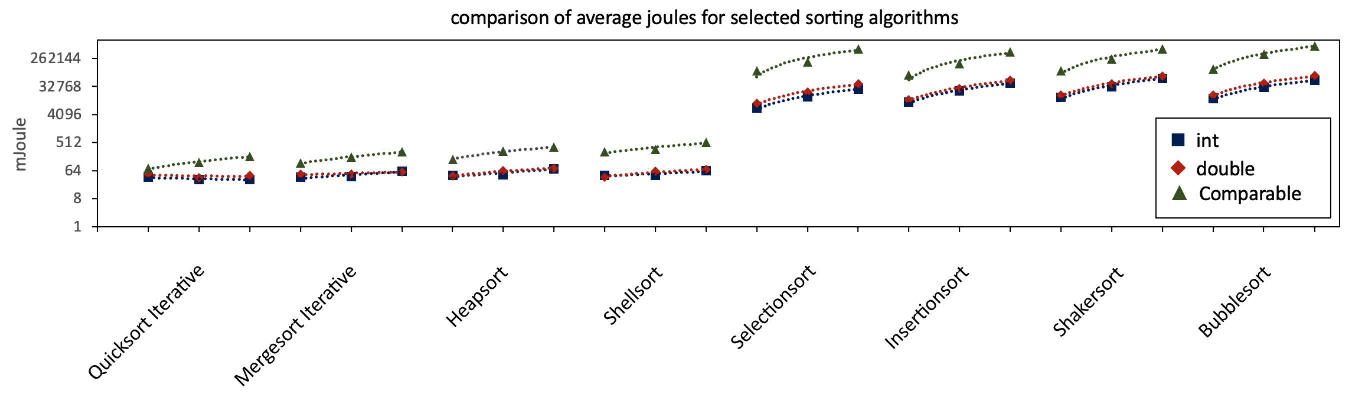

Figure 2. Visual comparison of the the energy consumption of the examined sorting algorithms for different data types (Schuler and Anderst-Kotsis, 2019).

Graph representations have also been used with different levels of granularity (Nguyen and Nguyen; Henderson and Podgurski) and often considering not the source code but the call graphs between different methods (Oßner and Böhm; Cheng et al.; Liu et al.). These approaches are useful when the relationships between functions want to be analyzed, and can gain further information as the control and data flow can be considered for finding interesting patterns.

Abstract Syntax Trees (ASTs) have been used as a representation for pattern mining as well (Hanam et al.; Luan et al.). They provide the advantage of directly representing the source code, similar to sequential structures, and combine this advantage with the information on how the structures and branches are contained, similar to graph structures. As this work attempts to identify patterns that can help the developers to understand what parts of their code negatively impact energy consumption, ASTs have been chosen as a representation.

The domain of discriminative pattern mining (Liu et al.; Di Fatta et al., 2006; Cheng et al.; Thoma et al.) is based on the core concept that patterns will occur with a different frequency in between two groups. In literature these groups are positive and negative testing groups that denominate successful and failed tests respectively. This concept can be expanded for the purpose of energy pattern mining by considering 2 to n groups as necessary. For example the groups *efficient* and *inefficient* would still denominate positive and negative testing groups. But discriminative pattern mining can also consider the data types used in the specific implementation of algorithms to denominate the groups *int*, *double* and *Comparable*.

Discriminative pattern mining observes the frequency of occuring patterns. Frequent subgraph or subtree mining are complex issues in this domain. The reason being that the search space, i.e. all permuations, of a single tree is 2^n where n is the amount of relationships in the tree. As this is a complex topic that is highly relevant for the scalability of pattern mining approaches many different algorithms have been suggested as solutions (Yan et al.; Henderson and Podgurski; Nguyen et al., b,a; Han et al.; Ester et al.; Zaki). Generally these algorithms can be categorized into three different categories.

The *apriori* approach (Agrawal and Srikant) starts out by creating all permutations of size 1. It then evaluates these permutations and applies pruning, after which the size is increased by 1, and the process is repeated until the entire search space is evaluated.

Pattern growth algorithms (Zaki) start out with patterns and only grow a given pattern in a direction that is guaranteed to be interesting according to a specified metric. This ensures that only such patterns are grown that are of interest in the mining context, and makes this type of algorithm more efficient than apriori algorithms.

Non-optmiality guaranteeing methods, i.e. such methods that prioritize dealing with the search space in an efficient way over guaranteeing that all possible instances of the search space are evaluated. For example this is done with clustering of the data (Ester et al.) or via a greedy pruning of the search space (Luan et al.) which can also be done based on regular apriori or pattern growth mechanisms.

As the search space in this work is restricted to 8 algorithms with three data type implementations each, totalling 24 ASTs, an evaluation of the entire search space is possible without tradeoffs. Thus the basis for this work follows a pattern growth approach.

3. Methods

3.1. Encoding the Source Code

In order to have a uniform representation of the algorithms being analysed, as a preparatory step we encode the source code in form of an Abstract Syntax Tree (AST). In particular, our approach derives a per method AST, as the energy profiles available in the data set have been recorded and attributed with the same granularity. Thus, encoding a class results in not only one, but a number of abstract syntax trees which is equal to the number of methods being present in the examined classes. The ASTs being derived per method preserve the structural aspects of respective method down to individual expressions. The encoding step also takes method invocations into consideration. Whenever a method contains non-recursive invocations to a target method, the target method is inlined.

To prepare the selected data set for the experiment, using our approach, we obtained the abstract syntax tree for each algorithm under analysis and stored its representation in a neo4j database. For parsing and constructing the resulting syntax trees, we use ANTLR together with the officially maintained Java grammar available in the ANTLR GitHub repository (https://github.com/antlr/grammars-v4).

3.2. Mining Patterns from Evaluated ASTs

Mining patterns from an AST that was generated from source code means that in the mining itself no dynamic information, such as the call graph or branch execution counts are available. The only dynamic information available is the non-functional features captured during the profiling phase. These features allow us a classification into groups and utilization of a discriminative pattern mining approach. For this we mine on a per-method basis as the profiling provides a value per method as well.

3.2.1. *Preliminary Analysis*

Discriminative pattern mining in literature usually splits the dataset into two groups *passing* and *failing* according to if the code has passed or failed a test (Di Fatta et al., 2006). When considering energy consumption, methods that have vastly different concerns and thus no semantic and little syntactic overlap would be grouped into the same class (Schuler and Anderst-Kotsis, 2019). To ensure that meaningful patterns can still be mined from such a class we conduct a preliminary analysis of the given methods.

Algorithm 2: Find Smallest Infrequent Pattern

```
   input  : asts - the given asts for mining
            cnt - the total amount of asts
1  size ← 0;
2  patterns ← ∅;
3  while patterns = ∅ do
4      size + +;
       /* Find patterns of size                    */
5      patterns ← FrequentSubgraphs(asts, size);
       /* Filter all patterns occuring in all ASTs
          */
6      patterns ← Filter(patterns,);
   output : patterns
```

This is done via algorithm 2. The algorithm attempts to find the smallest patterns that show any difference, i.e. any pattern with less than 100% frequency, between a given group of ASTs, finding none if the ASTs are equivalent. It finds the smallest infrequent patterns instead of the largest, as an infrequent structural difference in an AST will always grow around core patterns resulting in more and more infrequent patterns growing around this one core pattern. While this algorithm does not provide a similarity score its results provide the following insights:

- The ASTs correlate more with each other the larger the found patterns are.
- The ASTs correlate less with each other the more patterns are found.
- The size of the pattern is an indicator of what size should be used for the pattern mining in following steps. Due to patterns growing around a core restricting to smaller sizes improves the *precision*, meaning how many patterns with a similar core meaning are mined.

3.2.2. *Discriminative Pattern Mining*

Algorithm 3: Discriminative Frequent Subgraph Mining

```
    input  : classes - the grouped asts for mining
             minSupport - minimum support threshold
             minDiff - minimum difference between
    classes
1   patternsPerClass ← ∅;
    /* Frequent subgraph mining per class.
       Patterns have a support frequency in
       percent                                        */
2   for class in classes do
3       patternsInClass ← FrequentSubgraphs(class,
         minSupport);
4       patternsPerClass.add(patternsInClass)
    /* Filtering patterns occuring more often in
       one class                                      */
5   distinctPatterns ← Distinct(patternsPerClass) ;
6   for distinctPattern in distinctPatterns do
7       for class in classes do
8           support ←
             patternsPerClass.class.distinctPattern;
9           for cmpClass in classes do
10              if cmpClass ≠ class then
11                  classSupport ←
                     patternsPerClass.class.distinctPattern;

12                  if minDiff > (support − classSupport)
                     then
13                      patterns.add(distinctPattern);
    output : patterns
```

Discriminative Pattern Mining is usually done for fault localization (Di Fatta et al., 2006; Cheng et al.; Henderson and Podgurski). Because of this it usually only discriminates between *passing* and *failing*, and provides metrics that are specific to these two classes. As we attempt to mine energy consumption we are not necessarily restricted to

Table 1. Curated list of examined algorithms, sample taken from benchmark described in Schuler and Anderst-Kotsis (2019).

	Time Complexity			Space Complexity
Algorithm	Best Case	Average Case	Worst Case	Worst Case
Quicksort Iterative	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log(n))$
Mergesort Iterative	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Heapsort	$\Omega(n\log(n))$	$\Theta(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$
Shellsort	$\Omega(n\log(n))$	$\Theta(n(\log(n))^2)$	$\mathcal{O}(n(\log(n))^2)$	$\mathcal{O}(1)$
Selectionsort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertionsort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Shakersort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Bubblesort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

two classes *inefficient* and *efficient* but could also compare along more efficiency classes. We also compare differences between the datatypes *int*, *double* and *Comparable*, which show a visible difference in all algorithms in Figure 2, and have been proven to be statistically significant between *int* and *Comparable* as well as *double* and *Comparable* by Schuler and Anderst-Kotsis (2019).

To enable discriminative pattern mining on an indeterminate amount of classes we introduce the Discriminative Frequent Subgraph Mining algorithm 3 algorithm. The algorithm works by conducting frequent subgraph mining on each of the given classes returning patterns with a given *minimum support threshold*, i.e. how often the pattern must at least occur in over all ASTs. This step provides all patterns relevant in a given class that may be responsible for its non-functional behaviour, e.g. why it is energy efficient or energy inefficient. To reduce false positives in a second step the patterns are filtered via *minimum difference* which ensures that false positives, i.e. patterns occurring often in more than one class, are reduced. The remaining patterns are likely candidates, that in the context of energy pattern mining, explain why a class is inefficient or efficient compared to others.

This filtering approach however comes with a caveat, namely that outliers are filtered out in the first step via *minimum support threshold*. These outliers are important as they may negate the assumption that a found pattern is responsible for energy behaviour. This makes it necessary that following the mining approach the ASTs in the dataset are checked for containment of the identified pattern to determine and explain possible outliers.

4. Evaluation

We based our study on the sorting algorithm benchmark described in Schuler and Anderst-Kotsis (2019). However, we did not include the originally examined algorithm Quicksort Dual Pivot. The main reason for this was, the original study referred to an implementation which is part of the Java standard library and therefore the algorithm is designed in compliance with general OO-principles. This however, makes it hard to factorize the relevant syntax trees to allow for a meaningful comparison in this study. Additionally, for each sorting algorithm examined in the original benchmark, we only consider its iterative counterpart, excluding the recursive version. As a result the final curated list of algorithms used in this study contains the 8 algorithms presented in Table 1.

From the original study we obtained the source code of each selected algorithm together with its energy and run-time characteristics. For each of the algorithms we computed the abstract syntax trees down the the expression level using ANTLR (cf. Section 3). If possible method calls were inlined. The resulting syntax trees are stored in a Neo4j database with attributed energy and run-time characteristics, respectively.

5. Results

To identify patterns in algorithms that have a negative impact on the energy consumption, the encoded ASTs of the given algorithms need to be clustered into groups. Figure 2 shows the evaluated sorting algorithms with their respective energy consumption in mJoule. Table 1 also shows the time complexity for these algorithms. From this the following clustering approaches become apparent:

Inefficient vs. efficient
: Selection- Insertion- Shaker- and Bubblesort show a similar efficiency that is worse than all other algorithms denoting the inefficient group. As the other algorithms show similar behaviour they can be denoted as their own group to compare to the efficient group.

Datatypes
: The datatype *Comparable* shows an energy inefficiency compared to int and double, which show a similar energy consumption to each other. This difference has been shown to be statistically significant (Schuler and Anderst-Kotsis, 2019), it is worth investigating if responsible patterns can be identified.

The discussion hereafter is based on the ASTs of the algorithm implementations as they were instrumented and profiled for energy consumption. The source code, ASTs and results of the mining approach are available in (Krauss and Schuler, 2021).

5.1. Preliminary Analysis

The methods presented in this work attempt to mine patterns exclusively on a structural basis, not taking into consideration dynamic information such as which AST nodes were actually called or how often, or the call graph between functions and classes.

As the analysis compares sorting algorithms, serving the same goal it can be assumed that a structural comparison makes sense, for comparing inefficient and efficient algorithms. For the datatypes however, pattern mining is irrelevant if an individual algorithm is implemented differently for the different datatypes. E.g. in the three algorithms Quicksort-int, Quicksort-double and Quicksort-Comparable only the used datatype should be different. As an initial step we compared the three datatype versions of each sorting algorithm with each other to verify this, using algorithm 2.

The resulting analysis shows absolutely no differences between *double* and *int*, and only one core difference to *Comparable* for every algorithm. This difference is always the invocation to *Comparable.compareTo*, with different pattern sizes, as algorithms contain method invocations, the pattern is found with a size 2 via *Expr* $\leftarrow$ *MethodInvoc*. In the cases of QuickSortIterative we find 7 patterns of size 3 and in MergeSortIterative we find 9 patterns of size 3, all of them growing around *compareTo* or the absence of it.

This finding also indicates that an analysis of the data types and inefficient vs. efficient algorithms should be conducted around a pattern size of 3. It has also been previously suggested that smaller patterns are more useful for evaluation (Cheng et al.), and a size 3 is the minimal size promising patterns seem to be identifiable.

5.2. Comparison of Data Types

The preliminary investigation has already shown that the call to the function *compareTo* is the pattern likely explaining why the comparable data type is more energy inefficient than int and double. Conducting discriminative pattern mining in the datatype categories does indeed not yield any further patterns. Figure 3 shows how most patterns are grouped. They are either not discriminative, occuring in all three data type implementations, or always occur in the *int* and *double* algorithms together but not the *Comparable*. The third option, only occuring in *Comparable* is omitted from the figure.

This indicates that the *compareTo* method call is responsible for the energy consumption overhead over int and double. However we can not conclusively claim that this is the only reason *Compare* is less energy efficient, as this is also the only data type that is a class and not a primitive data type, which *int* and *double* are.

5.3. Inefficient vs. Efficient Algorithms

To find patterns that may be responsible for the different energy behaviours of sorting algorithms we compared the two groups *inefficient* (Selection- Insertion- Shaker- Bubblesort) and *efficient* (QuickSortIterative, MergesortIterative, Heapsort, Shellsort) via discriminative pattern mining. The *minimum support threshold* for each group was set to 0.75, i.e. a pattern must occur in 75% or more trees of

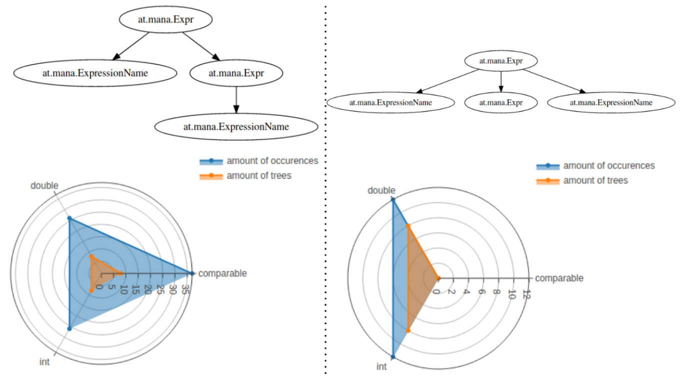

Figure 3. Most patterns occur in all data type implementations (left), while some occur only in *double* and *int* as *Comparable* has a method invocation at the position of the lower Expr node (right).

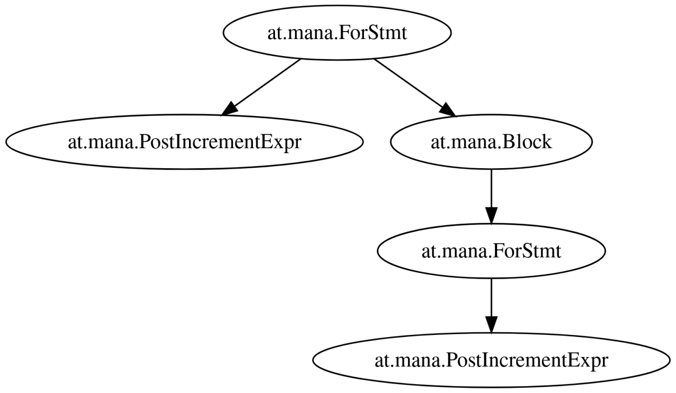

Figure 4. A nested for loop conducting a linear increment in both the inner and outer loop is responsible for the inefficient run-time performance as well as the energy performance of sorting algorithms.

a group to be relevant. To compare the differences between the groups *minimum difference* was set to 0.75, i.e. There must be at least an 75% difference between occurrences of efficient to inefficient and vice versa. The combination of those two settings means that essentially we mine only patterns that occur either mostly in the efficient or mostly in the inefficient groups.

As the preliminary analysis yielded a pattern size of 3 to be an interesting search space this size was chosen. Indeed a search of < 3 yields no patterns at all. The size of 3 yields a pattern around the Method Declaration in the *efficient* search space, as all algorithms share a variable initialization occuring before the loops conducting the sort. This pattern already identifies the variables utilized by the efficient algorithms in the loops itself but not its use. Thus we extended the pattern size to 4, which starts showing 12 patterns exclusive to the *inefficient* space where the loops favor increments of the loop variable, and 3 assignments exclusive to how the loop variables are handled in the *efficient* algorithms, we continued increasing the pattern size after this, but larger pattern sizes only show patterns grown around these core patterns but not yielding additional information.

The reason the *inefficient* algorithms are energy inefficient seems to be that all of them contain two nested for loops. The inner nested for loop conducts a PostIncrement which is exclusive to the inefficient algorithms. The pattern has one outlier in its class, the InsertionSort which conducts a PreIncrement (j−− instead of j++) in the inner loop. This is essentially the same functionality, but from the perspective of pattern mining not equal in one node, and thus an outlier. The outer loop of all these algorithms conducts a PostIncrement, identified in two patterns, one shared with ShellSort, and one shared with MergeSort. Figure 4 shows a pattern of size 5 containing both loops and respective increments exclusive to the negative search space. Listing 1 shows how ShellSort instead improves its efficiency by using an expression for the inner loop steps instead of a linear decrement or increment.

Listing 1. Nested for-loop in ShellSort does not produce O(n^2). The inner loop iterates only over every h'th element.

```
for (int i = h; i < N; i++) {
    for (int j = i;  j >= h &&
       (a[j].compareTo(a[j - h]) < 0 ); j -= h )
        Util.swap(a, j, j - h);
}
```

6. Threats to Validity

Although our study shows promising results, there are several threats to its validity.

First, we only consider sorting algorithms. Therefore, the generalizability of our results is limited. However, we believe, that the presented approach can easily be applied to different algorithm domains. Second, the results of our study are based on a dataset which was initially recorded on one device. It is likely that when the energy consumption is recorded on a different device, the energy behaviour will be different. However, in this study we are only interested in the relative energy difference between the examined algorithms. We are confident, that our results still hold if applied to data that was recorded on a different device.

Another threat to validity stems from how the mining was conducted. We conducted the mining only on the structural information of the source code represented as an AST. This omits information that could be gathered from the hot path, and thus provide additional information which parts of the code contribute most to the energy consumption as they are executed more often than others. Not utilizing this, however does mean that there are no additional threats from the instrumentation that would be introduced by analyzing the hot paths that could possibly influence the energy measures. Also the patterns we find occur related to the step variables of the loops, which are in the hot path of sorting algorithms. This threat is primarily to the generalizability of the approach.

The granularity of the approach itself may present a threat to the experiment design. We intentionally chose a high level of granularity, branches, statements and

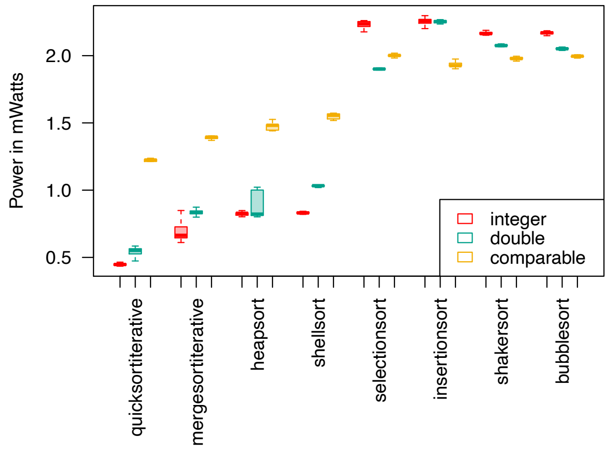

Figure 5. Comparison of average wattage between different data types for all examined algorithms (Schuler and Anderst-Kotsis, 2019).

method calls, as we assume that the energy consumption will be less likely attributable to more fine granular structures. A future comparison between different granularity levels is needed to verify if this assumption holds true.

7. Conclusion and Outlook

Schuler and Anderst-Kotsis (2019) have clearly shown that energy energy profiling can identify the energy consumption impact of different algorithm implementations and data types utilized. This work expands upon this concept and combines it with software pattern mining to also identify patterns in the source code that may be responsible for this behaviour. The results show that we can successfully identify such pattern, both in the energy consumption resulting from using different data types, introduced by the additional compareTo call on the Comparable datatype, and in the implementation of an algorithms from the use of nested for loops iterating in single steps without skipping any value over the entire data set.

The insights gained, especially concerning the *inefficient* algorithms and the identified nested for loop pattern are of particular interest to give an explanation on the power behavior of the examined algorithms. Figure 5 depicts the average power for the sorting algorithms (Pinto and Castor, 2017). What's interesting is that all efficient algorithms show an expected divergence in average power between examined data types, meaning the smaller the data type being used, the lower the average power (Pinto and Castor, 2017). This does not apply to the inefficient group in Figure 5. Therefore, we believe that the identified nested for loops could serve as a possible explanation for this behavior. In essence, the effect of the nested for loop on the average power is likely to superpose the effect examined considering power differences in data types.

This shows that using energy profiling via pattern mining, specifically to identify patterns that have a negative impact on energy consumption, can help to identify such patterns, and identify what datatypes to consider or constructs to avoid when designing an algorithm. In the future, identifying such patterns in code may help developers write code with energy efficiency in mind, which is a current issue in software development (Hasan et al., 2016).

Our approach can be applied on any scale of software, and applied to programming languages other than Java, as only an AST representation of the source code is needed, Energy measurements need to be provided according to the MANA approach, on a per-method basis.

7.1. Future Work

In the future we want to conduct an investigation of energy patterns in a larger setting. Energy consumption can be attributed towards method calls, as seen by the call to compareTo in the Comparable data type. This in itself bears further investigation if the type as a class instead of a primitive also influences the energy behaviour, which can be achieved by modifying the algorithms accordingly and conducting an additional energy performance evaluation.

The presented results were achieved with a relatively high level look at the source code, considering only control structures (for, if, ...) and statements but not the exact nature of a statement. While and For loops arguably should be considered the same type of node during pattern mining. In a similar vein the outlier in the *inefficient* algorithms which utilizes a variable decrement instead of an increment might also be considered the same. This indicates that multiple granularity levels of code may be interesting for observing patterns, and could be done by introducing a taxonomy where more fine granular concepts are generalized, e.g. generalizing while and for into loop, and in turn generalizing loop and if into branching statements. Further investigation of acceptable pattern sizes to identify patterns responsible for energy-behaviour is necessary, and how finely granular patterns can be to still attribute them to energy behaviour.

This work relies solely on the AST of given source code, ergo a static approach of mining. Static approaches are advantageous over dynamic ones as they do not require code execution and thus have less set-up effort and a decreased run-time. However since the energy consumption is measured, which requires execution of the code as a dynamic approach these advantages do not come into play for the presented approach. Future work will also consider additional information in the AST structures that can be extracted from call traces, to not only consider the source code, but also execution details. Care must be taken to ensure that conducting these traces will not negatively influence the energy measurement.

References

Aggarwal, K., Zhang, C., Campbell, J. C., Hindle, A., and Stroulia, E. (2014). The power of system call traces - predicting the software energy consumption impact of changes. *CASCON*.

Agrawal, R. and Srikant, R. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499. Morgan Kaufmann Publishers Inc.

Altamimi, M. L. and Naik, K. (2015). A Computing Profiling Procedure for Mobile Developers to Estimate Energy Cost. In *the 18th ACM International Conference*, pages 301–305, New York, New York, USA. ACM Press.

Balanyi, Z. and Ferenc, R. (2003). Mining design patterns from c++ source code. pages 305– 314.

Cheng, H., Lo, D., Zhou, Y., Wang, X., and Yan, X. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 141–152. Association for Computing Machinery.

de Oliveira Júnior, W., dos Santos, R. O., de Lima Filho, F. J. C., de Araújo Neto, B. F., and Pinto, G. H. L. (2019). Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 160–170. IEEE.

Di Fatta, G., Leue, S., and Stegantova, E. (2006). Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd International Workshop on Software Quality Assurance*, SOQUA '06, page 62–69, New York, NY, USA. Association for Computing Machinery.

Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., and De Lucia, A. (2017). Software-based energy profiling of Android apps: Simple, efficient and reliable? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER*, pages 103–114. IEEE.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press.

Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., and Hsu, M.-C. FreeSpan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 355–359. Association for Computing Machinery.

Hanam, Q., Brito, F. S. d. M., and Mesbah, A. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 144–156. ACM Press.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236.

Henderson, T. A. D. and Podgurski, A. Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 93–104.

Hindle, A. (2012). Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 78–87. IEEE Press.

Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C., and Romansky, S. (2014). GreenMiner: a hardware based mining software repositories software energy consumption framework. In *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 12–21, New York, New York, USA. ACM Press.

Hoque, M. A., Siekkinen, M., Khan, K. N., Xiao, Y., and Tarkoma, S. (2016). Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys (CSUR)*, 48(3):39.

Ishio, T., Date, H., Miyake, T., and Inoue, K. Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs. In *2008 15th Working Conference on Reverse Engineering*, pages 123–132. IEEE.

Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., and Ammann, P. (2015). EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software (GREENS)*, pages 8–14. IEEE.

Krauss, O. and Schuler, A. (2021). Identifying Energy Efficiency Patterns in Sorting Algorithms via Abstract Syntax Tree Mining.

Liu, C., Yan, X., Yu, H., Han, J., and Yu, P. S. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 286–297. Society for Industrial and Applied Mathematics.

Livshits, B. and Zimmermann, T. DynaMine: Finding common error patterns by mining software revision histories. 30(5):296–305.

Luan, S., Yang, D., Barnaby, C., Sen, K., and Chandra, S. Aroma: Code recommendation via structural code search. 3:152:1–152:28.

Monsoon Solutions (2019). Monsoon Solutions high voltage power monitor. `http://msoon.github.io/powermonitor/HVPM.html`. Accessed: 2020-05-25.

Nguyen, A. T. and Nguyen, T. N. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868. IEEE Press.

Nguyen, H. A., Nguyen, T. N., Dig, D., Nguyen, S., Tran, H., and Hilton, M. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 819–830. IEEE Press.

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. Association for Computing Machinery.

Oßner, C. and Böhm, K. Graphs for Mining-Based Defect Localization in Multithreaded Programs. 41(4):570–593.

Pinto, G. and Castor, F. (2017). Energy efficiency: a new concern for application software developers. *Communications of the ACM*, 60(12):68–75.

Qu, W., Jia, Y., and Jiang, M. (2014). Pattern mining of cloned codes in software systems. *Information Sciences*, 259:544 – 554.

Rocha, G., Castor, F., and Pinto, G. (2019). Comprehending energy behaviors of java i/o apis. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE.

Schuler, A. and Anderst-Kotsis, G. (2018). Towards a framework for detecting energy drain in mobile applications: an architecture overview. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 144–149.

Schuler, A. and Anderst-Kotsis, G. (2019). Examining the energy impact of sorting algorithms on android: An empirical study. In *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous '19, page 404–413, New York, NY, USA. Association for Computing Machinery.

Schuler, A. and Anderst-Kotsis, G. (2020). Characterizing energy consumption of third-party api libraries using api utilization profiles. ESEM '20, New York, NY, USA. Association for Computing Machinery.

Thoma, M., Cheng, H., Gretton, A., Han, J., Kriegel, H.-P., Smola, A., Song, L., Yu, P. S., Yan, X., and Borgwardt, K. M. Discriminative frequent subgraph mining with optimality guarantees. 3(5):302–318.

Ueda, Y., Ishio, T., Ihara, A., and Matsumoto, K. Mining Source Code Improvement Patterns from Similar Code Review Works. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 13–19. IEEE.

Wang, C., Yan, F., 0001, Y. G., and Chen, X. (2013). Power estimation for mobile applications with profile-driven battery traces. *ISLPED*, pages 120–125.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., and Zhang, D. Mining succinct and high-coverage API usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328. IEEE.

Wang, J., wu, G., Wu, X., and Wei, J. (2012). Detect and optimize the energy consumption of mobile app through static analysis. In *the Fourth Asia-Pacific Symposium*, pages 1–5, New York, New York, USA. ACM Press.

Wu, H., Yang, S., and Rountev, A. (2016). Static detection of energy defect patterns in Android applications. ACM.

Yan, X., Cheng, H., Han, J., and Yu, P. S. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 433–444. Association for Computing Machinery.

Zaki, M. J. Efficiently Mining Frequent Embedded Unordered Trees. 66(1-2):33–52.